Phonological CorpusTools
Documentation

Release 1.0.0

November 03, 2015
1 Introduction
   1.1 General Background .......................................................... 3
   1.2 Code and interfaces ......................................................... 4

2 Downloading and installing ...................................................... 7
   2.1 Download ................................................................. 7
   2.2 Windows Installer .......................................................... 7
   2.3 Mac Executable .............................................................. 7
   2.4 Linux / Fallback instructions ............................................. 8

3 Loading in corpora ........................................................................ 9
   3.1 Using a built-in corpus ...................................................... 9
   3.2 Creating a corpus ............................................................. 11
   3.3 Column-delimited files .................................................... 12
   3.4 Running Text ................................................................. 14
   3.5 Interlinear Text ............................................................... 15
   3.6 TextGrids ........................................................................ 16
   3.7 Other Standards .............................................................. 17
   3.8 Creating a corpus file on the command line ....................... 24
   3.9 Summary information about a corpus ................................... 25
   3.10 Subsetting a corpus ....................................................... 25
   3.11 Saving and exporting a corpus or feature file ...................... 26
   3.12 Setting preferences & options; Getting help and updates ......... 26

4 Example corpora ......................................................................... 29
   4.1 The example corpus ......................................................... 29
   4.2 The Lemurian corpus ....................................................... 29

5 Working with transcriptions and feature systems ............................. 33
   5.1 Required format of a feature file ....................................... 33
   5.2 Downloadable transcription and feature choices ..................... 34
   5.3 Using a custom feature system .......................................... 35
   5.4 Applying / editing feature systems ..................................... 36
   5.5 Edit inventory categories ................................................. 38
   5.6 Creating new tiers in the corpus ....................................... 41
   5.7 Adding, editing, and removing words, columns, and tiers ....... 45
   5.8 Phonological Search ...................................................... 47

6 Sound Selection ........................................................................... 51
Contents:
CHAPTER 1

Introduction

1.1 General Background

*Phonological CorpusTools* (PCT) is a freely available open-source tool for doing phonological analysis on transcribed corpora. For the latest information, please refer to the PCT website. PCT is intended to be an analysis aid for researchers who are specifically interested in investigating the relationships that may hold between individual sounds in a language. There is an ever-increasing interest in exploring the roles of frequency and usage in understanding phonological phenomena (e.g., [Bybee2001], [Ernestus2011], [Frisch2011]), but many corpora and existing corpus-analysis software tools are focused on dialogue- and sentence-level analysis, and/or the computational skills needed to efficiently handle large corpora can be daunting to learn.

PCT is designed with the phonologist in mind and has an easy-to-use graphical user interface that requires no programming knowledge, though it can also be used with a command-line interface, and all of the original code is freely available for those who would like access to the source. It specifically includes the following capabilities:

- Summary descriptions of a corpus, including type and token frequency of individual segments in user-defined environments;
- Calculation of the *phonotactic probability* of a word, given the other words that exist in the corpus (cf. [Vitevitch2004]);
- Calculation of *functional load* of individual pairs of sounds, defined at either the segment or feature level (cf. [Hockett1966]; [Surendran2003]; [Wedel2013]);
- Calculation of the extent to which any pair of sounds is *predictably distributed* given a set of environments that they can occur in, as a measure of phonological contrastiveness (cf. [Hall2009], [Hall2012]; [Hall2013a]);
- Calculation of the *Kullback-Leibler divergence* between the distributions of two sounds, again as a measure of phonological contrastiveness (cf. [Peperkamp2006]);
- Calculation of the extent to which pairs of words are *similar* to each other using either orthographic or phonetic transcription, and calculation of *neighbourhood density* (cf. [Frisch2004], [Khorsi2012]; [Greenberg1964]; [Luce1998]; [Yao2011]);
- Approximation of the *frequency with which two sounds alternate* with each other, given a measure of morphological relatedness (cf. [Silverman2006], [Johnson2010], [Lu2012]);
- Calculation of the *mutual information* between pairs of segments in the corpus (cf. [Brent1999]; [Goldsmith2012]); and
- Calculation of the *acoustic similarity* between sounds or words, derived from sound files, based on alignment of MFCCs (e.g., [Mielke2012]) or of logarithmically spaced amplitude envelopes (cf. [Lewandowski2012]).

The software can make use of pre-existing freely available corpora (e.g., the IPHOD corpus; [IPHOD]), which are included with the system, or a user may upload his or her own corpus in several formats. First, lexical lists with
transcription and token frequency information can be directly uploaded; such a list is what is deemed a “corpus” by PCT. Second, raw running text (orthographically and/or phonetically transcribed) can be uploaded and turned into lexical lists in columnar format (corpora) for subsequent analysis. Raw sound files accompanied by Praat TextGrids [PRAAT] may also be uploaded for analyses of acoustic similarity, and certain pre-existing special types of corpora can be uploaded natively (Buckeye [BUCKEYE], TIMIT [TIMIT]). Orthographic corpora can have their transcriptions “looked up” in a pre-existing transcribed corpus of the same language.

Phonological analysis can be done using built-in feature charts based on Chomsky & Halle [SPE] or Hayes [Hayes2009], or a user may create his or her own specifications by either modifying these charts or uploading a new chart. Feature specifications can be used to pull out separate “tiers” of segments for analysis (e.g., consonants vs. vowels, all nasal elements, tonal contours, etc.). PCT comes with IPA transcription installed, with characters mapped to the two feature systems mentioned above. Again, users may create their own transcription-to-feature mappings by modifying the existing ones or uploading a new transcription-to-feature mapping file, and several alternative transcription-to-feature mapping files are available for download.

Analysis can be done using type or token frequency, if token frequency is available in the corpus. All analyses are presented both on screen and saved to plain .txt files in user-specified locations.

The following sections walk through the specifics of downloading, installing, and using the various components of Phonological CorpusTools. We will do our best to keep the software up to date and to answer any questions you might have about it; questions, comments, and suggestions should be sent to Kathleen Currie Hall.

Version 1.1 (July 2015) differs from version 1.0.1 (March 2015) in three main areas:

1. Loading of corpora – The interface for corpus loading has been streamlined, and users have more options for adjusting the interpretation of transcriptions and columns as they initiate a corpus. Better support for interlinear glosses and TextGrids is also provided.

2. Specification of inventories, features, and environments – Inventories can now be displayed in IPA-like charts based on user-specified features. Feature selection in analysis functions has been streamlined and natural class selection is better supported. Environment selection is now iterative and more interactive.

3. Pronunciation variants – Analysis functions now provide users with options for how to handle pronunciation variants when they occur in a corpus.

Version 1.0 differs from the original release version (0.15, July 2014) primarily in its user interface; we switched the GUI from TK to QT and tried to reorganize the utility menus to be somewhat more intuitive. For example, the original release version had all segment inventory views in alphabetical order; segments are now arranged as closely as possible to standard IPA chart layouts (based on their featural interpretations). Additionally, we have added greater search and edit functions as well as some additional analysis tools (phonotactic probability, mutual information, neighbourhood density), and a greater ability to work with running text / spontaneous speech corpora.

1.2 Code and interfaces

PCT is written in Python 3.4, and users are welcome to add on other functionality as needed. The software works on any platform that supports Python (Windows, Mac, Linux). All code is available on the GitHub repository; the details for getting access are given in Downloading and installing.

There is both a graphical user interface (GUI) and a command-line interface for PCT. In the following sections, we generally discuss interface-independent aspects of some functionality first, and then detail how to implement it in both the GUI and the command line. All functions are available in the GUI; many, but not all, are currently available in the command line due to complications in entering in phonological transcriptions that match a given corpus in a command-line interface.

The command-line interface is accessed using command line scripts that are installed on your machine along with the core PCT GUI.
NOTE: If you did not install PCT on your computer but are instead running the GUI through a binary file (executable), then the command line scripts are not installed on your computer either. In order to run them, you will need to download the PCT source code and then find the scripts within the command_line subdirectory. These can then be run as scripts in Python 3.

The procedure for running command-line analysis scripts is essentially the same for any analysis. First, open a Terminal window (on Mac OS X or Linux) or a CygWin window (on Windows, can be downloaded at https://www.cygwin.com/). Using the “cd” command, navigate to the directory containing your corpus file. If the analysis you want to perform requires any additional input files, then they must also be in this directory. (Instead of running the script from the relevant file directory, you may also run scripts from any working directory as long as you specify the full path to any files.) You then type the analysis command into the Terminal and press enter/return to run the analysis. The first (positional) argument after the name of the analysis script is always the name of the corpus file.
PCT is currently available for Mac, PC, and Linux machines. It can be downloaded from PCT releases page using the following steps. Note that there are several dependencies that are pre-requisites before PCT can function properly. For Mac and Windows machines, we have created executable files that bundle most of the dependencies and the PCT software itself into a single package. Using these is the easiest / fastest way to get PCT up and running on your machine.

2.1 Download

1. Go to the PCT releases page.
2. Click on the link for your operating system with the highest number (= most recent version). As of June 2015, that is 1.1.

2.2 Windows Installer

1. NOTE: This method requires that you are running a 64-bit version of windows. You can check this in Control Panel -> System and Security -> System.
2. Download the file called “corpustools-1.1.0-amd64.msi” (or similar, for a more recent version), by clicking or right-clicking on the link. This is an installer program.
3. Run the downloaded installer program by double-clicking on it, wherever it has been saved locally.
4. PCT should now be available from your “Start” menu under “Programs.”
5. If you run into trouble, try the “Fallback” instructions in below.

2.3 Mac Executable

1. Download the file called ‘Phonological.CorpusTools-1.1.0.dmg’ by clicking or ctrl-clicking on the link.
2. Open the dmg file and drag the Phonological CorpusTools app into Applications.
3. Phonological CorpusTools is now available in your Applications, and can be opened as other applications. You may have to enable applications from third-party developers in your security settings.
4. If you run into trouble, try the “Fallback” instructions in below.
2.4 Linux / Fallback instructions

1. Dependencies: You’ll first need to make sure all of the following are installed. The third and fourth ones (NumPy and SciPy) are needed only for the Acoustic Similarity functionality to work.

1. Python 3.3 or higher
2. NumPy
3. SciPy
4. (NB: If you are on Windows and can’t successfully use the acoustic similarity module after installing NumPy and SciPy from the above sources, you may want to try installing them from precompiled binaries.)

2. Get the source code for PCT. Click on either the .zip or the .gz file on the PCT releases page or the GitHub repository, to download the zipped or tarball version of the code, depending on your preference.

3. After expanding the file, you will find a file called setup.py in the top level directory. Run it in one of the following ways:

1. Double-click it. If this doesn’t work, access the file properties and ensure that you have permission to run the file; if not, give them to yourself. In Windows, this may require that you open the file in Administrator mode (also accessible through file properties). If your computer opens the .py file in a text editor rather than running it, you can access the file properties to set Python 3.x as the default program to use with run .py files. If the file is opened in IDLE (a Python editor), you can use the “Run” button in the IDLE interface to run the script instead.

2. Open a terminal window and run the file. In Linux or Mac OS X, there should be a Terminal application pre-installed. In Windows, you may need to install Cygwin. Once the terminal window is open, navigate to the top level CorpusTools folder—the one that has setup.py in it. (Use the command ‘cd’ to navigate your filesystem; Google “terminal change directory” for further instructions.) Once in the correct directory, run this command: python3 setup.py install. You may lack proper permissions to run this file, in which case on Linux or Mac OS X you can instead run sudo python3 setup.py install. If Python 3.x is the only version of Python on your system, it may be possible or necessary to use the command python rather than python3.

4. Phonological CorpusTools should now be installed! Run it from a terminal window using the command pct. You can also open a “Run” dialogue and use the command pct there. In Windows, the Run tool is usually found in All Programs -> Accessories.
Loading in corpora

In order to use the analysis functions in PCT, you’ll first need to open up a corpus. When we say a “corpus” in PCT, we mean a file that has the following basic structure: a list of words with other possible information about each: e.g., its transcription, its frequency of occurrence, its lexical category, its syllable structure, etc. These are in columnar format; e.g., loaded from a CSV or tab-delimited text file. (See more at Required format of corpus.)

There are five possible ways of getting a corpus in PCT:

1. Use one of the built-in corpora to get started immediately. You can choose between two small, entirely invented corpora that have various parameters (see Example corpora) or use the Irvine Phonotactic Online Dictionary of English [IPHOD];
2. Use a corpus in the above format that is independently stored on your local computer;
3. Create a corpus from running text (e.g., straight transcriptions of speech or interlinear texts);
4. Create a corpus from Praat TextGrids [PRAAT];
5. Import a corpus from your own local copy of another standard corpus (currently, we support the Buckeye corpus [BUCKEYE] and the TIMIT corpus [TIMIT]).

Each of these options is discussed in more detail below.

3.1 Using a built-in corpus

To use a built-in corpus, simply go to the “File” menu and select “Load corpus...” from the list, which will open the “Load corpora” dialogue box.

The first time you want to use a built-in corpus, you’ll need to download it (from a Dropbox link accessed by PCT internally); you must therefore be connected to the internet to complete this step. To do so, click on “Download example corpora” from the right-hand menu. This will allow you to download either of the two example corpora (one is called “example” and the other called “Lemurian” (both are entirely made up; see Example corpora) and/or the IPHOD corpus [IPHOD]. Note that the version of the IPHOD corpus that is contained here has been altered from the freely downloadable version, in that it (1) does not have the derived columns and (2) has been re-formatted as a .corpus file for easy reading by PCT. It also contains only the following information: word, transcription, and token frequency (from the SUBTLEX corpus [SUBTLEX]). Please note that if you use the IPHOD corpus, you should use the following citation (see more on citing corpora and functions of PCT in Citing PCT and the algorithms used therein):


After the corpus has been downloaded, it appears in the lefthand side of the “Load corpora” dialogue box. Simply select the corpus and click on “Load selected corpus” at the bottom of the dialogue box. Once these corpora have been downloaded once, you don’t have to do so again; they will be saved automatically to your local system unless and until
you delete them. On subsequent loadings of the PCT software, you will still see these corpora listed in the lefthand side of the “Load corpora” dialogue box, as in the following diagram:

The example corpora and the included version of the IPHOD corpus include phonetic transcriptions in IPA, and are by default interpreted either using the feature system of [Mielke2012], which in turn is based on SPE features [SPE] [this is the default for the example corpus], or using the feature system suggested by [Hayes2009] [this is the default for the IPHOD corpus and the Lemurian corpus]. These systems are fully functional for doing subsequent analyses. Note, however, that this is a built-in functionality of these particular corpora, and does not allow you to use SPE or Hayes features with other corpora. To use SPE features with other corpora, or to change the feature system associated with a built-in corpus, you’ll need to download the actual feature files, as described in Working with transcriptions and feature systems. Features can be used for defining classes of sounds (e.g., creating separate tiers for different types of segments) and for defining environments (e.g., the environments in which segments might occur, for use in calculating their predictability of distribution).

The corpus may take several seconds to load, but will eventually appear; the following is the example corpus:
Summary information about a corpus gives more detail on how to find out summary information about your corpus. Typing a word or part-word in the “search” box takes you to each successive occurrence of that word in the corpus (hit “return” once to see the first instance; hit “return” again to see the second, etc.). Note that the “search” box searches only the “Spelling” column of the corpus. To do a phonological search, please use the “Phonological search” function under the “Corpus” menu (see detailed discussion in Phonological Search).

For more details on the structure of the Lemurian corpus, which has been built to show particular kinds of phenomena that may be of interest to PCT users, please see the section on The Lemurian corpus.

3.2 Creating a corpus

It is also possible to create a corpus within PCT. These can be pre-formatted columnar corpora or corpora that are compiled from running text, TextGrids, or special corpus formats. It may be helpful to first load the relevant feature system for your corpus into PCT, so that the transcriptions in your corpus can be interpreted; detailed instructions for
Section 3.3 Column-delimited files

If you have a corpus that is in the appropriate format (see Required format of corpus) and stored independently on your computer, you can read it in as a column-delimited file.

Once you have selected the file path and named the corpus (see Creating a corpus), make sure that the “Column-delimited file” tab is selected. PCT will automatically try to figure out what delimiter (e.g., comma, tab) is used to separate columns, but you can also enter it manually (e.g., a comma (,) or a tab (t)). Any symbol can be used; PCT will simply break elements at that symbol, so whatever symbol is used should be used only to delimit columns within the corpus.

If there is a column in the corpus that shows phonetic transcription, choose which feature system you would like to use. As noted above, in order for there to be feature systems to choose from, you must first have loaded them into PCT (Working with transcriptions and feature systems). If you haven’t yet added any, you may still import the corpus and then add them later.

On the right-hand side of the “Import corpus” dialogue box, you will see a “Parsing preview” window. This shows each of the columns in the corpus and allows you to specify particular parameters for each one. For details on this, please see the section on Parsing Parameters.

Once all selections have been made, click “Ok.” PCT will process the corpus (depending on how big it is, this may take a few minutes). It will then appear listed in the “Available corpora” window in the “Load corpus” dialogue box; you can select it and then click “Load selected corpus” to open it.

Note: the processed version of the corpus is stored in a .corpus file and automatically lives in a “CORPUS” folder in Documents / PCT / CorpusTools on your hard drive. See Preferences for information on how to change this.

See also Logging / Saving Parsing Parameters for information about how the parameters you picked when loading the corpus are temporarily saved.
Below is a picture of the “Import corpus” dialogue box set up to load in a .csv file with orthography, transcription, and frequency columns:

And here is the “Parsing settings” window of the transcription column:
3.4 Running Text

It is also possible to have PCT create a corpus for you from running text, either in orthographic or transcribed form. If the text is orthographic, of course, then segmental / phonological analysis won’t be possible, but if the text is itself
Once you have selected the file path or directory and named the corpus (see Creating a corpus), make sure that the “Running text” tab is selected. Select whether the text is spelling (“Orthography”) or phonetic transcription (“Transcribed”).

If the running text is transcribed, choose which feature system you would like to use. As noted above, in order for there to be feature systems to choose from, you must first have loaded them into PCT (Working with transcriptions and feature systems). If you haven’t yet added any, you may still import the corpus and then add them later.

If the running text is orthographic, and you have a corpus that contains transcriptions for the language of the running text, you can have PCT look up the transcriptions of words in that “support corpus.” This must be a corpus that has already been created in PCT. For example, you could first download the IPHOD corpus (see Using a built-in corpus) and then ask PCT to create a corpus from a .txt file that contains English prose, looking up each word’s transcription in the IPHOD corpus. You can specify that case should be ignored during lookup (e.g., to allow PCT to find the transcriptions of words even if they happen to be capitalized at the beginning of sentences in the running text).

At the right-hand side of the “Import corpus” dialogue box, you will see a “Parsing preview” window for the column of the corpus that will result from the running text. (The frequency of individual words in the text will be created automatically.) Please see the section on Parsing Parameters for details on how to make choices in this window.

Once all selections have been made, click “Ok.” PCT will process the corpus (depending on how big it is, this may take a few minutes). It will then appear listed in the “Available corpora” window in the “Load corpus” dialogue box; you can select it and then click “Load selected corpus” to open it.

Note: the processed version of the corpus is stored in a .corpus file and automatically lives in a “CORPUS” folder in Documents / PCT / CorpusTools on your hard drive. See Preferences for information on how to change this.

See also Logging / Saving Parsing Parameters for information about how the parameters you picked when loading the corpus are temporarily saved.

### 3.5 Interlinear Text

In addition to plain running text (Running Text), PCT also supports building corpora from interlinear texts, e.g., those with spelling and transcription on alternating lines. Interlinear texts may have any number of repeating lines.

Once you have selected the file path or directory and named the corpus (see Creating a corpus), make sure that the “Interlinear text” tab is selected.

PCT will start by automatically inspecting the text for characteristics that seem to repeat on particular sets of lines, to figure out how many lines there are per “unit.” E.g., a text that has spelling on the first line, transcription on the second, and glosses on the third will be automatically detected as having 3 lines per unit. The number can also be specified manually. Note that the text must maintain this pattern throughout; deviations will cause errors in how PCT reads in the data.

If the text is transcribed, choose which feature system you would like to use. As noted above, in order for there to be feature systems to choose from, you must first have loaded them into PCT (Working with transcriptions and feature systems). If you haven’t yet added any, you may still import the corpus and then add them later.

On the right hand side of the dialogue box, you’ll see a “Parsing preview” window which allows you to inspect each line of the gloss and specify how that line is interpreted. Please see the section on Parsing Parameters for details on how to make choices in this window.

Once all selections have been made, click “Ok.” PCT will process the corpus (depending on how big it is, this may take a few minutes). It will then appear listed in the “Available corpora” window in the “Load corpus” dialogue box; you can select it and then click “Load selected corpus” to open it.
3.6 TextGrids

PCT can also be used to create corpora from a collection of Praat TextGrids [PRAAT]. This is particularly useful for creating spontaneous speech corpora from recordings, especially if the transcription is based on what was actually spoken rather than on canonical forms of each word – PCT can keep track of the individual pronunciation variants associated with individual words (see Pronunciation Variants).

Once you have selected the file path or directory and named the corpus (see Creating a corpus), make sure that the “TextGrid” tab is selected (this should happen automatically if the file extension(s) is .TextGrid).

If any of the tiers in the TextGrid is a transcription tier, choose which feature system you would like to use. As noted above, in order for there to be feature systems to choose from, you must first have loaded them into PCT (Working with transcriptions and feature systems). If you haven’t yet added any, you may still import the corpus and then add them later.

If any of the tiers in the TextGrid is orthographic, and you have a corpus that contains transcriptions for the language of the text, you can have PCT look up the transcriptions of words in that “support corpus.” This must be a corpus that has already been created in PCT. For example, you could first download the IPHOD corpus (see Using a built-in corpus) and then ask PCT to create a corpus from a .txt file that contains English prose, looking up each word’s transcription in the IPHOD corpus. You can specify that case should be ignored during lookup (e.g., to allow PCT to find the transcriptions of words even if they happen to be capitalized at the beginning of sentences in the running text).

At the right-hand side of the “Import corpus” dialogue box, you’ll see a “Parsing preview” window. This will give you choices for how to parse each tier of the TextGrid, labelled with the original names of the tiers. Please see the section on Parsing Parameters for details on how to make choices in this window.

Once all selections have been made, click “Ok.” PCT will process the corpus (depending on how big it is, this may take a few minutes). It will then appear listed in the “Available corpora” window in the “Load corpus” dialogue box; you can select it and then click “Load selected corpus” to open it.
Note: the processed version of the corpus is stored in a .corpus file and automatically lives in a “CORPUS” folder in Documents / PCT / CorpusTools on your hard drive. See Preferences for information on how to change this.

See also Logging / Saving Parsing Parameters for information about how the parameters you picked when loading the corpus are temporarily saved.

### 3.7 Other Standards

Finally, PCT comes pre-equipped to handle certain other standard corpus types. At the moment, the only supported standards are the Buckeye corpus [BUCKEYE] and the TIMIT corpus [TIMIT]. You must obtain your own copy of either of these corpora through their usual means and store it locally; PCT simply gives you a way to easily open these corpora in the standard PCT format.

When selecting the corpus source, navigate to the directory where the Buckeye or TIMIT files are stored. PCT will automatically detect the format of files in the directory and select the “Other Standards” tab. Within that tab, it will also automatically select the file format.

If the text is transcribed, choose which feature system you would like to use. As noted above, in order for there to be feature systems to choose from, you must first have loaded them into PCT (Working with transcriptions and feature systems). If you haven’t yet added any, you may still import the corpus and then add them later. There is an option to download a Hayes-style feature system [Hayes2009] for the Buckeye corpus transcriptions.

At the right-hand side of the “Import corpus” dialogue box, you’ll see a “Parsing preview” window. This will give you choices for how to parse each part of the original corpus. Please see the section on Parsing Parameters for details on how to make choices in this window.

Once all selections have been made, click “Ok.” PCT will process the corpus (depending on how big it is, this may take a few minutes). It will then appear listed in the “Available corpora” window in the “Load corpus” dialogue box; you can select it and then click “Load selected corpus” to open it.

Note: the processed version of the corpus is stored in a .corpus file and automatically lives in a “CORPUS” folder in Documents / PCT / CorpusTools on your hard drive. See Preferences for information on how to change this.

See also Logging / Saving Parsing Parameters for information about how the parameters you picked when loading the corpus are temporarily saved.

#### 3.7.1 Required format of corpus

In order to use your own corpus, it must have certain properties. First, it should be some plain text file (e.g., .txt, .csv); it cannot, for example, be a .doc or .pdf file. The file should be set up in columns (e.g., imported from a spreadsheet) and be delimited with some uniform character (tab, comma, backslash, etc.). The names of most columns of information can be anything you like, but the column representing common spelling of the word should be called “spelling”; that with transcription should be called “transcription”; and that with token frequency should be called “frequency.” All algorithms for doing corpus analysis will assume these column names. If, for example, you were using a corpus that had different frequency columns for total frequency vs. the frequency of occurrence of the word in its lowercase form (cf. the SUBTLEX corpus), then whichever column is to be used for token frequency calculations should simply be labelled “frequency.”

### Parsing Parameters

This section outlines the choices that can be made in the “Parsing Preview” section of the import corpus dialogue box. In order for this section to be available, you need to have first started to import a corpus and selected a file, as described in the section on Creating a corpus.
1. **Name**: Specify the name of the column. If you are importing from a column-delimited file or a TextGrid with tiers, PCT will default to the name of the column / tier that is there. If you are reading from a running text or interlinear gloss file, and have specified that the file is either orthographic or transcribed, PCT will default to “Spelling” or “Transcription,” respectively. You may also manually enter the name.

2. **Annotation type**: Specify what type of information the column will contain. The default is simply a numeric or character column, depending on what type of information PCT automatically detects. **IMPORTANT**: You should specify which column you want PCT to treat as the “Orthography” and “Transcription” columns – without these named annotation types, some of the functions in PCT will not work, as they call on these particular types of columns.

3. **Word association**: Specify whether the information in the column should be associated with lexical items or should be allowed to vary within lexical items. Most types of information will be associated with lexical items (e.g., spelling, frequency). There are some kinds of information that do vary depending on the specific token, however, such as pronunciation variants of individual words or the identity of the speaker of an individual token. These are most likely to arise when creating a corpus from a TextGrid that has a tier for lexical items (e.g., based on spelling on canonical transcriptions) and then a separate tier that will show the characteristics of particular tokens (similar structures may be found with interlinear glosses). See also Pronunciation Variants and specifically Creating Pronunciation Variants.

4. **Delimiters and Special Characters**: For transcription and orthography columns, transcription and morpheme delimiters as well as any special characters are previewed at the right-hand side of the column information box. By clicking on “Edit parsing settings,” you can edit these, as follows:

   (a) **Example**: At the top of the “parsing” dialogue box, you will see an example of the entries in the column, to remind yourself of what sort of entries you are dealing with.

   (b) **Transcription delimiter**: If your transcriptions are delimited (i.e., have special characters that indicate segment breaks, as in [t.ai.d] for the word ‘tide,’ you can enter the delimiting character here). PCT will automatically search for this delimiter, but you may adjust it manually as well. For more on understanding complex transcriptions, see Complex transcriptions (Digraphs and other multi-character sequences).

   (c) **Morpheme delimiter**: If your transcriptions include a morpheme delimiter (i.e., have special characters that indicate morpheme breaks, as in [ri-du] for the word ‘redo,’ you can enter the delimiting character here. PCT will automatically search for this delimiter, but you may adjust it manually as well.

   (d) **Number parsing**: If PCT detects that there are numbers in the transcriptions, you have several options. Sometimes, numbers are simply used as alternatives for segmental transcriptions (e.g., [2] is used in the Lexique corpus [LEXIQUE] for IPA [ø]); in this case, simply select that they should be treated the “Same as other characters.” In other cases, numbers may be used to indicate tone (e.g., [Lei6.d.a1.k.s.eoi3] “profits tax” might be used in a Cantonese corpus like the Hong Kong Cantonese Adult Language Corpus [HKCAC] to indicate the tone number associated with each vowel). In this case, select that number parsing should be “Tone.” Finally, numbers might be used to indicate stress (e.g., [EH2.R.OW0.D.AY0.N.AE1.M.IH0.K] is the representation of the word “aerodynamic” in the IPHOD corpus [IPHOD] using CMU [CMU] transcriptions that include stress).

   (e) **Punctuation to ignore**: If there are punctuation marks in the file, and these have not already been specified as being used as either transcription of morpheme delimiters, then they will be listed as possible punctuation marks that PCT can ignore. Ignoring punctuation allows PCT to compile an accurate count of unique words, especially from running texts; for example, the words “example” and “example,” should be treated as two tokens of the same word, ignoring the comma at the end of the second one. Punctuation can be included, however; this might be desirable in a case where a punctuation symbol is being used within the transcription system (e.g., [!] used for a retroflex click). Each symbol can be ignored or included as needed. (Clicking on the symbol so that it is selected makes PCT IGNORE the symbol in the corpus creation.)

   (f) **Multicharacter segments**: See the discussion in Constructed multicharacter sequences in the section on Complex transcriptions (Digraphs and other multi-character sequences) for details.
3.7.2 Complex transcriptions (Digraphs and other multi-character sequences)

There is no way for PCT to know automatically when a single sound is represented by a sequence of multiple characters – e.g., that the digraphs [ai], [t], [xw], [p'], [f], and [i:] are intended to represent single sounds rather than sequences of two sounds. There are currently three possible ways of ensuring that characters are interpreted correctly:

1. **One-to-one transcriptions**: The first way is to use a transcription system with a one-to-one correspondence between sounds and symbols, such as DISC. If you need to create a novel transcription system in order to accomplish this (e.g., using [A] to represent [ai] and [2] to represent [t], etc.), you may certainly do so; it is then necessary to create a novel feature file so that PCT can interpret your symbols using known features. See detailed instructions on how to do this in Downloadable transcription and feature choices. The word tide in American English might then be transcribed as [2Ad]. This is a relatively easy solution to implement by using find-and-replace in a text editing software, though it does result in less easily human-readable transcriptions.

2. **Delimited transcriptions**: The second way is to use a standard transcription system, such as IPA, but to delimit every unitary sound with a consistent mark that is not otherwise used in the transcription system (e.g., a period). Thus the word *tide* in American English might be transcribed in IPA as [t.a.i.d], with periods around every sound that is to be treated as a single unit. When creating the corpus, PCT will give you the option of specifying what the character is. PCT will then read in all elements between delimiting characters as members of a single “segment” object, which can be looked up in a standard feature file (either an included one or a user-defined one; see Using a custom feature system). This solution makes it easy to read transcribed words, but can be more labour-intensive to implement without knowledge of more sophisticated searching options (e.g., using regular expressions or other text manipulation coding) to automatically insert delimiters in the appropriate places given a list of complex segments. A first pass can be done using, e.g., commands to find “ai” and replace it with “.ai” – but delimiters will also have to be added between the remaining single characters, without interrupting the digraphs.

3.7.3 Constructed multicharacter sequences

The third option is to tell PCT what the set of multicharacter sequences is in your corpus manually, and then to have PCT automatically identify these when it creates the corpus. This can be done by editing the parsing settings for a column during the import of a corpus. In the “Import corpus” dialogue box, there is an option to edit the parsing settings for each column in the corpus. At the bottom there is an option for listing multicharacter segments in the corpus. You may enter these manually, separated by commas, or choose “Construct a segment” to have help from PCT. If you are entering them manually, you may copy and paste from other documents (e.g., if you have created a list of such sequences independently). If you choose “Construct a segment,” PCT will scan the selected file for single characters and present these to you as options for constructing multi-character segments from.

For example, in the following box, all of the single characters in a Gitksan text file are presented, and can be selected sequentially to create the appropriate multi-character segments. This method is somewhat more labour-intensive in terms of knowing ahead of time what all the multi-character segments are and being able to list them, but ensures that all such occurrences are found in the text file. Note, however, that if there’s a **distinction** to be made between a sequence of characters and a digraph (e.g., [tf] as a sequence in *great ship* vs. as an affricate in *grey chip*), this method will be unable to make that distinction; all instances will be treated as multi-character segments. Each multi-character segment can be as long as you like. If there are shorter sequences that are subsets of longer sequences, PCT will automatically look for the longer sequences first, and separate them out; it will then scan for the shorter sequences. E.g., it will search for and delimit [ts’] before it searches for [ts], regardless of the order in which the sequences are entered. Note that the list of multicharacter segments is **temporarily** saved in a log file for the current PCT session; you may want to open the log file and copy and paste the set of multicharacter segments to a new file for your own later use. For instance, this is useful for times when you may want to re-create the corpus with different settings or formatting and don’t want to have to re-construct all the multi-character sequences by hand, as the entire list of multicharacter segments can simply be copy-pasted into the parsing dialogue box. See details on this feature in the Logging / Saving Parsing Parameters section.
Logging / Saving Parsing Parameters

When you import a new corpus into PCT, there are many parameters that you choose, such as the name of the corpus, the type of corpus, the various delimiters, ignored punctuation, multicharacter sequences, etc. – see Creating a corpus. Sometimes, you may find it necessary to tweak the parameters originally chosen once you’ve imported a corpus and loaded it in (for instance, you might realize that you forgot a particular digraph when you were specifying multicharacter segments). PCT automatically keeps a temporary log of the import settings on any given session. You can, for example, copy and paste the set of digraphs from this log to save and re-use in future sessions, rather than having to re-create them from scratch just to add a new one in. To limit the size of the log file, though, PCT will overwrite it every time PCT is re-launched with a new corpus import, so any information that is important should be saved from the log file manually.

To access the log file, go to the directory where your PCT files are stored. By default, this is Documents / PCT / CorpusTools, but you can change this location; to do so, see the Preferences section. Within this directory, click on the “log” folder; you will see a pct_gui.log file. This can be opened in any text editor. Information from this file can be copied and pasted into a separate document that can be saved for future reference.

Here’s an example of the log file after importing a 3-line interlinear gloss file of Gitksan:

```
The following shows an example of a transcribed Gitksan story transformed into a (small!) corpus (with grateful acknowledgement to Barbara Sennott and the UBC Gitksan language research group, headed by Lisa Matthewson & Henry Davis, for granting permission to use this text):

1. The original transcribed story:
```
2. The transcription delimited with periods to show unitary characters:

```
lukux:s-x=r-s c’i:c’ qen=t ak’i tku:ix=m hanaq’-t
?i: qo-T=q=t kat ?i: wa-tx=s c’i:c’ wa-tx=t
ne-ti=t na: co wil wil xsa pagaqt-ia:
?i: la: ?as-m=q gapt=t k’x=’u=i: ho: wa-tx=s c’i:c’
?aq=t ne-ti: t’a: tan an= nakε-ε=t ak’i tku:ix=m hanaq’
?i: getx=q= qo:t-t tom wil lu: qo:T=q=t wilnoa:t’s=q=t-t
?i: hec’am=ho: k’ila ta:wi=q=t k’iy=’u=i: ho: k’iy=’u=i:
la: ?aim= qo-A=x=’a-ot ?i: la: hukux=q= qo-so=’a ak’u t’ix-=m hanaq’ ki tom naks-ε=t
?i: ho: wa-tx=s C’i:c’ wa-tx=t
?i: k’iy=’u=i: sa ?i: pa=’lk=k s’-tx=’a wi: ps’=a kat qo=:q c’im qon-qan
?i=t ka=t k’u t’ix-=m hanaq’ ki ?i=t hox=c=t: ka=q=t ak’u t’ix-=m hanaq’ nιt
?i: na=wi:T=q=t-t T?i: na=ni:nisxw=t-t
?i: sam lu= ps’=m=i qo:t-s c’i:c’ tom wil qani=’i:la yc=q wilnoa:t’o:’t-t
nιt wilw so=t’a=na=t ?an-spa=’ax= tiya=q t qo:t-γ
lis-x=’u
```

3. The dialogue boxes for creating the corpus from text. Note that hyphens and equal signs, which delimit morphological boundaries in the original, have been ignored during the read-in. A feature system called gitksan2hayes_delimited, which maps the delimited transcription system used in this example to the features given in [Hayes2009], has already been loaded into PCT (see Using a custom feature system), and so is selected here. In this case, the multicharacter segments are indicated manually.
Example: ii yukw 'nim leseexw=hl wila

Transcription delimiter

Morpheme delimiter

Number parsing  No numbers

Punctuation to ignore

Check all  Uncheck all

Multicharacter segments

hl,kw,kw',gw,xw,'w','y','n','m,ts,ts',t',k',p',aa

Construct a segment

Ok  Cancel
4. The resulting corpus, ready for subsequent analysis:

The corpus appears on the left, in the familiar columnar format. The original text of the corpus appears at the right. Right-clicking on a word in the corpus list gives you the option to “Find all tokens” in the running text; these words will be highlighted. Similarly, right-clicking a word in the running text gives you the option to “Look up word,” which will highlight the word’s entry in the corpus list.

Here is an example of creating a corpus based on three .TextGrid files from the Corpus of Spontaneous Japanese [CSJ]. Note that the hovering over the box labelled “Mouseover for included files” shows a list of the names of the files in the chosen directory. In the “parsing preview” window, each set of boxes corresponds to one tier of the TextGrids, and the original name of the TextGrid is shown at the top (e.g., “word,” “seg”). Note that here, the orthographic tier is associated with the lexical item, while the transcription tier is allowed to vary within lexical item, such that pronunciation variants are kept track of.

Once the TextGrids have been processed, they appear in a window such as the following. The regular corpus view is in the centre, with frequency counts aggregated over the entire set of speakers / TextGrids. Note that the transcription column may be blank for many words; this is because in spontaneous speech, the citation / spelled words often have multiple different transcribed forms in the corpus itself. To see these various transcriptions, right-click on any word in the corpus and select “List pronunciation variants.” A new dialogue box will pop up that shows the individual pro-
nunciation variants that occur in the corpus for that word, along with their token frequencies. (See also Pronunciation Variants.)

In this example, each TextGrid is interpreted as belonging to a different speaker, and these individual speakers are listed on the left. Clicking on one of the speakers shows the transcript of that speaker’s speech in a box on the right. This is not a corpus, but rather a sequential listing of each word that was extracted, along with the transcription and the timestamp of the beginning of that word in the TextGrid. Right-clicking on a word in this list will give you the option to look up the word’s summary entry in the corpus itself, which appears in the centre. Right-clicking a word in the overall corpus will give you the option to “Find all tokens” of that word in the transcriptions, where they will simply be highlighted.

3.8 Creating a corpus file on the command line

In order to create a corpus file on the command line, you must enter a command in the following format into your Terminal:

```
pct_corpus TEXTFILE FEATUREFILE
```

...where TEXTFILE is the name of your input text file and FEATUREFILE is the name of your feature file. You may specify file names using just the file name itself (plus extension) if your current working directory contains the files; alternatively, you can specify the full path to these files. Please do not mix short and full paths. You may also use command line options to change the column delimiter character or segment delimiter character from their defaults (\t and ’’, respectively). Descriptions of these arguments can be viewed by running `pct_corpus -h` or `pct_corpus --help`. The help text from this command is copied below, augmented with specifications of default values:

Positional arguments:

-`-h`  
  `--help`
  Show this help message and exit

-`-d` DELIMITER
  `--delimiter` DELIMITER
  Character delimiting columns in input file, defaults to \t

-`-t` TRANS_DELIMITER
  `--trans_delimiter` TRANS_DELIMITER
  Character delimiting segments in input file, defaults to the empty string
EXAMPLE:

If your pre-formatted text file is called mytext.txt and your features are hayes.feature, and if mytext.txt uses ; as column delimiters and . as segment delimiters, to create a corpus file, you would need to run the following command:

```
pct_corpus mytext.txt hayes.feature -d ; -t .
```

### 3.9 Summary information about a corpus

Phonological CorpusTools allows you to get summary information about your corpus at any time. To do so, go to “Corpus” / “Summary.”

1. **General information**: At the top of the “Corpus summary” dialogue box, you’ll see the name of the corpus, the feature system currently being used, and the number of word types (entries) in the corpus.

2. **Inventory**: Under the “Inventory” tab, there will generally be three sections, “Consonants,” “Vowels,” and “Other.” (Note that this assumes there is an interpretable feature system being used; if not, then all elements in the inventory will be shown together.) If there is a feature system in place, consonants and vowels will be arranged in a manner similar to an IPA chart. (For more on how to edit this arrangement, see *Edit inventory categories.*) Any other symbols (e.g., the symbol for a word boundary, #) will be shown under “Other.”

   (a) **Segments**: Clicking on any individual segment in the inventory will display its type and token frequency in the corpus, both in terms of the raw number of occurrences and the percentage of occurrences.

3. **Columns**: Under the “Columns” tab, you can get information about each of the columns in your corpus (including any that you have added as tiers or other columns; see *Adding, editing, and removing words, columns, and tiers*). The column labels are listed in the drop-down menu. Selecting any column will show you its type (spelling, tier, numeric, factor) and other available information. Tier columns (based on transcriptions) will indicate which segments are included in the tier. Numeric columns will indicate the range of values contained.

Once you are finished examining the summary information, click “Done” to exit.

### 3.10 Subsetting a corpus

It is possible to subset a corpus, creating new corpora that have only a portion of the original corpus. For example, one might want to create a subset of a corpus that contains only words with a frequency greater than 1, or contains only words of a particular part of speech or that are spoken by a particular talker (if such information is available). The new subsetted corpus will be saved and made available to open in PCT as simply a new corpus.

To create a subset, click on “File” / “Generate a corpus subset” and follow these steps:

1. **Name**: Enter the name for your new corpus. The default is to use the name of the current corpus, followed by “_subset,” but a more informative name (e.g., “Gitksan_nouns”) may be useful.

2. **Filters**: Click on “Add filter” to add a filter that will be used to subset the corpus. You can filter based on any numeric or factor tier / column that is part of your corpus. For a numeric column (e.g., frequency), you can specify that you want words that have values that are equal to, greater than, less than, greater than or equal to, less than or equal to, or not equal to any given value. For a factor column (e.g. an abstract CV skeleton tier), you can add as many or as few levels of the factor as you like.

3. **Multiple filters**: After a filter has been created, you can choose to “Add” it or “Add and create another” filter. The filters are cumulative; i.e., having two filters will mean that the subset corpus will contain items that pass through BOTH filters (rather than, say, either filter, or having two subsets, one for each filter).

4. **Create subset**: Once all filters have been selected, click on “Create subset corpus.” You will be returned to your current corpus view, but the subsetted corpus is available if you then go to “File” / “Load corpus...” – it will

---

**3.9. Summary information about a corpus** 25
automatically be added to your list of available corpora. Note that the subset corpus will automatically contain any additional tiers that were created in your original corpus before subsetting.

3.11 Saving and exporting a corpus or feature file

If “Auto-save” is on (which is the default; see Preferences), most changes to your corpus (adding a feature system, words, tiers, etc.) will be saved automatically and will be available the next time you load the corpus in PCT. Some changes are not automatically saved (removing or editing word entries), even if Auto-save is on, to prevent inadvertent loss of information. If you have made changes that have not been automatically saved, and then quit PCT, you will receive a warning message indicating that there are unsaved changes. At that point, you may either choose “Don’t save” (and therefore lose any such changes), “Save” (to save the changes in its current state, to be used the next time it is loaded into PCT), or “Cancel” (and return to the corpus view).

It is also possible to export the corpus as a text file (.txt), which can be opened in other software, by selecting “File” / “Export corpus as text file” and entering the file name and location and the column and transcription delimiters. (Note: use t to indicate a tab.) You can also choose whether and how to export pronunciation variants, if there are any in the corpus (see Pronunciation Variants and the subsection Exporting Pronunciation Variants: for more details).

Similarly, the feature system can also be exported to a .txt file by selecting “File” / “Export feature system as text file” and selecting the file name and location and the column delimiter. See more about the utility of doing so in Working with transcriptions and feature systems.

See also information about the temporary log file that is created when a new corpus is imported by going to Logging / Saving Parsing Parameters; this file has information about the various Parsing Parameters that were chosen in the creation of any given corpus.

3.12 Setting preferences & options; Getting help and updates

3.12.1 Preferences

There are several preferences that can be set in PCT. These can be selected by going to “Options” / “Preferences....” The following are available:

1. Storage:

   (a) File location: By default, PCT will save corpus, feature, and result files to your local “Documents” directory, which should exist under the default settings on most computers. When saving a particular output file, you can generally specify the particular storage location as you are saving. However, it is also possible to change the default storage location by changing the file path in this dialogue box. You may enter the path name directly, or select it from a system window by selecting “Choose directory...”.

   (b) Auto-save: By default, PCT will automatically save changes to a corpus (e.g., if you have updated a feature system, added a tier, etc.). De-select this option if you prefer to manually save such changes (PCT will prompt you before closing without saving). Changes to word entries (removing or editing a word) are NOT auto-saved and should be saved manually if you want them to be saved; again, PCT will prompt you to save in these cases before exiting. Once Auto-save is deselected, PCT will remember that this is your preference for the next time you open the software - it will not automatically get turned back on.

2. Display: By default, PCT will display three decimal places in on-screen results tables (e.g., when calculating predictability of distribution or string similarity, etc.). The number of displayed decimal places can be globally changed here. Note that regardless of the number specified here, PCT will save results to files using all of the decimal places it has calculated.
3. **Processing**: Some calculations consume rather a lot of computational resources and can be made faster by using multiprocessing. To allow PCT to use multiprocessing on multiple cores when that is possible, select this option and indicate how many cores should be used (enter 0 to have PCT automatically use ¾ of the number of cores available on your machine).

### 3.12.2 Help and warnings

When using PCT, hovering over a dialogue box within a function will automatically reveal quick ToolTips that give brief information about the various aspects of the function. These can be turned on or off by going to “Options” / “Show tooltips.”

PCT will also issue certain warnings if various parameters aren’t met. It is possible to turn warning messages off by going to “Options” / “Show warnings.”

There is also extensive documentation for all aspects of PCT (of which the current text is part). There are several options for accessing this information:

1. In the main PCT window (i.e., when viewing your corpus), click on “Help” from the “Help” menu. This will take you to the main help file, from which you can navigate to other specific topics.
2. Go to the online PCT documentation to get access to the help files online.
3. Go to the PCT website and download a .pdf copy of the entire help file for off-line use.
4. While working in PCT, most dialogue boxes have options at the lower right-hand corner that say either “Help” or “About...” (e.g., “About functional load...”). Clicking this button will pull up the relevant help file.

### 3.12.3 Copying and pasting

It is possible to highlight the cells in any table view (a corpus, a results window, etc.) and copy / paste a tab-delimited string version of the data into another program (e.g., a spreadsheet or text editor) using your standard copy & paste keyboard commands (i.e., Ctrl-C and Ctrl-V on a PC; Command-C and Command-V on a Mac).

### 3.12.4 Updates

To manually see whether there is a more recent version of PCT available for download, click on “Help” / “Check for updates...”.

To be automatically notified of new versions of PCT or any other major news that is relevant to all users, please sign up for the PCT mailing list, available from the PCT website.
There are two example corpora that can be used in PCT that consist of entirely made-up data. The first is a very small corpus called the “example” corpus; the second is a slightly larger corpus called “Lemurian.” Their purpose is to serve as “practice” corpora, allowing the user to become familiar with PCT while working with an unfamiliar language.

4.1 The example corpus

The example corpus was included in the earliest versions of PCT and has a few useful patterns for testing out analysis functions. For practical purposes, it is essentially superseded by the Lemurian corpus, which is more complex, but we continue to include it since many of the “help” examples are based on its contents.

4.1.1 Inventory:

Stops: /t/
Nasals: /m, n/
Fricatives: /s, f/
Vowels: /i, e, u, o, a/

4.1.2 Phonological restrictions:

1. /e/ and /o/ are allophones of [i] and [u], respectively, which occur only immediately before a nasal consonant.
2. Non-low vowel harmony with blocking nasals: there are no sequences of a non-low front vowel followed (at any distance) by a non-low mid vowel or vice-versa without an intervening nasal consonant.
3. Left-spreading /j/-dominant sibilant harmony: there are no sequences of [s] followed (at any distance) by [ʃ].
4. Purely CV syllable structure.

4.2 The Lemurian corpus

The Lemurian corpus is generated by a Python script such that it will contain patterns that are easily detected with the analysis functions in PCT. It can be generated to any size. The specific instantiation of the Lemurian corpus that is available in PCT is generated with 30 words. It is a bit larger than the example corpus and has a few specific
characteristics that users may find useful. Not all of the following may be particularly visible in this sample of Lemurian, but these are the guidelines along which the corpus is built:

### 4.2.1 Inventory:

**Stops:** /p, b, t, d, k/

**Nasals:** /m, n/

**Fricatives:** /f, s, x/

**Liquids:** /l, r/

**Glides:** /j, w/

**Vowels:** /i, e, u, o, a/

### 4.2.2 Phonotactics:

Words in the corpus can be anywhere from 1 to 5 syllables long. Lemurian has a maximum syllable of C1C2VC3 with the following phonotactics:

1. Codas and onsets are always optional.
2. C1 can be any consonant in the inventory.
3. C2 can be a glide, a stop, or a nasal. Glides can occur after any consonants.
4. C2 can only be a stop or a nasal if C1 is a fricative.
5. C3 if present must be a nasal.

Lemurian has front/back vowel harmony, and a word can only contain vowels from one of those categories. Front vowels are /i, e/, back vowels are /u, o/. The vowel /a/ is neutral and can appear with vowels from either set.

### 4.2.3 Other phonological patterns:

1. The sound [z] occurs only as an allophone of /s/ between vowels, i.e. s -> z / V_V
2. Voiced and voiceless stops only contrast in C1 position of a syllable. If a stop appears in C2 (following a fricative) then it is necessarily a voiceless stop.
3. Coronals have a high functional load.
4. Word frequencies are randomly generated, so there is no guarantee about any sound or sound sequence being more or less common.

### 4.2.4 Orthography vs. transcription:

The Lemurian corpus contains a number of intentional mismatches between the spelling system and the transcription system. This allows users to test out the differences between selecting spelling and transcription for some of the analysis functions, e.g. string similarity.

1. All of the transcription symbols match the orthographic symbols, except for /x/ which is written as “h.”
2. Coda nasals are not distinguished in writing. Both use the symbol ‘N’. This obscures some minimal pairs.
3. If a syllable starts with /ju/ or /wu/, the glide is not written.
4. The allophone [z] is written as “s"

For example, the word [junxwa] would be spelled “uNhwa”
In order to do phonological analysis, PCT needs to know what segments exist in a corpus, and what features are assigned to each segment. There are a variety of built-in transcription systems and feature systems available, but users can also define their own for custom use. Transcription and feature systems are essentially packaged together as .txt files in the form of a spreadsheet, where particular transcription symbols are mapped to a set of features (described below in Required format of a feature file). In general, however, feature systems (i.e., files containing transcriptions and their features) must be explicitly loaded into PCT before they are available for use. Thus, it makes sense to start by loading in at least one such system before attempting to work with corpora.

5.1 Required format of a feature file

As mentioned above, transcription and feature systems are packaged together as .txt files in the form of a delimited spreadsheet. The first column in the file lists all the transcription symbols that are to be recognized. Note that this column must be labeled with the name “symbol” in order for PCT to correctly read in the file. Every symbol used in the corpus needs to be in this column. Subsequent columns list individual features that are to be used. Each cell in a row gives either a + or – for each feature, indicating what value the initial symbol in that row has for that feature; a 0 or n can also be used, to indicate that a particular feature is not defined for a given segment (0 is the default in the built-in hayes system; n is the default in the built-in spe system). The following shows an example; keep in mind that this is in fact a tab-delimited .txt file, but the names in the first row are longer than any of the values in subsequent rows, so the alignment is visually misleading. For example, the first row containing symbols contains the symbol [p], which is designated as [-syllabic], [+consonantal], [-sonorant], [-continuant], etc., although the column names aren’t aligned with the feature values visually.
5.2 Downloadable transcription and feature choices

Currently, the built-in transcription systems that are usable are IPA, ARPABET (used for the [CMU] dictionary), XSAMPA, CPA, CELEX, DISC, Klatt, and Buckeye. These transcription systems can be associated with either the features as laid out in [Mielke2012], which in turn are based on [SPE], or as laid out in [Hayes2009]. Each of these transcription-to-feature mappings is laid out as above in a .txt file that can be downloaded from within PCT. The former system is called “spe” for short within PCT, while the latter is called “hayes.”

To download one of these systems, click on “Corpus” / “Manage feature systems...” and follow these steps:

1. **Download**: Click on “Download feature systems” to open up the relevant dialogue box.

2. **Transcription**: Select which of the transcription systems you want (IPA, ARPABET, XSAMPA, CPA, CELEX, DISC, Klatt, or Buckeye).

3. **Feature system**: Select which set of features you would like to map the transcription symbols to (SPE or Hayes).

4. **Saving**: Click “OK” to have PCT load in the selected feature file (you must be connected to the internet to have this functionality). The newly downloaded feature file will now appear in your “Manage feature systems” dialogue box, and is available for all subsequent use of PCT unless and until you delete it (done by selecting the system and clicking “Remove selected feature system”). Click “Done” to return to the regular corpus analysis window.

See *Applying / editing feature systems* for more information about applying / editing feature systems in conjunction with corpora.

---

Note that the original [Hayes2009] system does not include diphthongs. We have included featural interpretations for common English diphthongs using two additional features, [diphthong] and [front-diphthong]. The former has a [+] value for all diphthongs, a [-] value for all vowels that are not diphthongs, and a [0] value for non-vowels. The latter references the end point of a diphthong: [ai], [ei], and [AI] are [+front-diphthong], [ao] and [oo] are [-front-diphthong]. All other segments are left unspecified for this feature. Other vowel features for diphthongs are specified based on the first element of the diphthong: e.g., all of [ai], [ei], [AI], [ao], and [oo] are [-high]; of these five, only [ai] and [ao] are [+low]; only [ei] is [+front]; only [oo] and [AI] are [+back]; only [oa] and [AI] are [+round].
5.3 Using a custom feature system

In addition to using one of the built-in feature systems, you can design your own transcription-to-feature mapping, of the format specific in *Required format of a feature file*.

5.3.1 Loading a custom feature system

Once you have a feature file in the required format (see *Required format of a feature file* and *Modifying an existing feature system’s text file*), go to “Corpus” / “Manage feature systems...” to load it in. Select “Create feature system from text file” and the “Import feature system” dialogue box will open.

1. **File selection**: Specify the file by entering its directory path or by selecting it using the “Choose file...” button.

2. **Transcription system**: Indicate which transcription system this is a feature file for. (For example, you can create a new feature file for existing IPA transcriptions.) If this is a brand-new system for PCT, i.e., a new transcription system being associated with features, then select “Custom” from the dropdown menu. Then, enter a name for the transcription system in the box.

3. **Feature system**: Indicate which feature system is being used (e.g., is this a case of assigning existing SPE features to a new transcription system?). If this is a brand-new set of features, then select “Custom” from the dropdown menu. Then, enter a name for the feature system in the box.

**Note**: For both existing transcription and feature systems, you still need to include both the transcriptions and the features in the .txt file itself; you can simply indicate here in PCT that these transcriptions and / or features are identical to ones that are already extant in the system, so that they can be used / interpreted consistently. The name of the transcription / feature system file in PCT will conventionally be transcription2features (e.g., ipa2hayes for IPA symbols interpreted using Hayes features), so it’s useful to be consistent about what the names are.

4. **Delimiter**: Indicate what the column delimiter in the custom file is. The default, tab, is indicated by \t.

Click “OK,” and the feature system should now appear in your “Available feature systems” window. Click “Done.” See *Applying / editing feature systems* for more information about applying the feature system to corpora. The image below shows the dialogue box used to load in the custom, tab-delimited feature file for interpreting the custom “gitksan” transcription system using Hayes features.
5.3.2 Modifying an existing feature system’s text file

A custom system can be created from scratch, following the format described in Required format of a feature file. It is probably easier, however, to create a new system by modifying an existing system’s file. While this can be done to a certain extent within PCT itself (see Applying / editing feature systems), large-scale changes are best done in separate text-editing software. To do so, you’ll need to start with an existing file, which can be obtained in one of two ways:

1. Through PCT: Download one of the built-in feature systems (Downloadable transcription and feature choices) and apply it to your corpus (Applying / editing feature systems). Then go to “File” / “Export feature system as text file...” and save the file locally.

2. Online: You can directly download .txt files with the currently available feature systems from the PCT SourceForge page, under “Files.” One advantage to this method is that there may be additional feature files that are created as .txt files and made available online before they are packaged into the next release of the downloadable software itself.

Once you have the file, open it in whatever software you prefer (e.g., TextEdit, OpenOffice, etc.); it may be easiest to import it into a spreadsheet reader (e.g., Excel, Calc, etc.) in terms of legibility. You can then add new symbols to the first column, feature specifications in the subsequent columns, new feature names, etc., etc. Remember that the name of the first column must always be “symbol” in order for PCT to read the file. Save the new file as a CSV or tab-delimited .txt file, and load it following the instructions in Loading a custom feature system.

5.4 Applying / editing feature systems

Once a feature system has been loaded into PCT (Downloadable transcription and feature choices, Using a custom feature system), it is available for use with a corpus. To do so, first load in a corpus (Loading in corpora); if you are using a custom corpus or creating a corpus from text, you can select the feature system you want to use during the loading. Once a corpus has been loaded (with or without a feature system), you can edit the system by clicking on “Features” / “View / change feature system....” The following options are shown:
1. **View system**: At the top of the “Edit feature system” dialogue box, you’ll see the current transcription and feature system being used, assuming one has been selected. The first column shows the individual symbols; subsequent columns give the feature specifications for each symbol. Clicking on the name of any column sorts the entire set by the values for that feature; clicking again flips the sort order based on that same column.

2. **Change transcription and feature systems**: If there is no feature system already specified, or if you would like to change the transcription or feature system, use the dropdown menus under “Change feature systems” to select from your currently available systems. If no system is available, or the system you want to use is not available, go back to Downloadable transcription and feature choices or Using a custom feature system to learn how to load feature systems in to PCT. Be sure to click on “Save changes to this corpus’s feature system” after selecting a new feature system in order to actually apply it to the corpus.

3. **Modify the feature system**: You can modify the current feature system directly within PCT. There are three options.
   
   (a) **Add segment**: To add a new segment and its associated feature values to the current feature system, click on “Add segment.” A new dialogue box will open up, with a space to input the symbol and dropdown menus for all of the features expected in the current system. You can also specify that all features should be set to a particular value, and then change / edit individual features as needed. Simply fill in all the values and click “OK”; the symbol and features will be added to the feature system.

   (b) **Edit segment**: To change the feature specifications of an existing segment, click on the row containing that segment and then on “Edit Segment.” Then use the resulting dialogue box to change the feature specifications.

   (c) **Add feature**: To add an additional feature to the current system, click on “Add feature.” Enter the name of the feature in the dialogue box, select the default value that all segments will have for this feature, and hit “OK.” The feature will be added to all the segments in the corpus, with the default value. To change the value of the feature for each segment, simply click on the segment and then use the “Edit segment” functionality described above; the new feature will automatically be added to the dialogue box for each segment.

   **Warning**: Be sure to click on “Save changes to this corpus’s feature system” after adding a new segment or feature, or editing the feature specifications of a segment, in order to actually apply the changes to the corpus.

4. **Corpus inventory coverage**: There are two tools built in to help you check the coverage in your corpus.
   
   (a) **Extraneous symbols**: The built-in feature systems are fairly extensive, and may include symbols for sounds that do not occur in your corpus. Click on “Hide all segments not used by corpus” to remove such segments from the viewing window. (This does NOT remove them from the feature system itself; it just de-clutters your view of the system.) To revert back to the full system, simply click on “Show full feature system.”

   (b) **Corpus coverage**: It’s possible that there are symbols used in your corpus that are not covered in whatever feature system you have selected. To find out, click on “Check corpus inventory coverage.” A new window will appear that either confirms that all symbols in the corpus are mapped onto features, or lists the symbols that are currently lacking. If there are symbols that are missing, you’ll need to add them before doing phonological analysis on the corpus. You can do so in two ways: (1) adding them within the PCT interface, following the instructions under “Modify the feature system,” immediately below; or (2) changing the feature system itself by editing the .txt file and reloading it (more information given in Modifying an existing feature system’s text file).

Below is an example of the “Edit feature” system dialogue box, loaded with the “ipa2spe” transcription and feature file:
5. **Display options**: The standard view (shown below) is to display the segments and features as a matrix. One can also select “tree” view, which allows one to see a list of the segments included in the transcription system, organized by phonetic characteristics (but currently without all of their feature specifications).

### 5.5 Edit inventory categories

There are many instances in which PCT needs to display the inventory of the corpus in order for sounds to be selected for search or analysis. The default is to display segments in alphabetical order, which is not necessarily particularly intuitive as far as a linguist’s ability to interact with the list. For example, here is the unordered version of the segments in the sample “Lemurian” corpus (see *Example corpora*):
Once a feature file has been associated with a corpus, this unordered set can be arranged into something more closely resembling an IPA chart. Here is the Lemurian corpus once the IPA symbols have been interpreted using Hayes-style features (note that for space reasons, only the consonant chart is shown; the vowel chart is below in the actual PCT window):
If the feature system being used is either the built-in [Hayes2009] or the [SPE] style feature system, the sorting of segments (regardless of the transcription system) into a standard IPA-like chart will be done automatically (though it can still be edited). If a different feature system is used, however, the sorting may not be correct if PCT does not recognize the features. Therefore, the inventory categories can be edited.

To do so, click on “Edit inventory categories” in the “Features” / “View / change feature system...” menu. The “Edit categories” dialogue box appears. Essentially, you are telling PCT which feature values are associated with which segments in the inventory. There are five sets of categories to be edited: “Major distinctions,” “Places of articulation,” “Manners of articulation,” “Vowel height,” and “Vowel backness.” Each is described below. In each case, PCT will ask for which feature or set of features is used to specify a particular set of segments. You can then type in the box the name of the feature; PCT will auto-complete feature names. Once a feature has been included, you can simply mouseover the box on the right-hand side to view which segments from the inventory are included by the selected features, to check that they are correct and exhaustive. (Note that the order of feature selection doesn’t matter.)

1. **Major distinctions**: Use the major distinctions tab to edit major class distinctions, i.e., vowels vs. consonants; voicing; diphthongs; and rounding in vowels. For example, the feature specifying vowels in the [SPE] system is +voc; the feature in the [Hayes2009] system is +syllabic.

2. **Places of articulation**: Use the places of articulation tab to indicate which features are associated with each standard place of articulation. For example, the features to pick out labial segments in the [SPE] system are -cor, +ant, -back; in the [Hayes2009] system, they are -coronal, +labial.

3. **Manners of articulation**: Use the manners of articulation tab to indicate which features are associated with each standard manner of articulation. For example, the features to pick out stops in the [SPE] system are -cont, -nasal, -son; in the [Hayes2009] system, they are -delayed_release, -sonorant, -nasal, -continuant.
4. **Vowel height**: Use the vowel height tab to indicate which features are associated with each standard height of vowels. For example, the features to pick out close vowels in the [SPE] system are +high, +tense, -low; in the [Hayes2009] system, they are +high, +tense, -low.

5. **Vowel backness**: Use the vowel backness tab to indicate which features are associated with each standard degree of backness of vowels. For example, the features to pick out front vowels in the [SPE] system are +tense, -back; in the [Hayes2009] system, they are -back, +tense, +front.

Here is an example of the “Edit categories” box:

![Edit categories box](image)

### 5.6 Creating new tiers in the corpus

It is possible to have PCT extract a tier of segments from the transcribed words in your corpus, based on any segment, feature, or set of features that are defined for your corpus. For example, it is easy to extract separate tiers for consonants and vowels. This extraction is particularly useful if, for example, one is interested in looking at an analysis of predictability of distribution where the conditioning contexts are non-adjacent segments; the extraction of a tier allows otherwise non-adjacent segments to be adjacent to each other on the selected tier. For example, one could examine the possibility of vowel height harmony in language X by extracting the vowels from all words and then calculating the extent to which high / low vowels are predictably distributed in high / low vowel contexts. (See also *Adding a...* )
column for information on how to add a column to a corpus, which contains any kind of user-specified information, and Adding a “count” column for information on how to add a count column to a corpus, which contains counts of specific elements within each entry in the corpus.

To create a new tier for a corpus that is currently open, click on the “Corpus” menu and select either “Add tier...” or “Add abstract tier...”; the “create tier” dialogue box opens. An “abstract” tier is a tier that is not based directly on the transcriptins themselves, but rather abstracts to a higher level. As of June 2015, the only abstract tier available is a CV skeleton tier. Before creating the tier, you can “preview” the tier as in the following example; this shows what segments PCT thinks are consonants and vowels in the current corpus.

The example corpus after an abstract CV tier has been added:
To create a less abstract tier, i.e., one that is just an extraction of all transcription symbols in the corpus that have some particular characteristic(s), use the following instructions after choosing “Corpus” / “Add tier...”:

1. **Name**: Enter a short-hand name for the tier, which will appear as the column header in your corpus. For example, “vowels” or “consonants” or “nasals.”

2. **Basis for creating tier**: You can create the tier using natural classes if you base the tier on features; you can also create “unnatural” tiers that are simply extractions of any set of user-defined segments.

3. **Segments**: To actually select the segments, using either features or individually, follow the directions given in *Sound Selection*.

The image below shows an example of creating a tier to contain all the non-mid vowels in the example corpus. (Note that the image shows the mid vowels highlighted but not yet selected; one would need to hit “enter” again or choose “Select highlighted” to make the actual selection):
Finalizing the tier: To create the tier and return to the corpus, click on “Create tier.” It may take a moment to process the entire corpus, but a new column should be added to the corpus that shows the segments matching these feature selections for every word in the corpus.

Saving the tier: The tier can be saved along with the corpus for future use by selecting “Corpus” / “Save corpus” from the menu items (this will be done automatically if auto-save is on; see Preferences). It is also possible to export the corpus as a text file (.txt), which can be opened in other software, by selecting “File” / “Export corpus as text file.”

Removing a tier: To delete a tier that has been created, simply click on “Corpus” / “Remove tier or column...” and select the tier you want to remove; then click “Remove.” You can also right-click on the column name and select “Remove column.” Note that only tiers that have been added through PCT can be removed; tiers that are inherent in loaded corpora cannot be removed in PCT. You can, of course, export the corpus itself to a text file, remove the column manually, and then re-load the changed corpus. To remove all the added tiers, leaving only the inherent (“essential”) tiers in the corpus, select “Remove all non-essential columns.” PCT will list which columns are non-essential and verify that you want to remove them before the removal is permanent. The “essential” columns for most corpora are “Spelling,” “Transcription,” and “Frequency.”

The following shows an example of the a vowel tier added to the example corpus using the SPE feature system:
5.7 Adding, editing, and removing words, columns, and tiers

5.7.1 Adding a column

In addition to the ability to add tiers based on information already in the corpus, as described above in Creating new tiers in the corpus, it is also possible to add a column containing any other user-specified information to a corpus (see also Adding a “count” column to find out how to add a column based on counts of elements within each corpus entry). For example, one could add a “Part of Speech” column and indicate what the lexical category of each entry in the corpus is. Note that as a general proposition, it is probably easier to add such information in a spreadsheet before importing the corpus to PCT, where sorting and batch updates are easier, but we include this functionality in a basic form in case it is useful.

To add a column, go to “Corpus” / “Add column...” and do the following:

1. **Name**: Enter the name of the new column.

2. **Type of column**: Indicate what type of information the column will contain. The choices are “Spelling,” “Numeric,” and “Factor.” A spelling column will have values that are written out as strings of characters, with each entry taken to be a unique string. A numeric column will have numeric values, upon with mathematical operations can be performed. A factor column will have values that can contain characters or numbers, but are limited in number, as in the levels of a categorical variable. This is useful when, for example, the column
encodes categorical information such as part of speech, with each entry in the corpus belonging to one of a limited set of categories such as “Noun,” “Verb,” and “Preposition.”

3. **Default value**: A default value for the column can be entered if desired, such that every entry in the corpus receives that value in the new column. Individual entries can subsequently be edited to reflect its actual value (see *Editing a word*).

Click “Add column” to return to the corpus and see the new column, with its default values.

### 5.7.2 Adding a “count” column

In addition to adding columns that contain any kind of user-specified information (*Adding a column*), and tiers that contain phonological information based on the entries themselves (*Creating new tiers in the corpus*), one can also add “Count” columns, which contain information about the number of occurrences of a feature or segment in each entry in a corpus. For example, one could add a column that lists, for each entry, the number of round vowels that are contained in that entry. To add a count column, go to “Corpus” / “Add count column...” and then do the following:

1. **Name**: Enter the name of the new column.
2. **Tier**: Specify what tier the count column should refer to in order to determine the counts (e.g., transcription or a derived tier such as a vowel tier).
3. **Segment selection**: Use the standard *Sound Selection* instructions to select which segments or types of segments to count.

Click “Add count column” to return to the corpus and see the new column, with its count values automatically filled in.

### 5.7.3 Removing a tier or column

To delete a tier or column that has been created, simply click on “Corpus” / “Remove tier or column...” and select the tier you want to remove; then click “Remove.” Note that only tiers that have been added through PCT can be removed; tiers that are inherent in loaded corpora cannot be removed in PCT. You can, of course, export the corpus itself to a text file, remove the column manually, and then re-load the changed corpus. To remove all the added tiers, leaving only the inherent (“essential”) tiers in the corpus, select “Remove all non-essential columns.” PCT will list which columns are non-essential and verify that you want to remove them before the removal is permanent. The “essential” columns for most corpora are “Spelling,” “Transcription,” and “Frequency.”

### 5.7.4 Adding a word

As a general proposition, we don’t recommend using PCT as a database manager. It is designed to facilitate analyses of pre-existing corpora rather than to be an interface for creating corpora. That said, it is occasionally useful to be able to add a word to a pre-existing corpus in PCT. Note that this function will actually add the word to the corpus (and, if auto-save is on, the word will be saved automatically in future iterations of the corpus). If you simply need to add a word temporarily, e.g., to calculate the neighbourhood density of a hypothetical word given the current corpus, you can also add a word in the relevant function’s dialogue box, without adding the word permanently to the corpus.

To do add the word globally, however, go to “Corpus” / “Add new word...” and do the following:

1. **Spelling**: Type in the orthographic representation of the new word.
2. **Transcription**: To add in the phonetic transcription of the new word, it is best to use the provided inventory. While it is possible to type directly into the transcription box, using the provided inventory will ensure that all characters are understood by PCT to correspond to existing characters in the corpus (with their concomitant featural interpretation). (If there is no featural interpretation of your inventory, you will simply see a list of all the available segments, but they will not be classified by major category.) Clicking on the individual segments...
will add them to the transcription. Note that you do NOT need to include word boundaries at the beginning
and end of the word, even when the boundary symbol is included as a member of the inventory; these will be
assumed automatically by PCT.

3. **Frequency**: Enter the token frequency of this word.

4. **Other**: If there are other tiers or columns in your corpus, you can also enter the relevant values for those columns
in the dialogue box. For tiers that are defined via features, the values should be automatically populated as you
enter the transcription. E.g., if you have a vowel tier, and add the word [pim.ku] to your corpus by selecting the
relevant segments from the inventory, the vowel tier should automatically fill in the entry as [i.u].

Once all values are filled in, select “Create word” to return to the corpus with the word added. If auto-save is not on,
you can save this new version of the corpus for future use by going to “File” / “Save corpus.” If you have added a word
and the corpus has NOT been saved (either manually or through auto-save) afterward, and then try to quit PCT, it will
warn you that you have unsaved changes and ask that you verify that you want to quit without saving them.

### 5.7.5 Removing a word

To remove a word from the corpus, select it in the corpus view and right-click (ctrl-click on a Mac) on it. Choose
“Remove word” from the menu. Regardless of whether warnings are turned on or not (see **Help and warnings**), PCT
will verify that you want to remove the word before committing the change. Word removal is not auto-saved with a
corpus, even if “Auto-save” is turned on (see **Preferences**); if you want to save the new version of the corpus with the
word removed, you should explicitly go to “File” / “Save corpus.” If you have removed a word and NOT manually
saved the corpus afterward, and then try to quit PCT, it will again warn you that you have unsaved changes and ask
that you verify that you want to quit.

### 5.7.6 Editing a word

To edit a word in the corpus, right-click on the word’s entry and choose “Edit word details,” or double-click the
word’s entry in the corpus. A dialogue box opens that shows the word’s spelling, transcription, frequency, and any
other information that is included in the corpus. Most of these entries can be edited manually, though a few, such as
tiers that are dependent on a word’s transcription, cannot themselves be directly edited. To edit such a derived tier, edit
the transcription of the word; the derived tier will update automatically as the new transcription is provided.

### 5.7.7 Hiding / showing non-transcribed items

When working with a corpus, it is possible to hide all entries that do not have a transcription (if any such entries exist).
To do this, right-click anywhere in the corpus itself and select “Hide non-transcribed items.” To reveal them again,
right-click anywhere in the corpus itself and select “Show non-transcribed items.”

### 5.8 Phonological Search

PCT allows you to do searches for various strings, defined by segments or features. The search returns two types of
information: one, a general count of the number of entries that fit the search description, and two, the actual list of
all the words in the corpus that contain the specified string. To conduct a search, choose “Corpus” / “Phonological
search...” and do the following:

1. **Environments**: Select the strings you want to search for. See **Environment Selection** and **Sound Selection** for
details.

2. **Tier**: Select the tier on which phonological search should be performed. The default would be the transcription
tier, so that phonological environments are defined locally. But, for example, if a vowel tier is selected, then
one could search for the occurrence of, e.g., [i] before mid vowels on that tier (hence ignoring intervening consonants). (Note that it is not currently possible to do a phonological search within *Pronunciation Variants*; the search will look only at the canonical forms or whatever forms are listed in the specified tier.)

An example of adding environments (in this case, the environment “word-initial, before a vowel”):

An example of the phonological search window, set up to search for voiceless stops word-initially before vowels and between [a] vowels, on the transcription tier:

3. **Results**: Once all selections have been made, click on “Calculate phonological search.” If there is not already an existing results table, or you want to start a new one, choose the “Start new results table” option. If you want to add the results to a pre-existing table, choose the “Add to current results table” option. The results appear in a new dialogue box that first shows the summary results, i.e., a list that contains the segment that was searched for, each environment that was searched for, the total count of words that contain that segment in that environment, and the total token frequency for those words (note that these are the frequencies of the WORDS containing the specified environments, so if for example, a particular word contains multiple instances of the same environment, this is NOT reflected in the counts). The individual words in the corpus that match the search criteria can be shown by clicking on “Show individual results” at the bottom of the screen; this opens a new dialogue box in which each word in the corpus that matches the search criteria is listed, including the transcription of the word, the segment that was found that matches the search criteria, and which environment that segment occurred in in that word. Note that the results can be sorted by any of the columns by clicking on that column’s name (e.g., to get all the words that contained the [a_a] environment together, simply click on
To return to the summary results, click on “Show summary results.” Each set of results can be saved to a .txt file by clicking “Save to file” at the bottom of the relevant results window. To return to the search selection dialogue box, click on “Reopen function dialogue.” Otherwise, when finished, click on “Close window” to return to the corpus.

An example of the summary results window for the above phonological search:

And the individual results from the same search, sorted by environment:
5.8.1 Classes and functions

For further details about the relevant classes and functions in PCT’s source code, please refer to *API Reference.*
Sound Selection

There are many instances in PCT where you are given the opportunity to select sounds that will be used in an analysis or a search. This section describes the general interface used in all such instances, including using featural descriptions to select sounds. See also Feature Selection for information about how to select features on which to do analyses.

When selecting a sound or a pair of sounds, you will be presented with a dialogue box that contains all of the unique segments that occur in the corpus. If there is no feature system associated with the corpus, then these will be arranged in alphabetical order as a solid block. If there is an associated feature system, then the segments will be arranged in an IPA-style format. You can customize the layout of this display; for details on how to do so, see Edit inventory categories. If there are associated features, there will also be an option at the top of the “select segment” box to “select by feature.”

To select sounds, there are two options:

First, you can simply click on individual segments in the list or chart of segments. Multiple segments can be selected at once. If you are selecting sounds for an analysis that involves a single segment, then each selected segment will undergo the same analysis. (E.g., in a Phonological Search, if you select A, B, C, and D as the target sounds, then each of those sounds will be searched for in the same environment.) If you are selecting sounds for an analysis that involves a pair of sounds, then all the pairwise combinations of the selected sounds will be created. (E.g., if you select A, B, C, and D as sounds in a pairwise selection, then all of the pairs, i.e., AB, AC, AD, BC, BD, and CD, will be selected. In many analyses the order of the sounds in the pair doesn’t matter, but PCT also will allow you to switch the order after the pairs have been created.)

Second, if there is an associated feature file, you can select sounds using featural descriptions. To do so, simply start typing a feature value into the “Select by feature” box at the top of the sound selection box. As soon as any value or characters are typed, a dropdown box will appear that lists all of the available features that match the current typing. For example, just typing a “+” sign will reveal + values of all features that have a + specification in the feature chart (e.g., +anterior, +coronal, +long, +nasal, +sonorant, +vocalic, etc.). Typing a letter, such as “c” will show all features that start with that letter (e.g., -consonantal, -constricted glottis, -consonantal, -coronal, +consonantal, +constricted glottis, +consonantal, +coronal, etc.). You can continue typing out the feature or select one from the list by either clicking on it or hitting “return” when it is the one highlighted. Once a feature is entered, all segments that have that feature specification will be highlighted in red on the chart. This does not in fact “select” them yet – it just indicates which segments match the currently listed specifications. Once segments are highlighted, you may continue entering features to winnow down the selection, or revert to clicking on individual segments (e.g., from among the highlighted ones). As more features are typed in and selected, the highlighting in the chart will update to match the current feature specification. (Features in the list can be separated by commas or spaces.) To actually SELECT all the highlighted segments, you can simply hit “enter” again after the names of the features are completely entered, and the highlighting will change to selection. Alternatively, you can click on the “Select highlighted” button. Note that if you just leave them highlighted, no segments will actually be selected.
Environment Selection

There are many instances in PCT where you are given the opportunity to define environments that will be used in an analysis. This section describes the general interface used in all such instances.

When you have the opportunity to define an environment, you will see a blank “Environments” box with a button in it for “New environment.” Start by clicking this button to create a new environment – you can click it as many times as you need to define multiple environments.

When a new environment has been added, it starts out blank. Environments are basically divided into three sections: the target, the left-hand side, and the right-hand side.

The central rectangle marks the “target” of the environment and has an underscore at the top and a set of empty curly brackets, {}, beneath. Depending on where you are building the environment, you may be able to edit the contents of this target. E.g., if you are doing a Phonological Search, you can put the target(s) of the search here, and fill in the environment around them. On the other hand, if you are building an environment to be used with pairs of sounds, as in the Predictability of Distribution analysis, you will have specified the target pairs of sounds independently, and cannot edit the targets in the environment selection box. In order to select a target in cases where it is editable, simply click on the central target box; a sound selection box will open. See Sound Selection for details on how to use this function, but basically, sounds can be selected either by clicking on segments or by specifying feature values. Multiple sounds can be selected as the targets; each will be placed in the same surrounding environments.

On either side of the central target rectangle, there is a “+” button. These allow you to add segments to either the left-hand or the right-hand side of the environment in an iterative fashion, starting with segments closest to the target and working out. Clicking on one of the “+” buttons adds an empty set {} to the left or right of the current environment.

To fill the left- or right-hand side, click on the rectangle containing the empty set {}. This again brings up the sound selection box; see Sound Selection for details. The environment can be filled by either clicking on segments or specifying features.

Note that regardless of whether targets and environments are selected by segments or by features, the result will be a disjunctive set of all segments that have been selected.

For example, to set up an environment that might be used to search for [n] vs. [m] vs. [ɲ] in words that start with [ɾ] and in which the nasal is followed by voiceless stop, one could do the following:

1. Click on “New environment.” This gives you a blank environment:
2. Click on the left-hand “+” sign twice, and the right-hand “+” sign once. This will give you the following, still blank environment:

3. Click the central “target” rectangle.

4. Select [m], [n], and [ŋ] either by hand or through their features (e.g., [+nasal, -vocalic] or whatever combination of features is relevant for the corpus). (Be sure to fully select the segments if you’re using features; don’t just have them highlighted, or they won’t get added to the environment. See Sound Selection for details.)

5. Click the leftmost empty set.

6. Select the word boundary symbol, #.

7. Click the empty set immediately to the left of the targets.

8. Select the vowel [i].

9. Select the rightmost empty set.

10. Select all voiceless stops (e.g., by using the features [-voice, -continuant, -delayed release]).

This now gives you an environment that looks something like the following (depending on the total inventory and the transcription system of your corpus; this example is from the IPHOD corpus):
To add additional environments, simply click “New environment.” To edit a current environment, simply click on the rectangle containing the part of the environment you want to edit and re-select the sounds. To remove an environment entirely, click on the “Remove environment” button to the right.
There are several instances in PCT where you are given the opportunity to select sounds for an analysis based on shared features. This section describes the general interface used in all such instances. See also Sound Selection for general information on how to select individual segments for analysis, including using features to identify classes of segments.

In many cases, you will just want to select individual sounds for an analysis (e.g., [e] vs. [o]). Occasionally, however, it is useful to be able to compare classes of sounds that differ along some dimension (e.g., comparing front vs. back non-low vowels, i.e., [i, e] on the one hand vs. [u, o] on the other).

To do this in an analysis window, click on “Add pair of features” to open the “Select feature pair” dialogue box. At the top of the box, there is a place to enter the feature(s) along which the pairs will have OPPOSITE values. No “+” or “-” value should be entered here; rather, it should just be the name of the feature (e.g., “back”). Note that currently, PCT has some ability to automatically detect redundant features within a given domain. For instance, if the example corpus is open and associated with SPE features, and one wanted to calculate the predictability of distribution of [i, e] on the one hand vs. [u, o] on the other, one would could enter either “back” or “round”, but would also have enter “-low” in the “Filter pairs” box. To accomplish the same thing with Hayes features, one needs to enter only one of “back, front, round, or labial.” On the other hand, if one wanted to calculate the predictability of distribution for high vowels [i, u] on the one hand and [e, o] on the other, one need only list the feature [high]. The automatic detection is based on the inventory size, so smaller inventories will have more detectable redundancies than larger inventories. Larger inventories will thus have to have more features entered for the segment sets to be selected.

As soon as a feature or set of features has been entered that describes two sets of sounds that differ on exactly the feature values for the listed features, the sounds themselves will be shown in the box under “First segments” and “Second segments.”

One can then filter the entire set by entering in specific values of other features. E.g., if one wanted to limit the comparison to [i] vs. [e], one could enter “high” in the “feature to make pairs” box and then [round] in the “filter pairs” box. (Of course, in this case, it might be easier to simply select those two sounds, [i] and [e], directly as segments, but the same principle works for more complicated sets of segments.)

Once the correct segments are listed, click “Add” to add the pairs to the segment list in the original analysis dialogue box. If additional pairs are needed, one can click “Add and create another” instead.

Here’s an example of using both features and filters in the Lemurian corpus to select [o,u] vs. [e,i], to the exclusion of [w,j]:

---

CHAPTER 8

Feature Selection

---
9.1 About Pronunciation Variants:

In some corpora (e.g., spontaneous speech corpora), there may be different pronunciation variants that are associated with the same lexical item. For instance, the word “probably” might be variably produced as [pəɔbəli] (its canonical form) or as [pəəli], [pəəl], [pəəli], or any of a number of other reduced forms. In the Buckeye corpus, for example, there are 290 tokens of the word “probably”, only 50 of which have the canonical pronunciation ([BUCKEYE]).

A number of studies have looked at the effects of this kind of surface variation on lexical representations (e.g., [Connine2008]; [Pinnow2014]; [Pitt2009]; [Pitt2011]; [Sumner2009]). For example, [Pinnow2014] show that lexical items with high-frequency variants that include deleted schwas have processing advantages over similar items where the deleted variant is less frequent, suggesting that frequency matters in the representations.

Because such differences may matter, PCT includes several options (Options for Pronunciation Variants:) for handling pronunciation variants in its analysis functions (though note that not every option is available in every analysis function as of version 1.1). To the best of our knowledge, none of the analysis functions we provide has ever been applied to anything other than the canonical pronunciations of forms, so it is a completely open question as to which of these options is best, or even how to determine which one might be best (e.g., in terms of correlations with other linguistic or behavioural patterns). It is simply our hope that by allowing the functions to be applied in a variety of ways, such empirical questions can be answered in the future.

9.2 Creating Pronunciation Variants:

To create a corpus that has pronunciation variants, there are several options (see also Loading in corpora). First is to use a specially formatted corpus, such as the Buckeye corpus ([BUCKEYE]), that PCT already recognizes as containing pronunciation variants. Second is to create a corpus from an interlinear gloss file that has separate lines for lexical items and their pronunciations. For example, a fragment of the text might be as follows:

<table>
<thead>
<tr>
<th>Line 1:</th>
<th>Well,</th>
<th>I’m</th>
<th>probably</th>
<th>going</th>
<th>to</th>
<th>go</th>
<th>to</th>
<th>the</th>
<th>pool</th>
<th>after.</th>
</tr>
</thead>
<tbody>
<tr>
<td>wrl</td>
<td>am</td>
<td>prəəbəli</td>
<td>goʊŋŋ</td>
<td>tu</td>
<td>goʊ</td>
<td>tu</td>
<td>ðə</td>
<td>pul</td>
<td>æftʃ</td>
<td></td>
</tr>
<tr>
<td>wrl</td>
<td>am</td>
<td>prəli</td>
<td>ɡn</td>
<td>ə</td>
<td>goʊ</td>
<td>ə</td>
<td>ðə</td>
<td>pul</td>
<td>æftʃ</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line 2:</th>
<th>Is</th>
<th>Max</th>
<th>going</th>
<th>with</th>
<th>us?</th>
</tr>
</thead>
<tbody>
<tr>
<td>iz</td>
<td>məks</td>
<td>goʊŋŋ</td>
<td>wɪθ</td>
<td>s</td>
<td></td>
</tr>
<tr>
<td>z</td>
<td>məks</td>
<td>goʊŋŋ</td>
<td>wɪθ</td>
<td>s</td>
<td></td>
</tr>
</tbody>
</table>

| Line 3: |
Oh, no, he’ll probably stay home.

The first line shows the spelling; the second the canonical pronunciation of each word; and the third the pronunciation of the word as it was actually said in context. This would get read into PCT as:

<table>
<thead>
<tr>
<th>spelling</th>
<th>transcription</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>I’m</td>
<td>aɪ.ɛm</td>
<td>1</td>
</tr>
<tr>
<td>is</td>
<td>ɪ.z</td>
<td>1</td>
</tr>
<tr>
<td>Max</td>
<td>mæ.ks</td>
<td>1</td>
</tr>
<tr>
<td>oh</td>
<td>ʌʊ</td>
<td>1</td>
</tr>
<tr>
<td>well</td>
<td>wɛ.ɹ.l</td>
<td>1</td>
</tr>
<tr>
<td>after</td>
<td>æ.ɹ.t.ɛr</td>
<td>1</td>
</tr>
<tr>
<td>go</td>
<td>g.oʊ</td>
<td>1</td>
</tr>
<tr>
<td>going</td>
<td>g.ʊʊ.ɹ.tj</td>
<td>2</td>
</tr>
<tr>
<td>he’ll</td>
<td>h.ɹ.l</td>
<td>1</td>
</tr>
<tr>
<td>home</td>
<td>h.ʊ.ʊ.m</td>
<td>1</td>
</tr>
<tr>
<td>no</td>
<td>n.ʊʊ</td>
<td>1</td>
</tr>
<tr>
<td>pool</td>
<td>p.u.l</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.ɹ.ɹ.ɹ.b.l.i</td>
<td>2</td>
</tr>
<tr>
<td>stay</td>
<td>s.t.ɛɾ</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>ð.ə</td>
<td>1</td>
</tr>
<tr>
<td>to</td>
<td>t.u</td>
<td>2</td>
</tr>
<tr>
<td>us</td>
<td>.s</td>
<td>1</td>
</tr>
<tr>
<td>with</td>
<td>w.ɹ.tθ</td>
<td>1</td>
</tr>
</tbody>
</table>

...where the words “going,” “to,” and “probably” each have multiple variants associated with them (see Viewing Pronunciation Variants: for more on how to view these). (Note that to achieve this result, you would indicate in the Parsing Parameters that both lines 2 and 3 are “Transcription” lines, but that line 2 is associated with the lexical item while line 3 is allowed to vary within lexical items.)

A similar method works for creating pronunciation variants from TextGrid files. As in the interlinear gloss files, you would have three tiers in a TextGrid; one with spelling, one with canonical pronunciations; and one with the specific pronunciations used in particular instances. The Parsing Parameters would be filled in similarly.

Note that the canonical pronunciations are in fact optional; it is possible to simply associate pronunciation variants with spelling forms (though this may limit some of the functionality of searching and some analyses).

9.3 Viewing Pronunciation Variants:

If a corpus has pronunciation variants, you can view these by right-clicking on any word in the corpus and selecting “List pronunciation variants.” A new dialogue box will pop up that shows the individual pronunciation variants that occur in the corpus for that word, along with their token frequencies. (See also Exporting Pronunciation Variants: for information about how to save these to a .txt file for use outside of PCT.)
9.4 Options for Pronunciation Variants:

There are four basic options in PCT for dealing with pronunciation variants, each of which is described below. In all cases, the way that PCT handles them is to create a version of the corpus that is set up with the selected option; analysis functions are then applied as normal. Thus, each time that a non-canonical approach is used, there will be a slight delay in processing time while the alternative corpus structure is set up.

1. **Canonical forms**: The first option is for PCT to use only the canonical forms in analyses. This is the default and the option that is usually reported in the literature. For many corpora, only the canonical pronunciation is available anyway, and will be the only option. In corpora with pronunciation variants, this option is available only if there is a form that is known by PCT to be the canonical pronunciation (see Creating Pronunciation Variants). Note that the token frequency values are summed across all variants.

   As an example, the canonical form for the word “cat” is [kæt]; the canonical form for the word “probably” is [pəˈbli]. A fragment of the corpus would be as follows:

<table>
<thead>
<tr>
<th>Spelling</th>
<th>Transcription</th>
<th>Type frequency</th>
<th>Token frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>kæt</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>probably</td>
<td>pəˈbli</td>
<td>1</td>
<td>290</td>
</tr>
</tbody>
</table>

2. **Most frequent forms**: Alternatively, PCT can use only the most frequent variant of each lexical item in analyses. This option puts priority on forms as they are actually used most often in the corpus. If there are two or more forms that have equal frequencies, and one of them is the canonical form, then PCT will fall back on the canonical form, if one is available. If no canonical form is available or if it is not one of the forms that is tied for being most frequent, then the longest of the most frequent forms will be chosen (on the assumption that this will be closest to the canonical form). If there is a tie in terms of frequencies AND a tie in terms of the lengths of the tied forms, then PCT will simply use the variant that is first alphabetically.

   As an example, the most frequent form of the word “probably” in the Buckeye corpus is [pəˈbli]; 66 of the 290 tokens of the word have this form (whereas only 50 are the canonical pronunciation). Similarly, the most frequent form for the word “cat” is [kæ]; 3 of the six tokens of “cat” have this pronunciation. Thus, searches and analyses using the most frequent forms would use these transcriptions instead. The token frequency values are again combined across all variants.

   A fragment of the corpus would be as follows:

<table>
<thead>
<tr>
<th>Spelling</th>
<th>Transcription</th>
<th>Type frequency</th>
<th>Token frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>kæt</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>probably</td>
<td>pəˈbli</td>
<td>1</td>
<td>290</td>
</tr>
</tbody>
</table>
3. **Each word token separately**: The third option is for PCT to treat each pronunciation variant as its own separate lexical entry. This allows all variants to be considered, regardless of canonical-ness or frequency. At the same time, it will somewhat artificially inflate the number of occurrences of segments that relatively stably occur in words that otherwise have lots of variation. For example, there are 74 different pronunciation variants of the word “probably” in the Buckeye corpus; 73 of these begin with [p] (one, [frai], begins with [f]). Thus, while this method is useful for seeing the range of variability elsewhere in the word, it will make word-initial [p] seem much more relatively frequent than it actually is. It allows every pronunciation variant to count equally as far as word types are concerned. Token frequencies for each individual variant are used or each variant is assigned a frequency of 1 if type frequencies are used.

A fragment of the corpus would be as follows:

<table>
<thead>
<tr>
<th>Spelling</th>
<th>Transcription</th>
<th>Type frequency</th>
<th>Token frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>k.æ.t</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>cat</td>
<td>k.æ.</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>cat</td>
<td>k.æ.</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.i.a.b.b.i.i</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>probably</td>
<td>p.i.a.b.i.i</td>
<td>1</td>
<td>66</td>
</tr>
<tr>
<td>probably</td>
<td>p.i.a.i.i</td>
<td>1</td>
<td>35</td>
</tr>
</tbody>
</table>

... (not all variants of the word “probably” are shown)

4. **Weighted word types by the frequency of each variant**: The fourth option is for PCT to weight each variant’s frequency by the overall token frequency (if using token frequency) or by the number of variants (if using type frequency).

As an example, the word “probably” has 74 variants in the Buckeye corpus. The most frequent, [pæblí], occurs 66 times out of the 290 tokens. 66/290 = 0.2276. So, there would be a lexical entry in the corpus for [pæblí], with a type frequency of 0.2276 (instead of 1). Similarly, the canonical pronunciation, [pæbblí], occurs with a relative frequency of 50/290 = 0.1724, so that would be the type frequency for its lexical entry. Thus, the total type frequency across all variants of a single lexical item sum to 1. The token frequencies match the original numbers.

A fragment of the corpus would be as follows:

<table>
<thead>
<tr>
<th>Spelling</th>
<th>Transcription</th>
<th>Type frequency</th>
<th>Token frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>k.æ.t</td>
<td>0.333</td>
<td>2</td>
</tr>
<tr>
<td>cat</td>
<td>k.æ.</td>
<td>0.5</td>
<td>3</td>
</tr>
<tr>
<td>cat</td>
<td>k.æ.</td>
<td>0.167</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.i.a.b.b.i.i</td>
<td>0.172</td>
<td>50</td>
</tr>
<tr>
<td>probably</td>
<td>p.i.a.b.i.i</td>
<td>0.228</td>
<td>66</td>
</tr>
<tr>
<td>probably</td>
<td>p.i.a.i.i</td>
<td>0.121</td>
<td>35</td>
</tr>
</tbody>
</table>

... (not all variants of the word “probably” are shown)

### 9.5 Exporting Pronunciation Variants:

It is possible to export pronunciation variants with a corpus for easy reference or use outside of PCT. General information about exporting a corpus can be found in *Saving and exporting a corpus or feature file*. The basic procedure is to go to “File” / “Export corpus as text file” and enter the file name and location and the column and transcription delimiters.

PCT provides three options for exporting pronunciation variants. They can simply be excluded entirely (by selecting “Do not include”); the resulting file will have only the canonical pronunciations, assuming the corpus contains these. The following is an example of the resulting single line of the output file from the Buckeye corpus for the word “probably”:
Alternatively, pronunciation variants can be included in either of two formats. Selecting “Include in each word’s line” will organize the output by lexical item, with exactly one line per item. Pronunciation variants of that item will be listed at the end of the line. Here’s an example of the single line that results for the word “probably” in this version of the export of the Buckeye corpus:

The other format for exporting pronunciation variants, “Have a line for each variant,” will put each different variant on a separate line in the exported corpus. Each will also include the spelling and canonical transcription (if available). This version also lists the frequency with which each different variant occurs in the corpus. Here’s an example of the 74 lines that result for the word “probably” in this version of the export of the Buckeye corpus:
<table>
<thead>
<tr>
<th>Spelling</th>
<th>Transcription</th>
<th>Frequency</th>
<th>Token_Transcription</th>
<th>Token_Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.raa.bah.bl</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.raa.bel.ih</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.rah.b.iy</td>
<td>2</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.raa.b.w.iy.jh</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.raa.b.liy</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.raa.w.iy.jh</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.p.raa.b.l.iy</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.raa.b.uh.b.l.iy</td>
<td>3</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.raa.b.ih</td>
<td>2</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.rah.ay</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.rah.b.ih</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.raa.ih.m</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.raa.el.iy</td>
<td>2</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.raa.bah.bl.iy</td>
<td>50</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.raa.b.liy</td>
<td>6</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.raa.bl.ah</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.aa.how.bl.iy</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.er.reh.ih</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.aa.bow.liy</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.raa.b.lih</td>
<td>5</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.raa.b.liy</td>
<td>11</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.rah.b.w.iy</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.raa.bow.bl.ey</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.raa.bl.eliy</td>
<td>2</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.aa.bow.liy</td>
<td>2</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.rah.b.liy</td>
<td>4</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.raa.bah.bl.ih</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.raa.v.iy</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.rah.ey</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.aa.b.ih</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.rah.ih</td>
<td>2</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.aa.ih.ih</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.raa.bl.eliy</td>
<td>3</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.raa.riy</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.raa.liuw</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.aa.ih</td>
<td>2</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.aa.bl.iy</td>
<td>2</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.raa.bl.rih</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.ah.bl.iy</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.rao.b.ih</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.ah.lih</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.aa.r</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.raa.w.iy</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.rao.ey</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.raa</td>
<td>2</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.raa.1iy</td>
<td>35</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.row.iy</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.ay</td>
<td>16</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.aa.1iy</td>
<td>1</td>
</tr>
<tr>
<td>probably</td>
<td>p.raa.bah.bl.iy</td>
<td>290</td>
<td>p.rah.b.uh.bl.iy</td>
<td>1</td>
</tr>
</tbody>
</table>
CHAPTER 10

Phonotactic Probability

10.1 About the function

Phonotactic probability refers to the likelihood of a given set of segments occurring in a given order for a given corpus of transcriptions. For instance, blick is a phonotactically probable nonword in English, but bnick is phonotactically improbable. Words as well as nonwords can be assessed for their phonotactic probability, and this measure has been used in behavioural research ([Vitevitch1999] and others). In particular, the phonotactic probability of words has been correlated with their ability to be segmented, acquired, processed, and produced; see especially the discussion in [Vitevitch2004] for extensive references.

10.2 Method of calculation

One method for computing the phonotactic probability, and the current algorithm implemented in PCT, uses average unigram or bigram positional probabilities across a word ([Vitevitch2004]; their online calculator for this function is available here). For a word like blick in English, the unigram average would include the probability of /b/ occurring in the first position of a word, the probability of /l/ in the second position, the probability of /I/ occurring in the third position, and the probability of /k/ occurring in the fourth position of a word. Each positional probability is calculated by summing the log token frequency of words containing that segment in that position divided by the sum of the log token frequency of all words that have that position in their transcription. The bigram average is calculated in an equivalent way, except that sequences of two segments and their positions are used instead of single segments. So for blick that would be /bl/, /lI/, /Ik/ as the included positional probabilities. As with all n-gram based approaches, bigrams are preferable to unigrams. In the example of blick versus bnick, unigrams wouldn’t likely capture the intuitive difference in phonotactic probability, since the probability of /n/ and /l/ in the second position isn’t necessarily radically different. Using bigrams, however, would capture that the probability of /bl/ versus /bn/ in the first position is radically different.

There are other ways of calculating phonotactic probability that don’t have the strict left-to-right positional assumptions that the Vitevitch & Luce algorithm has, such as the constraint-based method in BLICK by Bruce Hayes (Windows executable available on the Blick homepage, Python package available at python-blick on PyPi with source code available at python-blick on GitHub). However, such algorithms require training on a specific language, and the constraints are not computed from transcribed corpora in as straightforward a manner as the probabilities used in the Vitevitch & Luce algorithm. Therefore, PCT currently supports only the Vitevitch & Luce style algorithm.

10.3 Calculating phonotactic probability in the GUI

To start the analysis, click on “Analysis” / “Calculate phonotactic probability...” in the main menu, and then follow these steps:
1. **Phonotactic probability algorithm**: Currently the only offered algorithm is the Vitevitch & Luce algorithm, described above.

2. **Query type**: Phonotactic probability can be calculated for one of three types of inputs:
   
   (a) **One word**: The phonotactic probability of a single word can be calculated by entering that word’s orthographic representation in the query box.

   (b) **One word/nonword not in the corpus**: The phonotactic probability can be calculated on a word that is not itself in the corpus, but using the probabilities derived from the corpus. These words are distinct from the corpus and won’t be added to it, nor will their creation affect any future calculations. See *Adding a word* for information on how to more permanently add a new word to the corpus. Words can be created through the dialogue opened by pressing the button:

      i. **Spelling**: Enter the spelling for your new word / nonword using the regular input keyboard on your computer. The spelling is how the word will be referenced in the results table, but won’t affect the calculation of phonotactic probability.

      ii. **Transcription**: To add in the phonetic transcription of the new word, it is best to use the provided inventory. While it is possible to type directly into the transcription box, using the provided inventory will ensure that all characters are understood by PCT to correspond to existing characters in the corpus (with their concomitant featural interpretation). Click on “Show inventory” and then choose to show “Consonants,” “Vowels,” and/or other. (If there is no featural interpretation of your inventory, you will simply see a list of all the available segments, but they will not be classified by major category.) Clicking on the individual segments will add them to the transcription. The selections will remain even when the sub-inventories are hidden; we allow for showing / hiding the inventories to ensure that all relevant buttons on the dialogue box are available, even on small computer screens. Note that you do NOT need to include word boundaries at the beginning and end of the word, even when the boundary symbol is included as a member of the inventory; these will be assumed automatically by PCT.

      iii. **Frequency**: This can be left at the default. Note that entering a value will NOT affect the calculation; there is no particular need to enter anything here (it is an artifact of using the same dialogue box here as in the “Add Word” function described in *Adding a word*).

      iv. **Create word**: To finish and return to the “Phonotactic probability” dialogue box, click on “Create word.”

   (c) **List of words**: If there is a specific list of words for which phonotactic probability is to be calculated (e.g., the stimuli list for an experiment), that list can be saved as a .txt file with one word per line and uploaded into PCT for analysis. If words in the list are not in the corpus, you can still calculate their phonotactic probability by entering in the spelling of the word and the transcription of the word in a single line delimited by a tab. The transcription should be delimited by periods.

   (d) **Whole corpus**: Alternatively, the phonotactic probability for every current word in the corpus can be calculated. The phonotactic probability of each word will be added to the corpus itself, as a separate column; in the “query” box, simply enter the name of that column (the default is “Phonotactic probability”).

3. **Tier**: Phonotactic probability can be calculated from transcription tiers in a corpus (e.g., transcription or tiers that represent subsets of entries, such as a vowel or consonant tier).

4. **Pronunciation variants**: Specify whether phonotactic probability should be calculated based on the canonical pronunciations of each word or the most frequent pronunciations (which may not be the same). See more in *Pronunciation Variants*.

5. **Type vs. token frequency**: Specify whether phonotactic probabilities should be based on word type frequency or token frequency. The original Vitevitch & Luce algorithm uses token frequency. Token frequency will use the log frequency when calculating probabilities.
6. **Probability type**: Specify whether to use biphone positional probabilities or single segment positional probabilities. Defaults to biphone.

7. **Results**: Once all options have been selected, click “Calculate phonotactic probability.” If this is not the first calculation, and you want to add the results to a pre-existing results table, select the choice that says “add to current results table.” Otherwise, select “start new results table.” A dialogue box will open, showing a table of the results, including the word, its phonotactic probability, the transcription tier from which phonotactic probability was calculated, whether type or token frequency was used, whether the algorithm used unigram or bigram probabilities, and the phonotactic probability algorithm that was used. If the phonotactic probability for all words in the corpus is being calculated, simply click on the “start new results table” option, and you will be returned to your corpus, where a new column has been added automatically.

8. **Saving results**: The results tables can each be saved to tab-delimited .txt files by selecting “Save to file” at the bottom of the window. If all phonotactic probabilities are calculated for a corpus, the corpus itself can be saved by going to “File” / “Export corpus as text file,” from where it can be reloaded into PCT for use in future sessions with the phonotactic probabilities included.

An example of the “Phonotactic Probability” dialogue box for calculating the probability of the non-word “pidger” [pɪdʒə], or [P.IH.JH.ER] in Arpabet, using unigram position probabilities (using the IPHOD corpus):

To return to the function dialogue box with your most recently used selections, click on “Reopen function dialog.” Otherwise, the results table can be closed and you will be returned to your corpus view.

### 10.4 Classes and functions

For further details about the relevant classes and functions in PCT’s source code, please refer to *Phonotactic probability*.
11.1 About the function

Functional load is a measure of the “work” that any particular contrast does in a language, as compared to other contrasts (e.g., [Hockett1955], [Hockett1966]; [Kucera1963]; [King1967]; [Surendran2003]). Two contrasts in a language, such as [d] / [t] vs. [ð] / [θ] in English, may have very different functional loads. The difference between [d] and [t] is used to distinguish between many different lexical items, so it has a high functional load; there are, on the other hand, very few lexical items that hinge on the distinction between [ð] and [θ], so its functional load is much lower. One of the primary claims about functional load is that it is related to sounds’ propensity to merge over time, with pairs of sounds that have higher functional loads being less likely to merge than pairs of sounds with lower functional loads (e.g., [Wedel2013], [Todd2012]). The average functional load of a particular sound has also been claimed to affect its likelihood of being used as an epenthetic vowel [Hume2013]. Functional load has also been illustrated to affect the perceived similarity of sounds [Hall2014a].

11.2 Method of calculation

There are two primary ways of calculating functional load that are provided as part of the PCT package. One is based on the change of entropy in a system upon merger of a segment pair or set of segment pairs (cf. [Surendran2003]); the other is based on simply counting up the number of minimal pairs (differing in only the target segment pair or pairs) that occur in the corpus.

11.2.1 Change in entropy

The calculation based on change in entropy is described in detail in [Surendran2003]. Entropy is an Information-Theoretic measure of the amount of uncertainty in a system [Shannon1949], and is calculated using the formula in (1); it will also be used for the calculation of predictability of distribution (see Method of calculation). For every symbol \( i \) in some inventory (e.g., every phoneme in the phoneme inventory, or every word in the lexicon), one multiplies the probability of \( i \) by the \( \log_2 \) of the probability of \( i \); the entropy is the sum of the products for all symbols in the inventory.

Entropy:

\[
H = - \sum_{i \in N} p_i \times \log_2(p_i)
\]

The functional load of any pair of sounds in the system, then, can be calculated by first calculating the entropy of the system at some level of structure (e.g., words, syllables) with all sounds included, then merging the pair of sounds in question and re-calculating the entropy of the new system. That is, the functional load is the amount of uncertainty (entropy) that is lost by the merger. If the pair has a functional load of 0, then nothing has changed when the two are
merged, and $H_1$ will equal $H_2$. If the pair has a non-zero functional load, then the total inventory has become smaller through the conflating of pairs of symbols that were distinguished only through the given pair of sounds.

Functional load as change in entropy:
\[ \Delta H = H_1 - H_2 \]

Consider a toy example, in which the following corpus is assumed (note that, generally speaking, there is no “type frequency” column in a PCT corpus, as it is assumed that each row in the corpus represents 1 type; it is included here for clarity):

Consider a toy example, in which the following corpus is assumed (note that, generally speaking, there is no “type frequency” column in a PCT corpus, as it is assumed that each row in the corpus represents 1 type; it is included here for clarity):

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>hot</td>
<td>[hæt]</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>song</td>
<td>[sæŋ]</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>hat</td>
<td>[hæt]</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sing</td>
<td>[sɪŋ]</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>tot</td>
<td>[tæt]</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>dot</td>
<td>[dæt]</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>hip</td>
<td>[hɪp]</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>hid</td>
<td>[hɪd]</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>team</td>
<td>[tɛm]</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>deem</td>
<td>[dɛm]</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>toot</td>
<td>[tʊt]</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>dude</td>
<td>[dʊd]</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>his</td>
<td>[hɪs]</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>his</td>
<td>[hɪz]</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>siz-</td>
<td>[sɪzɪl]</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>dizzy</td>
<td>[dɪzɪ]</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>tizzy</td>
<td>[tɪzɪ]</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td>17</td>
<td>70</td>
<td>17</td>
</tr>
</tbody>
</table>

The starting entropy, assuming word types as the relative unit of structure and counting, is:
\[
H_{1\text{-types}} = -\left(\frac{2}{17}\log_2\left(\frac{2}{17}\right) + \frac{4}{17}\log_2\left(\frac{4}{17}\right) + \frac{1}{17}\log_2\left(\frac{1}{17}\right) + \frac{5}{17}\log_2\left(\frac{5}{17}\right) + \frac{3}{17}\log_2\left(\frac{3}{17}\right) + \frac{9}{17}\log_2\left(\frac{9}{17}\right) + \frac{1}{17}\log_2\left(\frac{1}{17}\right) + \frac{17}{17}\log_2\left(\frac{17}{17}\right)\right) = 4.087
\]

The starting entropy, assuming word tokens, is:
\[
H_{1\text{-tokens}} = -\left(\frac{2}{70}\log_2\left(\frac{2}{70}\right) + \frac{4}{70}\log_2\left(\frac{4}{70}\right) + \frac{1}{70}\log_2\left(\frac{1}{70}\right) + \frac{5}{70}\log_2\left(\frac{5}{70}\right) + \frac{3}{70}\log_2\left(\frac{3}{70}\right) + \frac{9}{70}\log_2\left(\frac{9}{70}\right) + \frac{1}{70}\log_2\left(\frac{1}{70}\right) + \frac{70}{70}\log_2\left(\frac{70}{70}\right)\right) = 3.924
\]

Upon merger of [h] and [ŋ], there is no change in the number of unique words; there are still 17 unique words with all their same token frequencies. Thus, the entropy after an [h] / [ŋ] merger will be the same as it was before the merger. The functional load, then would be 0, as the pre-merger and post-merger entropies are identical.

Upon merger of [t] and [d], on the other hand, four pairs of words have been collapsed. E.g., the difference between team and deem no longer exists; there is now just one word, [Xim], where [X] represents the result of the merger. Thus, there are only 13 unique words, and while the total token frequency count remains the same, at 70, those 70 occurrences are divided among only 13 unique words instead of 17.
Thus, the entropy after a [t] / [d] merger, assuming word types, is:

\[
H_{1\text{-types}} = -\left(\frac{7}{10}\log_2\left(\frac{7}{10}\right) + \frac{1}{10}\log_2\left(\frac{1}{10}\right) + \frac{2}{10}\log_2\left(\frac{2}{10}\right) + \frac{3}{10}\log_2\left(\frac{3}{10}\right) + \frac{4}{10}\log_2\left(\frac{4}{10}\right)\right)
\]

And the entropy after a [t] / [d] merger, assuming word tokens, is:

\[
H_{1\text{-tokens}} = -\left(\frac{7}{11}\log_2\left(\frac{7}{11}\right) + \frac{4}{11}\log_2\left(\frac{4}{11}\right) + \frac{2}{11}\log_2\left(\frac{2}{11}\right) + \frac{3}{11}\log_2\left(\frac{3}{11}\right) + \frac{5}{11}\log_2\left(\frac{5}{11}\right)\right)
\]

\[
\Delta H = H_{1\text{-types}} - H_{2\text{-types}} = 0.087\times3.700 = 0.387
\]

And the functional load of [t] / [d] based on word tokens is:

\[
\Delta H = H_{1\text{-tokens}} - H_{2\text{-tokens}} = 3.924\times3.466 = 0.458
\]

11.2.2 (Relative) Minimal Pair Counts

The second means of calculating functional load that is included in PCT is a straight count of minimal pairs, which can be relativized to the number of words in the corpus that are potential minimal pairs—i.e. the number of words in the corpus with at least one of the target segments.

In the above example, the number of minimal pairs that hinge on [h] vs. [ŋ] is of course 0, so the functional load of [h] / [ŋ] is 0. The number of minimal pairs that hinge on [t] / [d] is 3, and the number of words with either [t] or [d] is 11; the functional load as a relativized minimal pair count would therefore be 3/11 = 0.273. Note that here, a relatively loose definition of minimal pair is used; specifically, two words are considered to be a minimal pair hinging on sounds A and B if, upon merger of A and B into a single symbol X, the words are identical. Thus, too-t and dude are considered a minimal pair on this definition, because they both become [XuX] upon merger of [t] and [d].

The resulting calculations of functional load are thus quite similar between the two measures, but the units are entirely different. Functional load based on change in entropy is measured in \textit{bits}, while functional load based on relativized minimal pair counts is simply a percentage. Also note that functional load based on minimal pairs is only based on type frequency; the frequency of the usage of the words is not used as a weighting factor, the way it can be under the calculation of functional load as change in entropy.

11.2.3 Average Functional Load

\cite{Hume2013} suggests that the average functional load (there called “relative contrastiveness”) is a useful way of indicating how much work an individual segment does, on average, in comparison to other segments. This is calculated by taking an individual segment, calculating the pairwise functional load of that segment and each other segment in the inventory, and then taking the average across all those pairs. This calculation can also be performed in PCT.

11.3 Calculating functional load in the GUI

As with most analysis functions, a corpus must first be loaded (see \textit{Loading in corpora}). Once a corpus is loaded, use the following steps.

1. \textbf{Getting started:} Choose “Analysis” / “Calculate functional load...” from the top menu bar.

2. \textbf{Sound selection:} First, decide whether you want to calculate the average functional load of a single segment (i.e., its functional load averaged across all possible pairwise comparisons), or the more standard functional load of a pair of sounds, defined over segments or features. To calculate the average functional load of a single sound, choose “Add one segment”; to calculate the pairwise functional load of two segments, choose “Add pair of segments”; to calculate the pairwise functional load based on features, choose “Add pair of features.”
For details on how to actually select segments or features, see Sound Selection or Feature Selection as relevant. When multiple individual segments or individual pairs are selected, each entry will be treated separately.

3. **Functional load algorithm**: Select which of the two methods of calculation you want to use—i.e., minimal pairs or change in entropy. (See discussion above for details of each.)

4. **Minimal pair options**: If minimal pairs serve as the means of calculation, there are three additional parameters can be set.

   (a) **Raw vs. relative count**: First, PCT can report only the raw count of minimal pairs that hinge on the contrast in the corpus, if you just want to know the scope of the contrast. On the other hand, the default is to relativize the raw count to the corpus size, by dividing the raw number by the number of lexical entries that include at least one instance of any of the target segments.

   (b) **Include vs. ignore homophones**: Second, PCT can either include homophones or ignore them. For example, if the corpus includes separate entries for the words *sock* (n.), *sock* (v.), *shock* (n.), and *shock* (v.), this would count as four minimal pairs if homophones are included, but only one if homophones are ignored. The default is to ignore homophones.

   (c) **Output list of minimal pairs to a file**: It is possible to save a list of all the actual minimal pairs that PCT finds that hinge on a particular chosen contrast to a .txt file. To do so, enter a file path name, or select “Choose file...” to use a regular system dialogue box. If nothing is entered here, no list will be saved, but the overall output will still be provided (and can be saved independently).

5. **Change in entropy options**: If you are calculating functional load using change in entropy, one additional parameter can be set.

   (a) **Type or token frequency**: As described in Change in entropy, entropy can be calculated using either type or token frequencies. This option determines which to use.

6. **Tier**: Select which tier the functional load should be calculated from. The default is the “transcription” tier, i.e., looking at the entire word transcriptions. If another tier has been created (see Creating new tiers in the corpus), functional load can be calculated on the basis of that tier. For example, if a vowel tier has been created, then “minimal pairs” will be entries that are identical except for one entry in the vowels only, entirely independently of consonants. Thus, the words [mapotik] and [Sôağefli] would be treated as a minimal pair, given that their vowel-tier representations are [aoi] and [aei].

7. **Pronunciation variants**: If the corpus contains multiple pronunciation variants for lexical items, select what strategy should be used. For details, see Pronunciation Variants.

8. **Minimum frequency**: It is possible to set a minimum token frequency for words in the corpus in order to be included in the calculation. This allows easy exclusion of rare words; for example, if one were calculating the functional load of [s] vs. [ʃ] in English and didn’t set a minimum frequency, words such as *santy* (vs. *shanty*) might be included, which might not be a particularly accurate reflection of the phonological knowledge of speakers. To include all words in the corpus, regardless of their token frequency, set the the minimum frequency to 0.

    Here is an example of selecting [m] and [n], with functional load to be calculated on the basis of minimal pairs, only including words with a token frequency of at least 1, from the built-in example corpus (which only has canonical forms):
10. **Results**: Once all parameters have been set, click one of the two “Calculate functional load” buttons. If this is the first calculation, the option to “start new results table” should be selected. For subsequent calculations, the calculation can be added to the already started table, for direct comparison, or a new table can be started.

**Note**: that if a table is closed, new calculations will not be added to the previously open table; a new table must be started.

Either way, the results table will have the following columns, with one row per calculation: segment 1, segment 2, which tier was used, which measurement method was selected, the resulting functional load, what the minimum frequency was, what strategy was used for dealing with pronunciation variants, and for calculations using minimal pairs, whether the count is absolute or relative and whether homophones were ignored or not. (For calculations using change in entropy, “N/A” values are entered into the latter two columns.)

11. **Saving results**: Once a results table has been generated for at least one pair, the table can be saved by clicking on “Save to file” at the bottom of the table to open a system dialogue box and save the results at a user-designated location.

**Note**: that in the above screen shot, not all columns are visible; they are visible only by scrolling over to the right, due to constraints on the window size. All columns would be saved to the results file.)

To return to the function dialogue box with your most recently used selections, click on “Reopen function dialog.” Otherwise, the results table can be closed and you will be returned to your corpus view.

11.3. **Calculating functional load in the GUI**
11.4 Implementing the functional load function on the command line

In order to perform this analysis on the command line, you must enter a command in the following format into your Terminal:

```
pct_funcload CORPUSFILE [additional arguments]
```

...where CORPUSFILE is the name of your *.corpus file. If calculating FL from a file of segment pairs, it must list the pairs of segments whose functional load you wish to calculate with each pair separated by a tab (\t) and one pair on each line. Note that you must either specify a file or segment (using \-p) or request the functional loads of all segment pairs in the inventory (using \-l). You may also use command line options to change various parameters of your functional load calculations. Descriptions of these arguments can be viewed by running `pct_funcload -h` or `pct_funcload --help`. The help text from this command is copied below, augmented with specifications of default values:

Positional arguments:

```
corpus_file_name
  Name of corpus file
```

Mandatory argument group (call must have one of these two):

```
-p PAIRS_FILE_NAME_OR_SEGMENT
--pairs_file_name_or_segment PAIRS_FILE_NAME_OR_SEGMENT
  Name of file with segment pairs (or target segment if relative fl is True)
```

```
-l
--all_pairwise_fl
  Flag: calculate FL for all pairs of segments
```

Optional arguments:

```
-h
--help
  Show help message and exit
```

```
-c CONTEXT_TYPE
--context_type CONTEXT_TYPE
  How to deal with variable pronunciations. Options are ‘Canonical’, ‘MostFrequent’, ‘SeparatedTokens’, or ‘Weighted’. See documentation for details.
```

```
-a ALGORITHM
--algorithm ALGORITHM
  Algorithm to use for calculating functional load: “minpair” for minimal pair count or “deltah” for change in entropy. Defaults to minpair.
```

```
-f FREQUENCY_CUTOFF
--frequency_cutoff FREQUENCY_CUTOFF
  Minimum frequency of words to consider as possible minimal pairs or contributing to lexicon entropy.
```

```
-d Distinguishing_HOMOPHONES
--distinguish_homophones Distinguishing_HOMOPHONES
  For minimal pair FL: if False, then you’ll count sock~shock (sock=clothing) and sock~shock (sock=punch) as just one minimal pair; but if True, you’ll overcount alternative spellings of the same word, e.g. axel~actual and axle~actual. False is the value used by Wedel et al.
```

```
-t TYPE_OR_TOKEN
--type_or_token TYPE_OR_TOKEN
  For change in entropy FL: specifies whether entropy is based on type or token frequency.
```
-e RELATIVE_FL
--relative_fl RELATIVE_FL
  If True, calculate the relative FL of a single segment by averaging across the functional loads of it and all other segments.

-s SEQUENCE_TYPE
--sequence_type SEQUENCE_TYPE
  The attribute of Words to calculate FL over. Normally this will be the transcription, but it can also be the spelling or a user-specified tier.

-o OUTFILE
--outfile OUTFILE
  Name of output file

EXAMPLE 1: If your corpus file is example.corpus (no pronunciation variants) and you want to calculate the minimal pair functional load of the segments [m] and [n] using defaults for all optional arguments, you first need to create a text file that contains the text m\t\n (where \t is a tab). Let us call this file pairs.txt. You would then run the following command in your terminal window:

```
pct_funcload example.corpus -p pairs.txt
```

EXAMPLE 2: Suppose you want to calculate the relative (average) functional load of the segment [m]. Your corpus file is again example.corpus. You want to use the change in entropy measure of functional load rather than the minimal pairs measure, and you also want to use type frequency instead of (the default value of) token frequency. In addition, you want the script to produce an output file called output.txt. You would need to run the following command:

```
pct_funcload example.corpus -p m -a deltah -t type -o output.txt
```

EXAMPLE 3: Suppose you want to calculate the functional loads of all segment pairs. Your corpus file is again example.corpus. All other parameters are set to defaults. In addition, you want the script to produce an output file called output.txt. You would need to run the following command:

```
pct_funcload example.corpus -l -o output.txt
```

### 11.5 Classes and functions

For further details about the relevant classes and functions in PCT’s source code, please refer to *Functional load*.
Predictability of Distribution

12.1 About the function

Predictability of distribution is one of the common methods of determining whether or not two sounds in a language are contrastive or allophonic. The traditional assumption is that two sounds that are predictably distributed (i.e., in complementary distribution) are allophonic, and that any deviation from complete predictability of distribution means that the two sounds are contrastive. [Hall2009], [Hall2012] proposes a way of quantifying predictability of distribution in a gradient fashion, using the information-theoretic quantity of entropy (uncertainty), which is also used for calculating functional load (see Method of calculation), which can be used to document the degree to which sounds are contrastive in a language. This has been shown to be useful in, e.g., documenting sound changes [Hall2013b], understanding the choice of epenthetic vowel in a languages [Hume2013], modeling intra-speaker variability [Thakur2011], gaining insight into synchronic phonological patterns [Hall2013a], and understanding the influence of phonological relations on perception ([Hall2009], [Hall2014a]). See also the related measure of Kullback-Leibler divergence (Kullback-Leibler Divergence), which is used in [Peperkamp2006] and applied to acquisition; it is also a measure of the degree to which environments overlap, but the method of calculation differs (especially in terms of environment selection).

It should be noted that predictability of distribution and functional load are not the same thing, despite the fact that both give a measure of phonological contrast using entropy. Two sounds could be entirely unpredictably distributed (perfectly contrastive), and still have either a low or high functional load, depending on how often that contrast is actually used in distinguishing lexical items. Indeed, for any degree of predictability of distribution, the functional load may be either high or low, with the exception of the case where both are 0. That is, if two sounds are entirely predictably distributed, and so have an entropy of 0 in terms of distribution, then by definition they cannot be used to distinguish between any words in the language, and so their functional load, measured in terms of change in entropy upon merger, would also be 0.

12.2 Method of calculation

As mentioned above, predictability of distribution is calculated using the same entropy formula as above, repeated here below, but with different inputs.

Entropy:

\[ H = - \sum_{i \in N} p_i \log_2(p_i) \]

Because predictability of distribution is determined between exactly two sounds, \( i \) will have only two values, that is, each of the two sounds. Because of this limitation to two sounds, entropy will range in these situations between 0 and 1. An entropy of 0 means that there is 0 uncertainty about which of the two sounds will occur; i.e., they are perfectly predictably distributed (commonly associated with being allophonic). This will happen when one of the two sounds has a probability of 1 and the other has a probability of 0. On the other hand, an entropy of 1 means that there
is complete uncertainty about which of the two sounds will occur; i.e., they are in perfectly overlapping distribution (what might be termed “perfect” contrast). This will happen when each of the two sounds has a probability of 0.5.

Predictability of distribution can be calculated both within an individual environment and across all environments in the language; these two calculations are discussed in turn.

### 12.2.1 Predictability of Distribution in a Single Environment

For any particular environment (e.g., word-initially; between vowels; before a [+ATR] vowel with any number of intervening consonants; etc.), one can calculate the probability that each of two sounds can occur. This probability can be calculated using either types or tokens, just as was the case with functional load. Consider the following toy data, which is again repeated from the examples of functional load, though just the original distribution of sounds.

<table>
<thead>
<tr>
<th>Word</th>
<th>Original</th>
<th>Type Freq.</th>
<th>Token Freq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>hot</td>
<td>[hA]</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>song</td>
<td>[sAN]</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>hat</td>
<td>[hAT]</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sing</td>
<td>[sIN]</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>tot</td>
<td>[tAT]</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>dot</td>
<td>[dAT]</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>hip</td>
<td>[hIP]</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>hid</td>
<td>[hID]</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>team</td>
<td>[tIM]</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>deem</td>
<td>[dIM]</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>toot</td>
<td>[tUT]</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>dude</td>
<td>[dUD]</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>biss</td>
<td>[hIS]</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>his</td>
<td>[hIZ]</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>sizzle</td>
<td>[szI]</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>dizzy</td>
<td>[dizI]</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>tizzy</td>
<td>[tizI]</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>17</td>
<td>70</td>
</tr>
</tbody>
</table>

Consider the distribution of [h] and [n], word-initially. In this environment, [h] occurs in 6 separate words, with a total token frequency of 20. [n] occurs in 0 words, with, of course, a token frequency of 0. The probability of [h] occurring in this position as compared to [n], then, is 6/6 based on types, or 20/20 based on tokens. The entropy of this pair of sounds in this context, then, is:

\[
H\text{\_types/\_tokens} = -[1\log_2(1) + 0\log_2(0)] = 0
\]

Similar results would obtain for [h] and [n] in word-final position, except of course that it’s [n] and not [h] that can appear in this environment.

For [t] and [d] word-initially, [t] occurs 4 words in this environment, with a total token frequency of 21, and [d] also occurs in 4 words, with a total token frequency of 15. Thus, the probability of [t] in this environment is 4/8, counting types, or 21/36, counting tokens, and the probability of [d] in this environment is 4/8, counting types, or 15/36, counting tokens. The entropy of this pair of sounds is therefore:

\[
H_{\text{types}} = -\left[\left(\frac{4}{8}\log_2\left(\frac{4}{8}\right)\right) + \left(\frac{4}{8}\log_2\left(\frac{4}{8}\right)\right)\right] = 1
\]

\[
H_{\text{types}} = -\left(\frac{21}{36}\log_2\left(\frac{21}{36}\right)\right)\left(\frac{15}{36}\log_2\left(\frac{15}{36}\right)\right) = 0.98
\]

In terms of what environment(s) are interesting to examine, that is of course up to individual researchers. As mentioned in the preface to Predictability of Distribution, these functions are just tools. It would be just as possible to calculate the entropy of [t] and [d] in word-initial environments before [n], separately from word-initial environments before [u]. Or one could calculate the entropy of [t] and [d] that occur anywhere in a word before a bilabial nasal...etc., etc.
The choice of environment should be phonologically informed, using all of the resources that have traditionally been used to identify conditioning environments of interest. See also the caveats in the following section that apply when one is calculating systemic entropy across multiple environments.

### 12.2.2 Predictability of Distribution across All Environments (Systemic Entropy)

While there are times in which knowing the predictability of distribution within a particular environment is helpful, it is generally the case that phonologists are more interested in the relationship between the two sounds as a whole, across all environments. This is achieved by calculating the weighted average entropy across all environments in which at least one of the two sounds occurs.

As with single environments, of course, the selection of environments for the systemic measure need to be phonologically informed. There are two further caveats that need to be made about environment selection when multiple environments are to be considered, however: (1) **exhaustivity** and (2) **uniqueness**.

With regard to **exhaustivity**: In order to calculate the total predictability of distribution of a pair of sounds in a language, one must be careful to include all possible environments in which at least one of the sounds occurs. That is, the total list of environments needs to encompass all words in the corpus that contain either of the two sounds; otherwise, the measure will obviously be incomplete. For example, one would not want to consider just word-initial and word-medial positions for [h] and [ŋ]; although the answer would in fact be correct (they have 0 entropy across these environments), it would be for the wrong reason—i.e., it ignores what happens in word-final position, where they *could* have had some other distribution.

With regard to **uniqueness**: In order to get an *accurate* calculation of the total predictability of distribution of a pair of sounds, it is important to ensure that the set of environments chosen do not overlap with each other, to ensure that individual tokens of the sounds are not being counted multiple times. For example, one would not want to have both [#__] and [__i] in the environment list for [t]/[d] while calculating systemic entropy, because the words *team* and *deem* would appear in both environments, and the sounds would (in this case) appear to be “more contrastive” (less predictably distributed) than they might otherwise be, because the contrasting nature of these words would be counted twice.

To be sure, one can calculate the entropy in a set of individual environments that are non-exhaustive and/or overlapping, for comparison of the differences in possible generalizations. But, in order to get an accurate measure of the total predictability of distribution, the set of environments must be both exhaustive and non-overlapping. As will be described below, PCT will by default check whether any set of environments you provide does in fact meet these characteristics, and will throw a warning message if it does not.

It is also possible that there are multiple possible ways of developing a set of exhaustive, non-overlapping environments. For example, “word-initial” vs. “non-word-initial” would suffice, but so would “word-initial” vs. “word-medial” vs. “word-final.” Again, it is up to individual researchers to determine which set of environments makes the most sense for the particular phenomenon they are interested in. See [Hall2012] for a comparison of two different sets of possible environments in the description of Canadian Raising.

Once a set of exhaustive and non-overlapping environments has been determined, the entropy in each individual environment is calculated, as described in *Predictability of Distribution in a Single Environment*. The frequency of each environment itself is then calculated by examining how many instances of the two sounds occurred in each environment, as compared to all other environments, and the entropy of each environment is weighted by its frequency. These frequency-weighted entropies are then summed to give the total average entropy of the sounds across the environments. Again, this value will range between 0 (complete predictability; no uncertainty) and 1 (complete unpredictability; maximal uncertainty). This formula is given below; e represents each individual environment in the exhaustive set of non-overlapping environments.

Formula for systemic entropy:

\[ H_{total} = - \sum_{e \in E} H(e) \cdot p(e) \]

As an example, consider [t]/[d]. One possible set of exhaustive, non-overlapping environments for this pair of sounds is (1) word-initial and (2) word-final. The relevant words for each environment are shown in the table below, along
with the calculation of systemic entropy from these environments.

The calculations for the entropy of word-initial environments were given above; the calculations for word-final environments are analogous.

To calculate the probability of the environments, we simply count up the number of total words (either types or tokens) that occur in each environment, and divide by the total number of words (types or tokens) that occur in all environments.

Calculation of systemic entropy of [t] and [d]:

<table>
<thead>
<tr>
<th></th>
<th>words</th>
<th>Types</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H(e)</td>
<td>p(e)</td>
<td>p(e) * H(e)</td>
<td>H(e)</td>
</tr>
<tr>
<td>#_#</td>
<td>tot, team, toot, tizzy</td>
<td>dot, dude, deem, dizzy</td>
<td>1</td>
</tr>
<tr>
<td>__#</td>
<td>hot, hat, tot, dot, toot</td>
<td>hid, dude</td>
<td>0.863</td>
</tr>
<tr>
<td>V_V</td>
<td>sizzle, dizzy, tizzy</td>
<td>0</td>
<td>3/8</td>
</tr>
</tbody>
</table>

In this case, [t]/[d] are relatively highly unpredictably distributed (contrastive) in both environments, and both environments contributed approximately equally to the overall measure. Compare this to the example of [s]/[z], shown below.

Calculation of systemic entropy of [s] and [z]:

<table>
<thead>
<tr>
<th></th>
<th>words</th>
<th>Types</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H(e)</td>
<td>p(e)</td>
<td>p(e) * H(e)</td>
<td>H(e)</td>
</tr>
<tr>
<td>#_#</td>
<td>song, sing, sizzle</td>
<td>0</td>
<td>3/8</td>
</tr>
<tr>
<td>__#</td>
<td>hiss</td>
<td>his</td>
<td>1</td>
</tr>
<tr>
<td>V_V</td>
<td>sizzle, dizzy, tizzy</td>
<td>0</td>
<td>3/8</td>
</tr>
</tbody>
</table>

In this case, there is what would traditionally be called a contrast word finally, with the minimal pair *hiss* vs. *his*; this contrast is neutralized (made predictable) in both word-initial position, where [s] occurs but [z] does not, and intervocalic position, where [z] occurs but [s] does not. The three environments are roughly equally probable, though the environment of contrast is somewhat less frequent than the environments of neutralization. The overall entropy of the pair of sounds is on around 0.25, clearly much closer to perfect predictability (0 entropy) than [t]/[d].

Note, of course, that this is an entirely fictitious example—that is, although these are real English words, one would **not** want to infer anything about the actual relationship between either [t]/[d] or [s]/[z] on the basis of such a small corpus. These examples are simplified for the sake of illustrating the mathematical formulas!
### 12.2.3 “Predictability of Distribution” Across All Environments (i.e., Frequency-Only Entropy)

Given that the calculation of predictability of distribution is based on probabilities of occurrence across different environments, it is also possible to calculate the overall entropy of two segments using their raw probabilities and ignoring specific environments. Note that this doesn’t really reveal anything about predictability of distribution per se; it simply gives the uncertainty of occurrence of two segments that is related to their relative frequencies. This is calculated by simply taking the number of occurrences of each of sound 1 (N1) and sound 2 (N2) in the corpus as a whole, and then applying the following formula:

Formula for frequency-only entropy:

\[
H = (-1) \times \left[ \frac{N1}{N1+N2} \log_2 \left( \frac{N1}{N1+N2} \right) + \frac{N2}{N1+N2} \log_2 \left( \frac{N2}{N1+N2} \right) \right]
\]

The entropy will be 0 if one or both of the sounds never occur(s) in the corpus. The entropy will be 1 if the two sounds occur with exactly the same frequency. It will be a number between 0 and 1 if both sounds occur, but not with the same frequency.

Note that an entropy of 1 in this case, which was analogous to perfect contrast in the environment-specific implementation of this function, does not align with contrast here. For example, [h] and [ŋ] in English, which are in complementary distribution, could theoretically have an entropy of 1 if environments are ignored and they happened to occur with exactly the same frequency in some corpus. Similarly, two sounds that do in fact occur in the same environments might have a low entropy, close to 0, if one of the sounds is vastly more frequent than the other. That is, this calculation is based ONLY on the frequency of occurrence, and not actually on the distribution of the sounds in the corpus. This function is thus useful only for getting a sense of the frequency balance / imbalance between two sounds. Note that one can also get total frequency counts for any segment in the corpus through the “Summary” information feature (Summary information about a corpus).

### 12.3 Calculating predictability of distribution in the GUI

Assuming a corpus has been opened or created, predictability of distribution is calculated using the following steps.

1. **Getting started**: Choose “Analysis” / “Calculate predictability of distribution...” from the top menu bar.

2. **Segments**: First, select which pairs of sounds you want the predictability of distribution to be calculated for. There are two options for this. First is to add individual pairs of sounds. Do this by clicking on “Add pair of sounds”; the “Select segment pair” dialogue box will open. The order that the sounds are selected in is irrelevant; picking [i] first and [u] second will yield the same results as picking [u] first and [i] second. See more about interacting with the sound selection box (including, e.g., the use of features in selecting sounds and the options for selecting multiple pairs) in [Sound Selection](#).

The second alternative is to select pairs of sounds based on shared vs. contrasting features. This option allows you, for example, to test the predictability of distribution of the front/back contrast in vowels, regardless of vowel height. To do this, click on “Add pair of features”; the “Select feature pair” dialogue box will open. See [Feature Selection](#) for more information on using this interface.

Once sounds have been selected, click “Add.” Pairs will appear in the “Predictability of distribution” dialogue box.

3. **Environments**: Click on “New environment” to add an environment in which to calculate predictability of distribution. See [Environment Selection](#) for details on how to use this interface. Note that you will not be able to edit the “target” segments in this function, because the targets are automatically populated from the list of pairs selected on the left-hand side.

**Note**: If NO environments are added, PCT will calculate the overall predictability of distribution of the two sounds based only on their frequency of occurrence. This will simply count the frequency of each sound in the pair and calculate the entropy based on those frequencies (either type or token). See below for an example of calculating environment-free entropy for four different pairs in the sample corpus:
4. **Environment list:** Individual environments from the list can be selected and edited or removed if it is determined that an environment needs to be changed. It is this list that PCT will verify as being both exhaustive and unique; i.e., the default is that the environments on this list will exhaustively cover all instances in your corpus of the selected sounds, but will do so in such a way that each instance is counted exactly once.

5. **Tier:** Under “Options,” first pick the tier on which you want predictability of distribution to be calculated. The default is for the entire transcription to be used, such that environments are defined on any surrounding segments. If a separate tier has been created as part of the corpus (see Creating new tiers in the corpus), however, predictability of distribution can be calculated on this tier. For example, one could extract a separate tier that contains only vowels, and then calculate predictability of distribution based on this tier. This makes it much easier to define non-adjacent contexts. For instance, if one wanted to investigate the extent to which [i] and [u] are predictably distributed before front vs. back vowels, it will be much easier to to specify that the relevant environments are _[+back] and _[-back] on the vowel tier than to try to account for possible intervening segments on the entire transcription tier.

6. **Pronunciation variants:** If the corpus contains multiple pronunciation variants for lexical items, select what strategy should be used. For details, see Pronunciation Variants.

7. **Type vs. Token Frequency:** Next, pick whether you want the calculation to be done on types or tokens, assuming that token frequencies are available in your corpus. If they are not, this option will not be available. (Note: if you think your corpus does include token frequencies, but this option seems to be unavailable, see Required format of corpus on the required format for a corpus.)

8. **Exhaustivity & Uniqueness:** The default is for PCT to check for both exhaustivity and uniqueness of environments, as described above in Predictability of Distribution across All Environments (Systemic Entropy). Un-checking this box will turn off this mechanism. For example, if you wanted to compare a series of different possible environments, to see how the entropy calculations differ under different generalizations, uniqueness might not be a concern. Keep in mind that if uniqueness and exhaustivity are not met, however, the calculation of systemic entropy will be inaccurate.

   (a) If you ask PCT to check for exhaustivity, and it is not met, an error message will appear that warns you that the environments you have selected do not exhaustively cover all instances of the symbols in the corpus, as in the following; the “Show details...” option has been clicked to reveal the specific words that occur in the corpus that are not currently covered by your list of environments. Furthermore, a .txt file is automatically created that lists all of the words, so that the environments can be easily adjusted. This file is stored in the ERRORS folder within the working directory that contains the PCT software (see also Preferences), and can be accessed directly by clicking “Open errors directory.” If exhaustivity is not important, and only the entropy in individual environments matters, then it is safe to not enforce exhaustivity; it should be noted that the weighted average entropy across environments will NOT be accurate in this scenario, because not all words have been included.
(b) If you ask PCT to check for uniqueness, and it is not met, an error message will appear that indicates that the environments are not unique, as shown below. Furthermore, a .txt file explaining the error and listing all the words that are described by multiple environments in your list is created automatically and stored in the ERRORS folder within the working directory that contains the PCT software. Clicking “Show details” in the error box also reveals this information.

Here’s an example of correctly exhaustive and unique selections for calculating the predictability of distribution based on token frequency for [s] and [ʃ] in the example corpus (note that the environments were selected using features, e.g., _#, _[-voc], _[+voc, -high], _[+voc, +high], even though they appear as sets of segments in the environments):
8. **Entropy calculation / results**: Once all environments have been specified, click “Calculate predictability of distribution.” If you want to start a new results table, click that button; if you’ve already done at least one calculation and want to add new calculations to the same table, select the button with “add to current results table.” Results will appear in a pop-up window on screen. The last row for each pair gives the weighted average entropy across all selected environments, with the environments being weighted by their own frequency of occurrence. See the following example (noting that not all columns in the result file are visible on screen):

![Entropy Calculation Results](image)

9. **Output file / Saving results**: If you want to save the table of results, click on “Save to file” at the bottom of the table. This opens up a system dialogue box where the directory and name can be selected.

   To return to the function dialogue box with your most recently used selections, click on “Reopen function dialog.” Otherwise, the results table can be closed and you will be returned to your corpus view.

### 12.4 Classes and functions

For further details about the relevant classes and functions in PCT’s source code, please refer to *Predictability of distribution*. 
13.1 About the function

Another way of measuring the distribution of environments as a proxy for phonological relationships is the Kullback-Leibler (KL) measure of the dissimilarity between probability distributions [Kullback1951]. Sounds that are distinct phonemes appear in the same environments, that is, there are minimal or near-minimal, pairs. Allophones, on the other hand, have complementary distribution, and never appear in the same environment. Distributions that are identical have a KL score of 0, and the more dissimilar two distributions, the higher the KL score. Applied to phonology, the idea is to calculate the probability of two sounds across all environments in a corpus, and use KL to measure their dissimilarity. Scores close to 0 suggest that the two sounds are distinct phonemes, since they occur in many of the same environments (or else there is extensive free variation). Higher scores represent higher probabilities that the two sounds are actually allophones. Since KL scores have no upper bound, it is up to the user to decide what counts as “high enough” for two sounds to be allophones (this is unlike the predictability of distribution measure described in Predictability of Distribution). See [Peperkamp2006] for a discussion of how to use Z-Scores to make this discrimination.

As with the predictability of distribution measure in Predictability of Distribution, spurious allophony is also possible, since many sounds happen to have non-overlapping distributions. As a simple example, vowels and consonants generally have high KL scores, because they occur in such different environments. Individual languages might have cases of accidental complementary distribution too. For example, in English /h/ occurs only initially and [ŋ] only occurs finally. However, it is not usual to analyze them as being in allophones of a single underlying phonemes. Instead, there is a sense that allophones need to be phonetically similar to some degree, and /h/ and /ŋ/ are simply too dissimilar.

To deal with this problem, [Peperkamp2006] suggest two “linguistic filters” that can be applied, which can help identify cases of spurious allophones, such as /h/ and /ŋ/. Their filters do not straightforwardly apply to CorpusTools, since they use 5-dimensional vectors to represent sounds, while in CorpusTools most sounds have only binary features. An alternative filter is used instead, and it is described below.

It is important to note that this function’s usefulness depends on the level of analysis in your transcriptions. In many cases, corpora are transcribed at a phonemic level of detail, and KL will not be very informative. For instance, the IPHOD corpus does not distinguish between aspirated and unaspirated voiceless stops, so you cannot measure their KL score.

13.2 Method of calculation

All calculations were adopted from [Peperkamp2006]. The variables involved are as follows: s is a segment, c is a context, and C is the set of all contexts. The Kullback-Leibler measure of dissimilarity between the distributions of two segments is the sum for all contexts of the entropy of the contexts given the segments:

KL Divergence:
The notation \( P(c|s) \) means the probability of context \( c \) given segment \( s \), and it is calculated as follows:

\[
P(c|s) = \frac{n(c,s) + 1}{n(s) + N}
\]

...where \( n(c,s) \) is the number of occurrences of segments \( s \) in context \( c \). [Peperkamp2006] note that this equal to the number of occurrences of the sequence \( sc \), which suggests that they are only looking at the right hand environment. This is probably because in their test corpora, they were looking at allophones conditioned by the following segment. PCT provides the option to look only at the left-hand environment, only at the right-hand environment, or at both. [Peperkamp2006] then compare the average entropy values of each segment, in the pair. The segment with the higher entropy is considered to be a surface representation (SR), i.e. an allophone, while the other is the underlying representation (UR). In a results window in PCT, this is given as “Possible UR.” More formally:

\[
SR = \max_{SR,UR} \left[ \sum_{c} P(c|s) \log \frac{P(c|s)}{P(c)} \right]
\]

[Peperkamp2006] give two linguistic filters for getting rid of spurious allophones, which rely on sounds be coded as 5-dimensional vectors. In PCT, this concept as been adopted to deal with binary features. The aim of the filter is the same, however. In a results window the column labeled “spurious allophones” gives the result of applying this filter.

The features of the supposed UR and SR are compared. If they differ by only one feature, they are considered plausibly close enough to be allophones, assuming the KL score is high enough for this to be reasonable (which will depend on the corpus and the user’s expectations). In this case, the “spurious allophones?” results will say ‘No.’

If they differ by more than 1 feature, PCT checks to see if there any other sounds in the corpus that are closer to the SR than the UR is. For instance, if /p/ and /s/ are compared in the IPHOD corpus, /p/ is considered the UR and /s/ is the SR. The two sounds differ by two features, namely [continuant] and [coronal]. There also exists another sound, /t/, which differs from /s/ by [continuant], but not by [coronal] (or any other feature). In other words, /t/ is more similar to /s/ than /p/ is to /s/. If such an “in-between” sound can be found, then in the “spurious allophones?” column, the results will say ‘Yes.’

If the two sounds differ by more than 1 feature, but no in-between sound can be found, then the “spurious allophones?” results will say ‘Maybe.’

Note too that a more direct comparison of the acoustic similarity of sounds can also be conducted using the functions in Acoustic Similarity.

### 13.3 Calculating Kullback-Leibler Divergence in the GUI

To implement the KL function in the GUI, select “Analysis” / “Calculate Kullback-Leibler...” and then follow these steps:

1. **Pair of sounds:** First, select which pairs of sounds you want the functional load to be calculated for. Do this by clicking on either “Add pair of sounds” or “Add pair of features” – use the former for selecting segments (even if the segments are chosen using features); use the latter for selecting featural differences to calculate KL divergence for (e.g., the KL score for [+/-high]). See Sound Selection or Feature Selection for more on how to interact with these options.

2. **Tier:** Select which tier the KL-divergence should be calculated from. The default is the “transcription” tier, i.e., looking at the entire word transcriptions. If another tier has been created (see Creating new tiers in the corpus), KL can be calculated on the basis of that tier. For example, if a vowel tier has been created, then the sounds will be considered only in terms of their adjacent vowels, ignoring intervening consonants.

3. **Pronunciation variants:** If the corpus contains multiple pronunciation variants for lexical items, select what strategy should be used. For details, see Pronunciation Variants.

4. **Type or token frequency:** Select whether probabilities should be based on type or token frequencies.
5. **Contexts**: Using KL requires a notion of “context,” and there are three options: left, right, or both. Consider the example word [atema]. If using the “both” option, then this word consists of these environments: [t#], [a_e], [t_m], [e_a], and [m_#]. If the left-side option is chosen, then only the left-hand side is used, i.e., the word consists of the environments [t#], [a_], [t_], [e_], and [m_]. If the right-side option is chosen, then the environments in the word are [t#], [a_], [t_], [e_], and [m_]. Note that the word boundaries don’t count as elements of words, but can count as parts of environments.

6. **Results**: Once all selections have been made, click “Calculate Kullback-Leibler.” If you want to start a new results table, click that button; if you’ve already done at least one calculation and want to add new calculations to the same table, select the button with “add to current results table.” Results will appear in a pop-up window on screen. Each member of the pair is listed, along with which context was selected, what tier was used, what strategy was used for pronunciation variants, what kind of frequency was used, the entropy of each segment, the KL score, which of the two members of the pair is more likely to be the UR (as described above), and PCT’s judgment as to whether this is a possible case of spurious allophones based on the featural distance.

7. **Output file / Saving results**: If you want to save the table of results, click on “Save to file” at the bottom of the table. This opens up a system dialogue box where the directory and name can be selected.

   To return to the function dialogue box with your most recently used selections, click on “Reopen function dialog.” Otherwise, the results table can be closed and you will be returned to your corpus view.

An example of calculating the KL scores in the Example corpus (which has canonical forms only), with the sounds [s], [ʃ], [t], [n], [m], [e], [u] selected (and therefore all pairwise comparisons thereof calculated), examining only right-hand side contexts:

The “Select segment pair” dialogue box, within the “Kullback-Leibler” dialogue box:
The “Kullback-Leibler” dialogue box, with pairs of sounds and contexts selected:
13.4 Implementing the KL-divergence function on the command line

In order to perform this analysis on the command line, you must enter a command in the following format into your Terminal:

```
pct_funcload CORPUSFILE [additional arguments]
```

...where CORPUSFILE is the name of your *.corpus file. If calculating FL from a file of segment pairs, it must list the pairs of segments whose functional load you wish to calculate with each pair separated by a tab (\t) and one pair on each line. Note that you must either specify a file or segment (using -p) or request the functional loads of all segment pairs in the inventory (using -l). You may also use command line options to change various parameters of your functional load calculations. Descriptions of these arguments can be viewed by running `pct_funcload -h` or `pct_funcload --help`. The help text from this command is copied below, augmented with specifications of
default values:

Positional arguments:

**corpus_file_name**
Name of corpus file

**seg1**
First segment

**seg2**
Second segment

**side**
Context to check. Options are ‘right’, ‘left’ and ‘both’. You can enter just the first letter.

Optional arguments:

**-h**
--help
Show help message and exit

**-s** SEQUENCE_TYPE
--sequence_type SEQUENCE_TYPE
The attribute of Words to calculate KL-divergence over. Normally this will be the transcription, but it can also be the spelling or a user-specified tier.

**-t** TYPE_OR_TOKEN
--type_or_token TYPE_OR_TOKEN
Specifies whether quantifications are based on type or token frequency.

**-c** CONTEXT_TYPE
--context_type CONTEXT_TYPE
How to deal with variable pronunciations. Options are ‘Canonical’, ‘MostFrequent’, ‘SeparatedTokens’, or ‘Weighted’. See documentation for details.

**-o** OUTFILE
--outfile OUTFILE
Name of output file

**EXAMPLE 1:** If your corpus file is example.corpus (no pronunciation variants) and you want to calculate the KL-divergence of the segments [m] and [n] considering contexts on both sides and using defaults for all optional arguments, you would run the following command in your terminal window:

```
pct_kl example.corpus m n both
```

### 13.5 Classes and functions

For further details about the relevant classes and functions in PCT’s source code, please refer to *Kullback-Leibler divergence*. 

---

90 Chapter 13. Kullback-Leibler Divergence
14.1 About the function

String similarity is any measure of how similar any two sequences of characters are. These character strings can be strings of letters or phonemes; both of the methods of calculation included in PCT allow for calculations using either type of character. It is, therefore, a basic measure of overall form-based similarity.

String similarity finds more widespread use in areas of linguistics other than phonology; it is, for example, used in Natural Language Processing applications to determine, for example, possible alternative spellings when a word has been mistyped. It is, however, also useful for determining how phonologically close any two words might be.

String similarity could be part of a calculation of morphological relatedness, if used in conjunction with a measure of semantic similarity (see, e.g., [Hall2014b]). In particular, it can be used in conjunction with the Frequency of Alternation function of PCT (see Frequency of alternation) as a means of calculating the frequency with which two sounds alternate with each other in a language.

Some measure of string similarity is also used to calculate neighbourhood density (e.g. [Greenberg1964]; [Luce1998]; [Yao2011]), which has been shown to affect phonological processing. A phonological “neighbour” of some word X is a word that is similar in some close way to X. For example, it might differ by maximally one phone (through deletion, addition, or substitution) from X. X’s neighbourhood density, then, is the number of words that fit the criterion for being a neighbour.

14.2 Method of calculation

14.2.1 Levenshtein Edit Distance

Edit distance is defined as the minimum number of one-symbol deletions, additions, and substitutions necessary to turn one string into another. For example, turn and burn would have an edit distance of 1, as the only change necessary is to turn the <t> into a <b>, while the edit distance between turn and surfs would be 3, with <t> becoming <s>, <n> becoming <f>, and ∅ becoming <ss> at the end of the word. All such one-symbol changes are treated as equal in Levenshtein edit distance, unlike phonological edit distance, described in the following section. Generally speaking, the neighbourhood density of a particular lexical item is measured by summing the number of lexical items that have an edit distance of 1 from that item [Luce1998].

14.2.2 Phonological Edit Distance

Phonological edit distance is quite similar to Levenshtein edit distance, in that it calculates the number of one-symbol changes between strings, but it differs in that changes are weighted based on featural similarity. For example, depend-
ing on the feature system used, changing <t> to <s> might involve a single feature change (from [-cont] to [+cont]), while changing <t> to <b> might involve two (from [-voice, +cor] to [+voice, -cor]). By default, the formula for calculating the phonological distance between two segments—or between a segment and “silence”, i.e. insertion or deletion—is the one used in the Sublexical Learner [Allen2014]. When comparing two segments, the distance between them is equal to the sum of the distances between each of their feature values: the distance between two feature values that are identical is 0, while the distance between two opposing values (+/- or -/+ is 1, and the distance between two feature values in the case that just one of them is 0 (unspecified) is set to by default to 0.25. When comparing a segment to “silence” (insertion/deletion), the silence is given feature values of 0 for all features and then compared to the segment as normal.

14.2.3 Khorsi (2012) Similarity Metric

Khorsi (2012) proposes a particular measure of string similarity based on orthography, which he suggests can be used as a direct measure of morphological relatedness. PCT allows one to calculate this measure, which could be used, as Khorsi describes, on its own, or could be used in conjunction with other measures (e.g., semantic similarity) to create a more nuanced view.

This measure starts with the sum of the log of the inverse of the frequency of occurrence of each of the letters in the longest common shared sequence between two words, and then subtracts the sum of the log of the inverse of the frequency of the letters that are not shared, as shown below.

Formula for string similarity from [Khorsi2012]:

\[
\sum_{i=1}^{||LCS(w_1, w_2)||} \log \left( \frac{1}{\text{freq}(LCS(w_1, w_2)[i])} \right) - \sum_{i=1}^{||LCS(w_1, w_2)||} \log \left( \frac{1}{\text{freq}(LCS(w_1, w_2)[i])} \right)
\]

Note:

- \(w_1, w_2\) are two words whose string similarity is to be measured
- \(LCS(w_1, w_2)\) represents the Longest Common Shared Sequence of symbols between the two words

As with other functions, the frequency measure used for each character will be taken from the current corpus. This means that the score will be different for a given pair of words (e.g., pressed vs. pressure) depending on the frequency of the individual characters in the loaded corpus.

14.3 Calculating string similarity in the GUI

To start the analysis, click on “Analysis” / “Calculate string similarity...” in the main menu, and then follow these steps:

1. **String similarity algorithm**: The first step is to choose which of the three methods described above is to be used to calculate string similarity. The options are phonological edit distance, standard (Levenshtein) edit distance, and the algorithm described above and in [Khorsi2012].

2. **Comparison type**: Next, choose what kind of comparison is to be done. One can either take a single word and get its string similarity score to every other word in the corpus (useful, for example, when trying to figure out which words are most / least similar to a given word, as one might for stimuli creation), or can compare individual pairs of words (useful if a limited set of pre-determined words is of interest). For each of these, you can use words that already exist in the corpus or calculate the similarity for words (or non-words) that are not in the corpus. Note that these words will NOT be added to the corpus itself; if you want to globally add the word (and therefore have its own properties affect calculations), please use the instructions in Adding a word.

   a. **One word in the corpus**: To compare the similarity of one word that already exists in the corpus to every other word in the corpus, simply select “Compare one word to entire corpus” and enter the single word into the dialogue box, using its standard orthographic representation. Note that you can choose later which tier string similarity will be calculated on (spelling, transcription, etc.); this simply identifies the word for PCT.
(b) **One word not in the corpus:** Click on “Calculate for a word/nonword not in the corpus” and then select “Create word/nonword” to enter the new word.

i. **Spelling:** Enter the spelling for your new word / nonword using the regular input keyboard on your computer.

ii. **Transcription:** To add in the phonetic transcription of the new word, it is best to use the provided inventory. While it is possible to type directly into the transcription box, using the provided inventory will ensure that all characters are understood by PCT to correspond to existing characters in the corpus (with their concomitant featural interpretation). Click on “Show inventory.” (See also *Edit inventory categories* for more on how to set up the inventory window.) Clicking on the individual segments will add them to the transcription. Note that you do NOT need to include word boundaries at the beginning and end of the word, even when the boundary symbol is included as a member of the inventory; these will be assumed automatically by PCT.

iii. **Frequency and other columns:** These can be left at the default. Note that entering values will NOT affect the calculation; there is no particular need to enter anything here (it is an artifact of using the same dialogue box here as in the “Add Word” function described in *Adding a word*).

iv. **Create word:** To finish and return to the “String similarity” dialogue box, click on “Create word.”

(c) **Single word pair (in or not in) the corpus:** If the similarity of an individual word pair is to be calculated, one can enter the pair directly into the dialogue box. For each word that is in the corpus, simply enter its standard orthographic form. For each word that is not in the corpus, you can add it by selecting “Create word/nonword” and following the steps described immediately above in (2b).

(d) **List of pairs of words (in the corpus):** If there is a long list of pairs of words, one can simply create a tab-delimited plain .txt file with one word pair per line. In this case, click on “Choose file” and select the .txt file in the resulting system dialogue box. Note that this option is currently available only for words that already exist in the corpus, and that these pairs should be listed using their standard orthographic representations.

2. **Tier:** The tier from which string similarity is to be calculated can be selected. Generally, one is likely to care most about either spelling or transcription, but other tiers (e.g., a vowel tier) can also be selected; in this case, all information removed from the tier is ignored. Words should always be entered orthographically (e.g., when telling PCT what word pairs to compare). If similarity is to be calculated on the basis of spelling, words that are entered are broken into their letter components. If similarity is to be calculated on the basis of transcription, the transcriptions are looked up in the corpus, or taken from the created nonword (see step # 1b above).

3. **Pronunciation variants:** If the corpus contains multiple pronunciation variants for lexical items, select what strategy should be used. For details, see *Pronunciation Variants*. Note that here, the only choices currently available are canonical or most-frequent forms.

4. **Frequency type:** If Khorsi similarity is to be calculated, the frequencies of the symbols is relevant, and so will be looked up in the currently loaded corpus. Either type frequency or token frequency can be used for the calculation. This option will not be available for either edit distance algorithm, because frequency isn’t taken into account in either one.

5. **Minimum / Maximum similarity:** If one is calculating the similarity of one word to all others in the corpus, an arbitrary minimum and maximum can be set to filter out words that are particularly close or distant. For example, one could require that only words with an edit distance of both at least and at most 1 are returned, to get the members of the standard neighbourhood of a particular lexical item. (Recall that the Khorsi calculation is a measure of similarity, while edit distance and phonological edit distance are measures of difference. Thus, a minimum similarity value is analogous to a maximum distance value. PCT will automatically interpret “minimum” and “maximum” relative to the string-similarity algorithm chosen.

Here’s an example for calculating the Khorsi similarity of the pair *mata* (which occurs in the corpus) and *mitoo* [mitu] (which does not), in the sample corpus, using token frequencies and comparing transcriptions:
6. **Results:** Once all options have been selected, click “Calculate string similarity.” If this is not the first calculation, and you want to add the results to a pre-existing results table, select the choice that says “add to current results table.” Otherwise, select “start new results table.” A dialogue box will open, showing a table of the results, including word 1, word 2, the result (i.e., the similarity score for Khorsi or distance score for either of the edit algorithms), what choice was made regarding pronunciation variants, whether type or token frequency was used (if the Khorsi method is selected; otherwise, N/A), and which algorithm was used. Note that the entries in the table will be written in spelling regardless of whether spelling or transcriptions were used. This file can be saved to a desired location by selecting “Save to file” at the bottom of the table.

Here’s an example result file for the above selection:

![Example result file](image)

To return to the function dialogue box with your most recently used selections, click on “Reopen function dialog.” Otherwise, the results table can be closed and you will be returned to your corpus view.

### 14.4 Classes and functions

For further details about the relevant classes and functions in PCT’s source code, please refer to *Symbol similarity.*
15.1 About the functions

Some measures of String similarity are used to calculate neighbourhood density (e.g. [Greenberg1964]; [Luce1998]; [Yao2011]), which has been shown to affect phonological processing. A phonological “neighbor” of some word X is a word that is similar in some close way to X. For example, it might differ by maximally one phone (through deletion, addition, or substitution) from X. X’s neighborhood density, then, is the number of words that fit the criterion for being a neighbour.

15.2 Method of calculation

A word’s neighborhood density is equal to the number of other words in the corpus similar to that word (or, if using token frequencies, the sum of those words’ counts). The threshold that defines whether two words are considered similar to each other can be calculated using any of the three distance metrics described in Method of calculation: Levenshtein edit distance, phonological edit distance, or Khorsi (2012) similarity. As implemented in PCT, for a query word, each other word in the corpus is checked for its similarity to the query word and then added to a list of neighbors if sufficiently similar.

For further detail about the available distance/similarity metrics, refer to Method of calculation.

15.3 Calculating neighbourhood density in the GUI

To start the analysis, click on “Analysis” / “Calculate neighbourhood density...” in the main menu, and then follow these steps:

1. String similarity algorithm: The first step is to choose which of the three methods of String similarity is to be used to calculate neighbourhood density. Note that the standard way of calculating density is using regular (Levenshttein) edit distance. We include the other two algorithms here as options primarily for the purpose of allowing users to explore whether they might be useful measures; we make no claims that either phonological edit distance or the Khorsi algorithm might be better than edit distance for any reason.

   (a) Minimal pair counts / Substitution neighbours: It is also possible to calculate neighbourhood density by using a variation of edit distance that allows for “substitutions only” (not deletions or insertions). This is particularly useful if, for example, you wish to know the number of or identity of all minimal pairs for a given word in the corpus, as minimal pairs are generally assumed to be substitution neighbours with an edit distance of 1. (Note that the substitution neighbours algorithm automatically assumes a threshold of 1; multiple substitutions are not allowed.)
2. **Query type**: Neighbourhood density can be calculated for one of four types of inputs:

   (a) **One word in the corpus**: The neighbourhood density of a single word can be calculated by entering that word’s orthographic representation in the query box.

   (b) **One word not in the corpus**: (Note that this will NOT add the word itself to the corpus, and will not affect any subsequent calculations. To globally add a word to the corpus itself, please see the instructions in *Adding a word*.) Select “Calculate for a word/nonword in the corpus,” then choose “Create word/nonword” to enter the new word and do the following:

      i. **Spelling**: Enter the spelling for your new word / nonword using the regular input keyboard on your computer.

      ii. **Transcription**: To add in the phonetic transcription of the new word, it is best to use the provided inventory. While it is possible to type directly in to the transcription box, using the provided inventory will ensure that all characters are understood by PCT to correspond to existing characters in the corpus (with their concomitant featural interpretation). Click on “Show inventory.” (See also *Edit inventory categories* for more on how to set up the inventory window.) Clicking on the individual segments will add them to the transcription. Note that you do NOT need to include word boundaries at the beginning and end of the word, even when the boundary symbol is included as a member of the inventory; these will be assumed automatically by PCT.

      iii. **Frequency and other columns**: These can be left at the default. Note that entering values will NOT affect the calculation; there is no particular need to enter anything here (it is an artifact of using the same dialogue box here as in the “Add Word” function described in *Adding a word*).

      iv. **Create word**: To finish and return to the “String similarity” dialogue box, click on “Create word.”

   (c) **List of words**: If there is a specific list of words for which density is to be calculated (e.g., the stimuli list for an experiment), that list can be saved as a .txt file with one word per line and uploaded into PCT for analysis. Note that in this case, if the words *are* in the corpus, either transcription- or spelling-based neighbourhood density can be calculated; either way, the words on the list should be written in standard orthography (their transcriptions will be looked up in the corpus if needed). If the words *are not* in the corpus, then only spelling-based neighbourhood density can currently be calculated; again, the words should be written orthographically.

   (d) **Whole corpus**: Alternatively, the neighbourhood density for every word in the corpus can be calculated. This is useful, for example, if one wishes to find words that match a particular neighbourhood density. The density of each word will be added to the corpus itself, as a separate column; in the “query” box, simply enter the name of that column (the default is “Neighborhood Density”).

3. **Tier**: Neighbourhood density can be calculated from most of the available tiers in a corpus (e.g., spelling, transcription, or tiers that represent subsets of entries, such as a vowel or consonant tier). (If neighbourhood density is being calculated with phonological edit distance as the similarity metric, spelling cannot be used.) Standard neighbourhood density is calculated using edit distance on transcriptions.

4. **Pronunciation variants**: If the corpus contains multiple pronunciation variants for lexical items, select what strategy should be used. For details, see *Pronunciation Variants*. Note that here, the only choices currently available are canonical or most-frequent forms.

5. **Type vs. token frequency**: If the Khorsi algorithm is selected as the string similarity metric, similarity can be calculated using either type or token frequency, as described in *Khorsi (2012) Similarity Metric*.

6. **Distance / Similarity Threshold**: A specific threshold must be set to determine what counts as a “neighbour.” If either of the edit distance metrics is selected, this should be the maximal distance that is allowed – in standard calculations of neighbourhood density, this would be 1, signifying a maximum 1-phone change from the starting word. If the Khorsi algorithm is selected, this should be the minimum similarity score that is required. Because this is not the standard way of calculating neighbourhood density, we have no recommendations for what value(s) might be good defaults here; instead, we recommend experimenting with the string similarity al-
algorithm to determine what kinds of values are common for words that seem to count as neighbours, and working backward from that.

7. **Output file**: If this option is left blank, PCT will simply return the actual neighbourhood density for each word that is calculated (i.e., the number of neighbours of each word). If a file is chosen, then the number will still be returned, but additionally, a file will be created that lists all of the actual neighbours for each word.

8. **Results**: Once all options have been selected, click “Calculate neighborhood density.” If this is not the first calculation, and you want to add the results to a pre-existing results table, select the choice that says “add to current results table.” Otherwise, select “start new results table.” A dialogue box will open, showing a table of the results, including the word, its neighbourhood density, the string type from which neighbourhood density was calculated, what choice was made regarding pronunciation variants, whether type or token frequency was used (if applicable), the string similarity algorithm that was used, and the threshold value. If the neighbourhood density for all words in the corpus is being calculated, simply click on the “start new results table” option, and you will be returned to your corpus, where a new column has been added automatically.

9. **Saving results**: The results tables can each be saved to tab-delimited .txt files by selecting “Save to file” at the bottom of the window. Any output files containing actual lists of neighbours are already saved as .txt files in the location specified (see step 7). If all neighbourhood densities are calculated for a corpus, the corpus itself can be saved by going to “File” / “Export corpus as text file,” from where it can be reloaded into PCT for use in future sessions with the neighbourhood densities included.

Here’s an example of neighbourhood density being calculated on transcriptions for the entire example corpus, using edit distance with a threshold of 1:

![Neighborhood Density](image)

The corpus with all words’ densities added:
An example of calculating all the transcription neighbours for a given word in the IPHOD corpus, and saving the resulting list of neighbours to an output file:

The on-screen results table, which can be saved to a file itself:
And the saved output file listing all 45 of the neighbours of *cat* in the IPHOD corpus:
An example .txt file containing one word per line, that can be uploaded into PCT so that the neighbourhood density of each word is calculated:
The resulting table of neighbourhood densities for each word on the list (in the IPHOD corpus, with standard edit distance and a threshold of 1):

To return to the function dialogue box with your most recently used selections after any results table has been created, click on “Reopen function dialog.” Otherwise, the results table can be closed and you will be returned to your corpus view.

### 15.4 Implementing the neighbourhood density function on the command line

In order to perform this analysis on the command line, you must enter a command in the following format into your Terminal:

```
pct_neighdens CORPUSFILE ARG2
```

...where CORPUSFILE is the name of your *.corpus file and ARG2 is either the word whose neighborhood density you wish to calculate or the name of your word list file (if calculating the neighborhood density of each word). The
word list file must contain one word (specified using either spelling or transcription) on each line. You may also use command line options to change various parameters of your neighborhood density calculations. Descriptions of these arguments can be viewed by running `pct_neighdens -h` or `pct_neighdens -help`. The help text from this command is copied below, augmented with specifications of default values:

Positional arguments:

- **corpus_file_name**: Name of corpus file
- **query**: Name of word to query, or name of file including a list of words

Optional arguments:

- **-h**
  - **--help**: Show this help message and exit
- **-c CONTEXT_TYPE**
  - **--context_type CONTEXT_TYPE**: How to deal with variable pronunciations. Options are ‘Canonical’, ‘MostFrequent’, ‘SeparatedTokens’, or ‘Weighted’. See documentation for details.
- **-a ALGORITHM**
  - **--algorithm ALGORITHM**: The algorithm used to determine distance
- **-d MAX_DISTANCE**
  - **--max_distance MAX_DISTANCE**: Maximum edit distance from the queried word to consider a word a neighbor.
- **-s SEQUENCE_TYPE**
  - **--sequence_type SEQUENCE_TYPE**: The name of the tier on which to calculate distance
- **-w COUNT_WHAT**
  - **--count_what COUNT_WHAT**: If ‘type’, count neighbors in terms of their type frequency. If ‘token’, count neighbors in terms of their token frequency.
- **-m**
  - **--find_mutation_minpairs**: This flag causes the script not to calculate neighborhood density, but rather to find minimal pairs—see documentation.
- **-o OUTFILE**
  - **--outfile OUTFILE**: Name of output file.

**EXAMPLE 1**: If your corpus file is example.corpus (no pronunciation variants) and you want to calculate the neighborhood density of the word ‘nata’ using defaults for all optional arguments, you would run the following command in your terminal window:

```
pct_neighdens example.corpus nata
```

**EXAMPLE 2**: Suppose you want to calculate the neighborhood distance of a list of words located in the file mywords.txt. Your corpus file is again example.corpus. You want to use the phonological edit distance metric, and you wish to count as a neighbor any word with a distance less than 0.75 from the query word. In addition, you want the script to produce an output file called output.txt. You would need to run the following command:
pct_neighdens example.corpus mywords.txt -a phonological_edit_distance -d 0.75 -o output.txt

EXAMPLE 3: You wish to find a list of the minimal pairs of the word ‘nata’. You would need to run the following command:

pct_neighdens example.corpus nata -m

### 15.5 Classes and functions

For further details about the relevant classes and functions in PCT’s source code, please refer to *Neighborhood density*.
16.1 About the function

The occurrence of alternations can be used in assessing whether two phonemes in a language are contrastive or allophonic, with alternations promoting the analysis of allophony (e.g., [Silverman2006], [Johnson2010], [Lu2012]), though it’s clear that not all alternating sounds are allophonic (e.g., the [k]~[s] alternation in electric~electricity).

In general, two phonemes are considered to alternate if they occur in corresponding positions in two related words. For example, [s]//[f] would be considered an alternation in the words [d@p@r@z] and [d@p@r@f@n] as they occur in corresponding positions and the words are morphologically related. [Johnson2010] make the point that alternations may be more or less frequent in a language, and imply that this frequency may affect the influence of the alternations on phonological relations. As far as we know, however, there is no literature that directly tests this claim or establishes how frequency of alternation could actually be quantified (though see discussion in [Hall2014b]).

16.2 Method of calculation

In PCT, frequency of alternation [1] is the ratio of the number of words that have an alternation of two phonemes to the total number of words that contain either phoneme, as in:

\[
\text{Frequency of alternation} = \frac{\text{Words with an alternation of } s_1 \text{ and } s_2}{\text{Words with } s_1 + \text{words with } s_2}
\]

To determine whether two words have an alternation of the targeted phonemes, one word must contain phoneme 1, the other must contain phoneme 2, and some threshold of “relatedness” must be met. In an ideal world, this would be determined by a combination of orthographic, phonological, and semantic similarity; see discussion in [Hall2014b]. Within PCT, however, a much more basic relatedness criterion is used: string similarity. This is indeed what [Khorsi2012] proposes as a measure of morphological relatedness, and though we caution that this is not in particularly close alignment with the standard linguistic interpretation of morphological relatedness, it is a useful stand-in for establishing an objective criterion. If both conditions are met, the two words are considered to have an alternation and are added to the pool of “words with an alternation of s1 and s2.”

It is also possible to require a third condition, namely, that the location of phoneme 1 and phoneme 2 be roughly phonologically aligned across the two words (e.g., preceded by the same material). Requiring phonological alignment will make PCT more conservative in terms of what it considers to “count” as alternations. However, the phonological alignment algorithm is based on [Allen2014] and currently only works with English-type morphology, i.e., a heavy reliance on prefixes and suffixes rather than any other kinds of morphological alternations. Thus, it should not be used with non-affixing languages.

Again, we emphasize that we do not believe this to currently be a particularly accurate reflection of morphological relatedness, so the resulting calculation of frequency of alternation should be treated with extreme caution. We include
it primarily because it is a straightforward function of string similarity that has been claimed to be relevant, not because the current instantiation is thought to be particularly valid.

16.3 Calculating frequency of alternation in the GUI

To start the analysis, click on “Analysis” / “Calculate frequency of alternation...” in the main menu, and then follow these steps:

1. **Segments**: First, select which pairs of sounds you want the functional load to be calculated for. Do this by clicking on “Add pair of sounds”; the “Select segment pair” dialogue box will open. The order that the sounds are selected in is irrelevant; picking [i] first and [u] second will yield the same results as picking [u] first and [i] second. Once sounds have been selected, click “Add.” Pairs will appear in the “Functional load” dialogue box. See more about interacting with the sound selection box (including, e.g., the use of features in selecting sounds and the options for selecting multiple pairs) in Sound Selection.

2. **String similarity algorithm**: Next, choose which distance / similarity metric to use. Refer to Method of calculation for more details.

3. **Threshold values**: If the Khorsi algorithm is selected, enter the minimum similarity value required for two words to count as being related. Currently the default is -15; this is an arbitrary (and relatively low / non-conservative) value. We recommend reading [Khorsi2012] and examining the range of values obtained using the string similarity algorithm before selecting an actual value here. Alternatively, if one of the edit distance algorithms is selected, you should instead enter a maximum distance value that is allowed for two words to count as being related. Again, there is a default (6) that is relatively high and non-conservative; an understanding of edit distances is crucial for applying this threshold in a meaningful way.

4. **Tier**: The tier from which string similarity is to be calculated can be selected. Generally, one is likely to care most about full transcriptions, but other tiers (e.g., a vowel tier) can also be selected; in this case, all information removed from the tier is ignored.

5. **Pronunciation variants**: If the corpus contains multiple pronunciation variants for lexical items, select what strategy should be used. For details, see Pronunciation Variants. Note that here, the only choices currently available are canonical or most-frequent forms.

6. **Frequency Type**: Next, select which frequency type to use for your similarity metric, either type or token frequency. This parameter is only available if using the Khorsi similarity metric, which relies on counting the frequency of occurrence of the sounds in the currently selected corpus; neither edit distance metric involves frequency.

7. **Minimal pairs**: Then, select whether you wish to include alternations that occur in minimal pairs. If, for example, the goal is to populate a list containing all instances where two segments potentially alternate, select “include minimal pairs.” Alternatively, if one wishes to discard known alternations that are contrastive, select “ignore minimal pairs.” (E.g., “bat” and “pat” look like a potential “alternation” of [b] and [p] to PCT, because they are extremely similar except for the sounds in question, which are also phonologically aligned.)

8. **Phonological alignment**: Choose whether you want the phones to be phonologically aligned or not, as per the above explanation.

9. **Corpus size**: Calculating the full set of possible alternations for a pair of sounds may be extremely time-consuming, as all words in the corpus must be compared pairwise. To avoid this problem, a subset of the corpus can be selected (in which case, we recommend running the calculation several times so as to achieve different random subsets for comparison). To do so, enter either (1) the number of words you’d like PCT to extract from the corpus as a subset (e.g., 5000) or (2) a decimal, which will result in that percentage of the corpus being used as a subset (e.g., 0.05 for 5% of the corpus).

10. **Output alternations**: You can choose whether you want PCT to output a list of all the words it considers to be “alternations.” This is useful for determining how accurate the calculation is. If you do want the list to be
created, enter a file path or select it using the system dialogue box that opens when you click on “Select file location.” If you do not want such a list, leave this option blank.

An example of selecting the parameters for frequency of alternation, using the sample corpus:

![Frequency Of Alternation Dialogue Box](image)

11. **Results**: Once all options have been selected, click “Calculate frequency of alternation.” If this is not the first calculation, and you want to add the results to a pre-existing results table, select the choice that says “add to current results table.” Otherwise, select “start new results table.” A dialogue box will open, showing a table of the results, including sound 1, sound 2, the total number of words with either sound, and total number of words with an alternation, the frequency of alternation and information about the specified similarity / distance metric and selected threshold values, and the selected option with respect to pronunciation variants. To save these results to a .txt file, click on “Save to file” at the bottom of the table.

An example of the results table:

![Results Table](image)

To return to the function dialogue box with your most recently used selections, click on “Reopen function dialog.” Otherwise, the results table can be closed and you will be returned to your corpus view.

**16.4 Classes and functions**

For further details about the relevant classes and functions in PCT’s source code, please refer to *Frequency of alternation*. 

---

**16.4. Classes and functions**

---

**107**
17.1 About the function

Mutual information is a measure of how much dependency there is between two random variables, $X$ and $Y$. That is, there is a certain amount of information gained by learning that $X$ is present and also a certain amount of information gained by learning that $Y$ is present. But knowing that $X$ is present might also tell you something about the likelihood of $Y$ being present, and vice versa. If $X$ and $Y$ always co-occur, then knowing that one is present already tells you that the other must also be present. On the other hand, if $X$ and $Y$ are entirely independent, then knowing that one is present tells you nothing about the likelihood that the other is present.

In phonology, there are two primary ways in which one could interpret $X$ and $Y$ as random variables. In one version, $X$ and $Y$ are equivalent random variables, each varying over “possible speech sounds in some unit” (where the unit could be any level of representation, e.g. a word or even a non-meaningful unit such as a bigram). In this case, one is measuring how much the presence of $X$ anywhere in the defined unit affects the presence of $Y$ in that same unit, regardless of the order in which $X$ and $Y$ occur, such that the mutual information of $(X; Y)$ is the same as the mutual information of $(Y; X)$, and furthermore, the pointwise mutual information of any individual value of each variable ($X = a; Y = b$) is the same as the pointwise mutual information of $(X = b; Y = a)$. Although his is perhaps the most intuitive version of mutual information, given that it does give a symmetric measure for “how much information does the presence of $a$ provide about the presence of $b$?” we are not currently aware of any work that has attempted to use this interpretation of MI for phonological purposes.

The other interpretation of MI assumes that $X$ and $Y$ are different random variables, with $X$ being “possible speech sounds occurring as the first member of a bigram” and $Y$ being “possible speech sounds occurring as the second member of a bigram.” This gives a directional interpretation to mutual information, such that, while the mutual information of $(X; Y)$ is the same as the mutual information of $(Y; X)$, the pointwise mutual information of $(X = a; Y = b)$ is NOT the same as the pointwise mutual information of $(X = b; Y = a)$, because the possible values for $X$ and $Y$ are different. (It is still, trivially, the case that the pointwise mutual information of $(X = a; Y = b)$ and $(Y = b; X = a)$ are equal.)

This latter version of mutual information has primarily been used as a measure of co-occurrence restrictions (harmony, phonotactics, etc.). For example, [Goldsmith2012] use pointwise mutual information as a way of examining Finnish vowel harmony; see also discussion in [Goldsmith2002]. Mutual information has also been used instead of transitional probability as a way of finding boundaries between words in running speech, with the idea that bigrams that cross word boundaries will have, on average, lower values of mutual information than bigrams that are within words (see [Brent1999], [Rytting2004]). Note, however, that in order for this latter use of mutual information to be useful, one must be using a corpus based on running text rather than a corpus that is simply a list of individual words and their token frequencies.

\[1\] The algorithm in PCT calculates what is sometimes referred to as the “pointwise” mutual information of a pair of units $X$ and $Y$, in contrast to “mutual information,” which would be the expected average value of the pointwise mutual information of all possible values of $X$ and $Y$. We simplify to use “mutual information” throughout.
17.2 Method of calculation

Both of the interpretations of mutual information described above are implemented in PCT. We refer to the first one, in which X and Y are interpreted as equal random variables, varying over “possible speech sounds in a unit,” as word-internal co-occurrence pointwise mutual information (pMI), because we specifically use the word as the unit in which to measure pMI. We refer to the second one, in which X and Y are different random variables, over either the first or second members of bigrams, as ordered pair pMI.

The general formula for pointwise mutual information is given below; it is the binary logarithm of the joint probability of X = a and Y = b, divided by the product of the individual probabilities that X = a and Y = b.

\[ pMI = \log_2 \left( \frac{p(\text{X} = a \& \text{Y} = b)}{p(\text{X} = a) \cdot p(\text{Y} = b)} \right) \]

**Word-internal co-occurrence pMI**: In this version, the joint probability that X = a and Y = b is equal to the probability that some unit (here, a word) contains both a and b (in any order). Therefore, the pointwise mutual information of the sounds a and b is equal to the binary logarithm of the probability of some word containing both a and b, divided by the product of the individual probabilities of a word containing a and a word containing b.

Pointwise mutual information for individual segments:

\[ pMI_{\text{word-internal}} = \log_2 \left( \frac{p(a \in W \& b \in W)}{p(a \in W) \cdot p(b \in W)} \right) \]

Ordered pair pMI: In this version, the joint probability that X = a and Y = b is equal to the probability of occurrence of the sequence ab. Therefore, the pointwise mutual information of a bigram (e.g., ab) is equal to the binary logarithm of the probability of the bigram divided by the product of the individual segment probabilities, as shown in the formula below.

Pointwise mutual information for bigrams:

\[ pMI_{\text{ordered-pair}} = \log_2 \left( \frac{p(ab)}{p(a) \cdot p(b)} \right) \]

For example, given the bigram [a, b], its pointwise mutual information is the binary logarithm of the probability of the sequence [ab] in the corpus divided by a quantity equal to the probability of [a] times the probability of [b]. Bigram probabilities are calculated by dividing counts by the total number of bigrams, and unigram probabilities are calculated equivalently.

Note that pMI can also be expressed in terms of the information content of each of the members of the bigram. Information is measured as the negative log of the probability of a unit \( I(a) = -\log_2 p(a) \), so the pMI of a bigram ab is also equal to \( I(a) + I(b) - I(ab) \).

Note that in PCT, calculations are not rounded until the final stage, whereas in [Goldsmith2012], rounding was done at some intermediate stages as well, which may result in slightly different final pMI values being calculated.

17.3 Calculating mutual information in the GUI

To start the analysis, click on “Analysis” / “Calculate mutual information...” in the main menu, and then follow these steps:

1. **Bigram**: Click on the “Add bigram” button in the “Mutual Information” dialogue box. A new window will open with an inventory of all the segments that occur in your corpus. Select the bigram by clicking on one segment from the “left-hand side” and one segment from the “right-hand side.” Note that the order of the sounds matters in this function! To add more than one bigram, click “Add and create another” to be automatically returned to the selection window. Once the last bigram has been selected, simply click “Add” to return to the Mutual Information dialogue box. All the selected bigrams will appear in a list. To remove one, click on it and select “Remove selected bigram.”
2. **Tier:** Mutual information can be calculated on any available tier. The default is transcription. If a vowel tier has been created, for example, one could calculate the mutual information between vowels on that tier, ignoring intervening consonants, to examine harmony effects.

3. **Pronunciation variants:** If the corpus contains multiple pronunciation variants for lexical items, select what strategy should be used. For details, see *Pronunciation Variants*.

4. **Type vs. Token Frequency:** Next, pick whether you want the calculation to be done on types or tokens, assuming that token frequencies are available in your corpus. If they are not, this option will not be available. (Note: if you think your corpus does include token frequencies, but this option seems to be unavailable, see *Required format of corpus* on the required format for a corpus.)

5. **Domain:** Choosing “set domain to word” will change the calculation so that the calculation is for word-internal co-occurrence pMI. In this case, the order and adjacency of the bigram does not matter; it is simply treated as a pair of segments that could occur anywhere in a word.

6. **Word boundary count:** A standard word object in PCT contains word boundaries on both sides of it (e.g., [#kat#] ‘cat’). If words were concatenated in real running speech, however, one would expect to see only one word boundary between each pair of words (e.g., [#mai#kat#] ‘my cat’ instead of [#mai##kat#]). To reproduce this effect and assume that word boundaries occur only once between words (as is assumed in *Goldsmith2012*), choose “halve word boundary count.” Note that this technically divides the number of boundaries in half and then adds one, to compensate for the extra “final” boundary at the end of an utterance. (It will make a difference only for calculations that include a boundary as one member of the pair.)

7. **Results:** Once all options have been selected, click “Calculate mutual information.” If this is not the first calculation, and you want to add the results to a pre-existing results table, select the choice that says “add to current results table.” Otherwise, select “start new results table.” A dialogue box will open, showing a table of the results, including sound 1, sound 2, the tier used, and the mutual information value. To save these results to a .txt file, click on “Save to file” at the bottom of the table.

The following image shows the inventory window used for selecting bigrams in the sample corpus:

The selected bigrams appear in the list in the “Mutual Information” dialogue box:
The resulting mutual information results table:

To return to the function dialogue box with your most recently used selections, click on “Reopen function dialog.” Otherwise, the results table can be closed and you will be returned to your corpus view.

17.4 Implementing the mutual information function on the command line

In order to perform this analysis on the command line, you must enter a command in the following format into your Terminal:

```
pct_mutualinfo CORPUSFILE [additional arguments]
```

...where CORPUSFILE is the name of your *.corpus file. If not calculating the mutual informations of all bigrams (using -l), the query bigram must be specified using -q, as `-q QUERY`. The bigram QUERY must be in the format `s1,s2` where `s1` and `s2` are the first and second segments in the bigram. You may also use command line options to change the sequency type to use for your calculations, or to specify an output file name. Descriptions of these
arguments can be viewed by running `pct_mutualinfo -h` or `pct_mutualinfo --help`. The help text from this command is copied below, augmented with specifications of default values:

Positional arguments:

**corpus_file_name**
Name of corpus file

Mandatory argument group (call must have one of these two):

- `-q QUERY`
  --query QUERY
  Bigram or segment pair, as str separated by comma

- `-l`
  --all_pairwise_mis
  Flag: calculate MI for all orders of all pairs of segments

Optional arguments:

- `-h`
  --help
  Show help message and exit

- `-c CONTEXT_TYPE`
  --context_type CONTEXT_TYPE
  How to deal with variable pronunciations. Options are ‘Canonical’, ‘MostFrequent’, ‘SeparatedTokens’, or ‘Weighted’. See documentation for details.

- `-s SEQUENCE_TYPE`
  --sequence_type SEQUENCE_TYPE
  The attribute of Words to calculate MI over. Normally, this will be the transcription, but it can also be the spelling or a user-specified tier.

- `-o OUTFILE`
  --outfile OUTFILE
  Name of output file

EXAMPLE 1: If your corpus file is example.corpus (no pronunciation variants) and you want to calculate the mutual information of the bigram ‘si’ using defaults for all optional arguments, you would run the following command in your terminal window:

```
pct_mutualinfo example.corpus -q s,i
```

EXAMPLE 2: Suppose you want to calculate the mutual information of the bigram ‘si’ on the spelling tier. In addition, you want the script to produce an output file called output.txt. You would need to run the following command:

```
pct_mutualinfo example.corpus -q s,i -s spelling -o output.txt
```

EXAMPLE 3: Suppose you want to calculate the mutual information of all bigram types in the corpus. In addition, you want the script to produce an output file called output.txt. You would need to run the following command:

```
pct_mutualinfo example.corpus -l -o output.txt
```

17.5 Classes and functions

For further details about the relevant classes and functions in PCT’s source code, please refer to *Mutual information.*
18.1 About the function

Acoustic similarity analyses quantify the degree to which waveforms of linguistic objects (such as sounds or words) are similar to each other. The acoustic similarity measures provided here have primarily been used in the study of phonetic convergence between interacting speakers; convergence is measured as a function of increasing similarity. These measures are also commonly used in automatic speech and voice recognition systems, where incoming speech is compared to stored representations. Phonologically, acoustic similarity also has a number of applications. For example, it has been claimed that sounds that are acoustically distant from each other cannot be allophonically related, even if they are in complementary distribution (e.g. [Pike1947]; [Janda1999]).

Acoustic similarity algorithms work on an aggregate scale, quantifying, on average, how similar one group of waveforms is to another. Representations have traditionally been in terms of mel-frequency cepstrum coefficients (MFCCs; [Delvaux2007]; [Mielke2012]), which is used widely for automatic speech recognition, but one recent introduction is multiple band amplitude envelopes [Lewandowski2012]. Both MFCCs and amplitude envelopes will be described in more detail in the following sections, and both are available as part of PCT.

The second dimension to consider is the algorithm used to match representations. The most common one is dynamic time warping (DTW), which uses dynamic programming to calculate the optimal path through a distance matrix [Sakoe1971], and gives the best alignment of two time series. Because one frame in one series can align to multiple frames in another series without a significant cost, DTW provides a distance independent of time. The other algorithm that is used is cross-correlation (see discussion in [Lewandowski2012], which aligns two time series at variable lags. Taking the max value of the alignment gives a similarity value for the two time series, with higher values corresponding to higher similarity.

18.2 Method of calculation

18.2.1 Preprocessing

Prior to conversion to MFCCs or amplitude envelopes, the waveform is pre-emphasized to give a flatter spectrum and correct for the higher drop off in amplitude of higher frequencies due to distance from the mouth.

18.2.2 MFCCs

The calculation of MFCCs in PCT’s function follows the Rastamat [Ellis2005]’s implementation of HTK-style MFCCs [HTK] in [Matlab]. Generating MFCCs involves windowing the acoustic waveform and transforming the windowed signal to the linear frequency domain through a Fourier transform. Following that, a filterbank of triangular filters is constructed in the mel domain, which gives greater resolution to lower frequencies than higher frequencies.
Once the filterbank is applied to the spectrum from the Fourier transform, the spectrum is represented as the log of the power in each of the mel filters. Using this mel spectrum, the mel frequency cepstrum is computed by performing a discrete cosine transform. This transform returns orthogonal coefficients describing the shape of the spectrum, with the first coefficient as the average value, the second as the slope of the spectrum, the third as the curvature, and so on, with each coefficient representing higher order deviations. The first coefficient is discarded, and the next X coefficients are taken, where X is the number of coefficients specified when calling the function. The number of coefficients must be one less than the number of filters, as the number of coefficients returned by the discrete cosine transform is equal to the number of filters in the mel filterbank.

Please note that the MFCCs calculated in PCT follow the HTK standard, and are not the same as those produced by Praat [PRAAT].

18.2.3 Amplitude envelopes

The calculation of amplitude envelopes follows the Matlab implementation found in [Lewandowski2012]. First, the signal is filtered into X number of logarithmically spaced bands, where X is specified in the function call, using 4th order Butterworth filters. For each band, the amplitude envelope is calculated by converting the signal to its analytic signal through a Hilbert transform. Each envelope is downsampled to 120 Hz.

18.2.4 Dynamic time warping (DTW)

PCT implements a standard DTW algorithm [SakoeChiba, 1971] and gives similar results as the dtw package [Giorgino2009] in [R]. Given two representations, a 2D matrix is constructed where the dimensions are equal to the number of frames in each representation. The initial values for each cell of the matrix is the Euclidean distance between the two feature vectors of those frames. The cells are updated so that they equal the local distance plus the minimum distance of the possible previous cells. At the end, the final cell contains the summed distance of the best path through the matrix, and this is the minimum distance between two representations.

18.2.5 Cross-correlation

Cross-correlation seeks to align two time series based on corresponding peaks and valleys. From each representation a time series is extracted for each frame’s feature and this time series is cross-correlated with the respective time series in the other representation. For instance, the time series for an amplitude envelope’s representation corresponds to each frequency band, and each frequency band of the first representation is cross-correlated with each respective frequency band of the second representation. The time series are normalized so that they sum to 1, and so matching signals receive a cross-correlation value of 1 and completely opposite signals receive a cross-correlation value of 0. The overall distance between two representations is the inverse of the average cross-correlation values for each band.

18.2.6 Similarity across directories

The algorithm for assessing the similarity of two directories (corresponding to segments) averages the similarity of each .wav file in the first directory to each .wav file in the second directory.

18.3 Calculating acoustic similarity in the GUI

To start the analysis, click on the “Calculate acoustic similarity...” in the Analysis menu and provide the following parameters. Note that unlike the other functions, acoustic similarity is not tied directly to any corpus that is loaded into PCT; sound files are accessed directly through directories on your computer.
1. **Comparison type**: There are three kinds of comparisons that can be done in PCT: single-directory, two-directory, or pairwise.

   (a) **Single directory**: If a single directory is selected (using the “Choose directory...” dialogue box), two types of results will be returned: (1) each of the pairwise comparisons and (2) an average of all these comparisons (i.e., a single value).

   (b) **Two directories**: Choose two directories, each corresponding to a set of sounds to be compared. For example, if one were interested in the similarity of [s] and [ʃ] in Hungarian, one directory would contain .wav files of individual [s] tokens, and the other directory would contain .wav files of individual [ʃ] tokens. Every sound file in the first directory will be compared to every sound file in the second directory, and the acoustic similarity measures that are returned will again be (1) all the pairwise comparisons and (2) an average of all these comparisons (i.e., a single value).

   (c) **Pairwise**: One can also use a tab-delimited.txt file that lists all of the pairwise comparisons of individual sound files by listing their full path names. As with a single directory, each pairwise comparison will be returned separately.

2. **Representation**: Select whether the sound files should be represented as MFCCs or amplitude envelopes (described in more detail above).

3. **Distance algorithm**: Select whether comparison of sound files should be done using dynamic time warping or cross-correlation (described in more detail above).

4. **Frequency limits**: Select a minimum frequency and a maximum frequency to use when generating representations. The human voice typically doesn’t go below 80 Hz, so that is the default cut off to avoid low-frequency noise. The maximum frequency has a hard bound of the Nyquist frequency of the sound files, that is, half their sampling rate. The lowest sampling rate that is typically used for speech is 16,000 Hz, so a cutoff near the Nyquist (8,000 Hz) is used as the default. The range of human hearing is 20 Hz to 20 kHz, but most energy in speech tends to fall off after 10 kHz.

5. **Frequency resolution**: Select the number of filters to be used to divide up the frequency range specified above. The default for MFCCs is for 26 filters to be constructed, and for amplitude envelopes, 8 filters.

6. **Number of coefficients (MFCC only)**: Select the number of coefficients to be used in MFCC representations. The default is 12 coefficients, as that is standard in the field. If the number of coefficients is more than the number of filters minus one, the number of coefficients will be set to the number of filters minus one.

7. **Output**: Select whether to return results as similarity (inverse distance) or to use the default, distance (inverse similarity). Dynamic time warping natively returns a distance measure which gets inverted to similarity and cross-correlation natively returns a similarity value which gets inverted to distance.

8. **Multiprocessing**: As the generation and comparison of representations can be time-intensive, using multiprocessing on parts that can be run in parallel can speed the process up overall. In order to make this option available, the python-acoustic-similarity module must be installed; multiprocessing itself can be enabled by going to “Options” / “Preferences” / “Processing” (see also §3.9.1).

Here’s an example of the parameter-selection box:
9. **Calculating and saving results**: The first time an analysis is run, the option to “Calculate acoustic similarity (start new results table)” should be selected. This will output the results to a pop-up window that lists the directories, the representation choice, the matching function, the minimum and maximum frequencies, the number of filters, the number of coefficients, the raw result, and whether the result is similarity (1) or distance (0). Subsequent analyses can either be added to the current table (as long as it hasn’t been closed between analyses) or put into a new table. Once a table has been created, click on “Save to file” at the bottom of the table window in order to open a system dialogue box and choose a directory; the table will be saved as a tab-delimited .txt file.

Here’s an example of the results file:

![Acoustic similarity results](image)

To return to the function dialogue box with your most recently used selections, click on “Reopen function dialog.” Otherwise, the results table can be closed and you will be returned to your corpus view.
18.4 Classes and functions

For further details about the relevant classes and functions in PCT’s source code, please refer to *API Reference*. 
Please cite PCT as the following (all authors after K. C. Hall are listed alphabetically):


If you need to cite a more traditional academic source rather than the software itself, please use:


If you are using the IPHOD corpus as distributed with PCT, please also be sure to cite:


and if you are making use of the SUBTLEX token frequencies as part of the IPHOD corpus, you should cite:


More generally, the algorithms that are implemented in PCT are taken from published sources. As mentioned in the introduction, we highly encourage users of PCT to cite the original sources of the algorithms rather than, for example, saying that “functional load was calculated using PCT” and just citing PCT itself. First, there are multiple parameters within PCT that can be selected for any given calculation, and these should themselves be specified for maximum clarity and replicability. Second, credit for the original creation or application of the algorithms should obviously be given to the proper sources. We have attempted to make this as easy as possible by both giving these sources here in the user’s manual and also embedding them in each function in the “About” option for each.

Furthermore, if you are the author of a function that is currently implemented in PCT and you disagree with the way in which it has been implemented, please contact us to let us know! We have done our best to faithfully replicate published descriptions, but it is obviously possible that we have made errors.

Finally, if you are the author of a function that you would like to see implemented in PCT, please contact us to discuss the possibility.
21.1 Lexicon classes

```python
lexicon.Attribute(name, att_type[, ...])
```
Attributes are for collecting summary information about attributes of Words or WordTokens, with different types of attributes allowing for different behaviour.

```python
lexicon.Corpus(name)
```
Lexicon to store information about Words, such as transcriptions.

```python
lexicon.Inventory([data])
```
Inventories contain information about a Corpus’ segmental inventory.

```python
lexicon.FeatureMatrix(name, feature_entries)
```
An object that stores feature values for segments

```python
lexicon.Segment(symbol)
```
Class for segment symbols

```python
lexicon.Transcription(seg_list)
```
Transcription object, sequence of symbols

```python
lexicon.Word(**kwargs)
```
An object representing a word in a corpus

```python
lexicon.EnvironmentFilter(middle_segments[, ...])
```
Filter to use for searching words to generate Environments that match

```python
lexicon.Environment(middle, position[, lhs, rhs])
```
Specific sequence of segments that was a match for an EnvironmentFilter

### 21.1.1 Attribute

```python
class corputools.corpus.classes.lexicon.Attribute(name, att_type, display_name=None,
                                                 default_value=None)
```
Attributes are for collecting summary information about attributes of Words or WordTokens, with different types of attributes allowing for different behaviour.

**Parameters**

- `name`: str
  Python-safe name for using `getattr` and `setattr` on Words and WordTokens

- `att_type`: str
  Either ‘spelling’, ‘tier’, ‘numeric’ or ‘factor’

- `display_name`: str
  Human-readable name of the Attribute, defaults to None

- `default_value`: object
  Default value for initializing the attribute
Attributes

<table>
<thead>
<tr>
<th>name</th>
<th>(string) Python-readable name for the Attribute on Word and WordToken objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>display_name</td>
<td>(string) Human-readable name for the Attribute</td>
</tr>
<tr>
<td>default_value</td>
<td>(object) Default value for the Attribute. The type of default_value is dependent on the attribute type. Numeric Attributes have a float default value. Factor and Spelling Attributes have a string default value. Tier Attributes have a Transcription default value.</td>
</tr>
<tr>
<td>range</td>
<td>(object) Range of the Attribute, type depends on the attribute type. Numeric Attributes have a tuple of floats for the range for the minimum and maximum. The range for Factor Attributes is a set of all factor levels. The range for Tier Attributes is the set of segments in that tier across the corpus. The range for Spelling Attributes is None.</td>
</tr>
</tbody>
</table>

Methods

```python
__init__(name, att_type[, display_name, ...])
```

Guess the attribute type for a sequence of values

```python
guess_type(values[, trans_delimiters])
```

Guess the attribute type for a sequence of values

```python
sanitize_name(name)
```

Sanitize a display name into a Python-readable attribute name

```python
update_range(value)
```

Update the range of the Attribute with the value specified.

```python
static guess_type(values, trans_delimiters=None)
```

Guess the attribute type for a sequence of values

Parameters

- **values** : list
  - List of strings to evaluate for the attribute type

- **trans_delimiters** : list, optional
  - List of delimiters to look for in transcriptions, defaults to . , ; , and ,

Returns

- **str**
  - Attribute type that had the most success in parsing the values specified

```python
static sanitize_name(name)
```

Sanitize a display name into a Python-readable attribute name

Parameters

- **name** : string
  - Display name to sanitize

Returns

- **str**
  - Sanitized name

```python
update_range(value)
```

Update the range of the Attribute with the value specified. If the attribute is a Factor, the value is added to the set of levels. If the attribute is Numeric, the value expands the minimum and maximum values, if applicable. If the attribute is a Tier, the value (a segment) is added to the set of segments allowed. If the attribute is Spelling, nothing is done.

Parameters

- **value** : object
  - Value to update range with, the type depends on the attribute type
21.1.2 Corpus

class corpustools.corpus.classes.lexicon.Corpus (name)

Lexicon to store information about Words, such as transcriptions, spellings and frequencies

Parameters name : string

  Name to identify Corpus

Attributes

<table>
<thead>
<tr>
<th>name</th>
<th>(str) Name of the corpus, used only for easy of reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
<td>(list of Attributes) List of Attributes that Words in the Corpus have</td>
</tr>
<tr>
<td>wordlist</td>
<td>(dict) Dictionary where every key is a unique string representing a word in a corpus, and each entry is a Word object</td>
</tr>
<tr>
<td>words</td>
<td>(list of strings) All the keys for the wordlist of the Corpus</td>
</tr>
<tr>
<td>specifier</td>
<td>(FeatureSpecifier) See the FeatureSpecifier object</td>
</tr>
<tr>
<td>inventory</td>
<td>(Inventory) Inventory that contains information about segments in the Corpus</td>
</tr>
</tbody>
</table>

Methods

___init__ (name)

add_abstract_tier (attribute, spec) Add a abstract tier (currently primarily for generating CV skeletons from tiers).

add_attribute (attribute[, initialize_defaults]) Add an Attribute of any type to the Corpus or replace an existing Attribute.

add_count_attribute (attribute, ...) Add an Numeric Attribute that is a count of a segments in a tier that match a given specification.

add_tier (attribute, spec) Add a Tier Attribute based on the transcription of words as a new Attribute that includes all segments that match the specification.

add_word (word[, allow_duplicates]) Add a word to the Corpus.

check_coverage () Checks the coverage of the specifier (FeatureMatrix) of the Corpus over the features_to_segments (feature_description) Given a feature description, return the segments in the inventory

find (word[, keyerror, ignore_case]) Search for a Word in the corpus

find_all (spelling) Find all Word objects with the specified spelling

get_features () Get a list of the features used to describe Segments

get_or_create_word (**kwargs) Get a Word object that has the spelling and transcription specified or create that Word, add it to the Corpus and return it.

get_random_subset (size[, new_corpus_name]) Get a new corpus consisting a random selection from the current corpus

iter_sort () Sorts the keys in the corpus dictionary, then yields the

iter_words () Sorts the keys in the corpus dictionary,

key (word)

keys ()

random_word () Return a randomly selected Word

remove_attribute (attribute) Remove an Attribute from the Corpus and from all its Word objects.

remove_word (word_key) Remove a Word from the Corpus using its identifier in the Corpus.

segment_to_features (seg) Given a segment, return the features for that segment.

set_feature_matrix (matrix) Set the feature system to be used by the corpus and make sure every word is using it too.

subset (filters) Generate a subset of the corpus based on filters.

update_inventory (transcription) Update the inventory of the Corpus to ensure it contains all

add_abstract_tier (attribute, spec) Add a abstract tier (currently primarily for generating CV skeletons from tiers).
Specifiers for abstract tiers should be dictionaries with keys that are the abstract symbol (such as ‘C’ or ‘V’) and the values are iterables of segments that should count as that abstract symbols (such as all consonants or all vowels).

Currently only operates on the transcription of words.

**Parameters**

- **attribute**: Attribute
  - Attribute to add/replace
- **spec**: dict
  - Mapping for creating abstract tier

**add_attribute** *(attribute, initialize_defaults=False)*

Add an Attribute of any type to the Corpus or replace an existing Attribute.

**Parameters**

- **attribute**: Attribute
  - Attribute to add or replace
- **initialize_defaults**: boolean
  - If True, words will have this attribute set to the default_value of the attribute, defaults to False

**add_count_attribute** *(attribute, sequence_type, spec)*

Add an Numeric Attribute that is a count of a segments in a tier that match a given specification.

The specification should be either a list of segments or a string of the format ‘+feature1,-feature2’ that specifies the set of segments.

**Parameters**

- **attribute**: Attribute
  - Attribute to add or replace
- **sequence_type**: string
  - Specifies whether to use ‘spelling’, ‘transcription’ or the name of a transcription tier to use for comparisons
- **spec**: list or str
  - Specification of what segments should be counted

**add_tier** *(attribute, spec)*

Add a Tier Attribute based on the transcription of words as a new Attribute that includes all segments that match the specification.

The specification should be either a list of segments or a string of the format ‘+feature1,-feature2’ that specifies the set of segments.

**Parameters**

- **attribute**: Attribute
  - Attribute to add or replace
- **spec**: list or str
  - Specification of what segments should be counted

**add_word** *(word, allow_duplicates=True)*

Add a word to the Corpus. If allow_duplicates is True, then words with identical spelling can be added. They are kept separate by adding a “silent” number to them which is never displayed to the user. If allow_duplicates is False, then duplicates are simply ignored.

**Parameters**

- **word**: Word
Word object to be added

allow_duplicates : bool
If False, duplicate Words with the same spelling as an existing word in the corpus will
not be added

check_coverage()
Checks the coverage of the specifier (FeatureMatrix) of the Corpus over the inventory of the Corpus

Returns list
List of segments in the inventory that are not in the specifier

features_to_segments(feature_description)
Given a feature description, return the segments in the inventory that match that feature description

Feature descriptions should be either lists, such as ['+feature1', '-feature2'] or strings that can be separated
into lists by ',', such as '+feature1,-feature2'.

Parameters feature_description : string or list
Feature values that specify the segments, see above for format

Returns list of Segments
Segments that match the feature description

find(word, keyerror=True, ignore_case=False)
Search for a Word in the corpus If keyerror == True, then raise a KeyError if the word is not found If
keyerror == False, then return an EmptyWord if the word is not found

Parameters word : str
String representing the spelling of the word (not transcription)
keyerror : bool
Set whether a KeyError should be raised if a word is not found

Returns Word
Word that matches the spelling specified

Raises KeyError
If keyerror == True and word is not found

find_all(spelling)
Find all Word objects with the specified spelling

Parameters spelling : string
Spelling to look up

Returns list of Words
Words that have the specified spelling

generate_features()
Get a list of the features used to describe Segments

Returns list of str

get_or_create_word(**kwargs)
Get a Word object that has the spelling and transcription specified or create that Word, add it to the Corpus
and return it.
Parameters spelling : string
    Spelling to search for

transcription : list
    Transcription to search for

Returns Word
    Existing or newly created Word with the spelling and transcription specified

get_random_subset(size, new_corpus_name='randomly_generated')
Get a new corpus consisting a random selection from the current corpus

Parameters size : int
    Size of new corpus

new_corpus_name : str

Returns new_corpus : Corpus
    New corpus object with len(new_corpus) == size

iter_sort()
Sorts the keys in the corpus dictionary, then yields the values in that order

Returns generator
    Sorted Words in the corpus

iter_words()
Sorts the keys in the corpus dictionary, then yields the values in that order

Returns generator
    Sorted Words in the corpus

random_word()
Return a randomly selected Word

Returns Word
    Random Word

remove_attribute(attribute)
Remove an Attribute from the Corpus and from all its Word objects.

Parameters attribute : Attribute
    Attribute to remove

remove_word(word_key)
Remove a Word from the Corpus using its identifier in the Corpus.
If the identifier is not found, nothing happens.

Parameters word_key : string
    Identifier to use to remove the Word

segment_to_features(seg)
Given a segment, return the features for that segment.

Parameters seg : string or Segment
    Segment or Segment symbol to look up
Returns dict
Dictionary with keys as features and values as feature values

**set_feature_matrix**(matrix)
Set the feature system to be used by the corpus and make sure every word is using it too.

**Parameters** matrix : FeatureMatrix
New feature system to use in the corpus

**subset**(filters)
Generate a subset of the corpus based on filters.
Filters for Numeric Attributes should be tuples of an Attribute (of the Corpus), a comparison callable
(__eq__, __neq__, __gt__, __gte__, __lt__, or __lte__) and a value to compare all such attributes in the Corpus to.
Filters for Factor Attributes should be tuples of an Attribute, and a set of levels for inclusion in the subset.
Other attribute types cannot currently be the basis for filters.

**Parameters** filters : list of tuples
See above for format

**Returns** Corpus
Subset of the corpus that matches the filter conditions

**update_inventory**(transcription)
Update the inventory of the Corpus to ensure it contains all the segments in the given transcription

**Parameters** transcription : list
Segment symbols to add to the inventory if needed

### 21.1.3 Inventory

class corpustools.corpus.classes.lexicon.Inventory(data=None)
Inventories contain information about a Corpus’ segmental inventory. In many cases, they are similar to FeatureMatrices, but more tailored to a specific corpus. Where a FeatureMatrix would deal in feature specifications, inventories will deal primarily in sets of segments.

**Parameters** data : dict, optional
Mapping from segment symbol to Segment objects
Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>features</td>
<td>(list) List of all features used as specifications for segments</td>
</tr>
<tr>
<td>possible_values</td>
<td>(set) Set of values that segments use for features</td>
</tr>
<tr>
<td>stresses</td>
<td>(dict) Mapping of stress values to segments that bear that stress</td>
</tr>
<tr>
<td>places</td>
<td>(dict) Mapping from place of articulation labels to sets of segments</td>
</tr>
<tr>
<td>manners</td>
<td>(dict) Mapping from manner of articulation labels to sets of segments</td>
</tr>
<tr>
<td>height</td>
<td>(dict) Mapping from vowel height labels to sets of segments</td>
</tr>
<tr>
<td>backness</td>
<td>(dict) Mapping from vowel backness labels to sets of segments</td>
</tr>
<tr>
<td>vowel_feature</td>
<td>(str) Feature value (i.e., ‘+voc’) that separates vowels from consonants</td>
</tr>
<tr>
<td>voice_feature</td>
<td>(str) Feature value (i.e., ‘+voice’) that codes voiced obstruents</td>
</tr>
<tr>
<td>diph_feature</td>
<td>(str) Feature value (i.e., ‘+diphthong’ or ‘.high’) that separates diphthongs from monophthongs</td>
</tr>
<tr>
<td>rounded_feature</td>
<td>(str) Feature value (i.e., ‘+round’) that codes rounded vowels</td>
</tr>
</tbody>
</table>

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td>(data) Categorize a segment into consonant/vowel, place of articulation, manner of articulation, voicing, vowel height, vowel backness, and vowel rounding.</td>
</tr>
<tr>
<td>categorize(seg)</td>
<td>Categorize a segment into consonant/vowel, place of articulation, manner of articulation, voicing, vowel height, vowel backness, and vowel rounding. For consonants, the category is of the format: (<code>Consonant</code>, PLACE, MANNER, VOICING) For vowels, the category is of the format: (<code>Vowel</code>, HEIGHT, BACKNESS, ROUNDED) Diphthongs are categorized differently: (<code>Diphthong</code>, <code>Vowel</code>) Parameters seg : Segment Segment to categorize Returns tuple or None Returns categories according to the formats above, if any are unable to be calculated, returns None in those places. Returns None if a category cannot be found.</td>
</tr>
<tr>
<td>features_to_segments(feature_description)</td>
<td>Given a feature description, return the segments in the inventory that match that feature description</td>
</tr>
<tr>
<td>find_min_feature_pairs(features[, others])</td>
<td>Find sets of segments that differ only in certain features,</td>
</tr>
<tr>
<td>get_redundant_features(features[, others])</td>
<td>Autodetects redundant features, with the ability to subset</td>
</tr>
<tr>
<td>items()</td>
<td></td>
</tr>
<tr>
<td>keys()</td>
<td></td>
</tr>
<tr>
<td>specify(specifier)</td>
<td>Specify segments in the inventory using a FeatureMatrix</td>
</tr>
<tr>
<td>valid_feature_strings()</td>
<td>Get all combinations of possible_values and features</td>
</tr>
<tr>
<td>values()</td>
<td></td>
</tr>
</tbody>
</table>
Feature descriptions should be either lists, such as `[+feature1`, `-feature2]` or strings that can be separated into lists by `,`, such as `+feature1,-feature2`.

**Parameters** `feature_description` : string or list  
Feature values that specify the segments, see above for format

**Returns** list of Segments  
Segments that match the feature description

**find_min_feature_pairs** *(features, others=None)*  
Find sets of segments that differ only in certain features, optionally limited by a feature specification

**Parameters** `features` : list  
List of features (i.e. ‘back’ or ‘round’)

`others` : list, optional  
Feature specification to limit sets

**Returns** dict  
Dictionary with keys that correspond to the values of `features` and values that are the set of segments with those feature values

**get_redundant_features** *(features, others=None)*  
Autodetects redundant features, with the ability to subset the segments

**Parameters** `features` : list  
List of features to find other features that consistently covary with them

`others` : list, optional  
Feature specification that specifies a subset to look at

**Returns** list  
List of redundant features

**specify** *(specifier)*  
Specify segments in the inventory using a FeatureMatrix

**Parameters** `specifier` : FeatureMatrix  
Specifier to use for updating feature specifications

**valid_feature_strings** ()  
Get all combinations of `possible_values` and `features`

**Returns** list  
List of valid feature strings

### 21.1.4 FeatureMatrix

**class** `corpustools.corpus.classes.lexicon.FeatureMatrix` *(name, feature_entries)*  
An object that stores feature values for segments

**Parameters** `name` : str  
Name to give the FeatureMatrix

`feature_entries` : list
List of dict with one dictionary per segment, requires the key of symbol which identifies the segment

**Attributes**

- **features**
  Get a list of features that are used in this feature system

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>(str) An informative identifier for the feature matrix</td>
</tr>
<tr>
<td>possible_values</td>
<td>(set) Set of values used in the FeatureMatrix</td>
</tr>
<tr>
<td>default_value</td>
<td>(str) Default feature value, usually corresponding to unspecified features</td>
</tr>
<tr>
<td>stresses</td>
<td>(dict) Mapping of stress values to segments that bear that stress</td>
</tr>
<tr>
<td>places</td>
<td>(dict) Mapping from place of articulation labels to a feature specification</td>
</tr>
<tr>
<td>manners</td>
<td>(dict) Mapping from manner of articulation labels to a feature specification</td>
</tr>
<tr>
<td>height</td>
<td>(dict) Mapping from vowel height labels to a feature specification</td>
</tr>
<tr>
<td>backness</td>
<td>(dict) Mapping from vowel backness labels to a feature specification</td>
</tr>
<tr>
<td>vowel_feature</td>
<td>(str) Feature value (i.e., ‘+voc’) that separates vowels from consonants</td>
</tr>
<tr>
<td>voice_feature</td>
<td>(str) Feature value (i.e., ‘+voice’) that codes voiced obstruents</td>
</tr>
<tr>
<td>diph_feature</td>
<td>(str) Feature value (i.e., ‘+diphthong’ or ‘.high’) that separates diphthongs from monophthongs</td>
</tr>
<tr>
<td>rounded_feature</td>
<td>(str) Feature value (i.e., ‘+round’) that codes rounded vowels</td>
</tr>
</tbody>
</table>

**Methods**

- **__init__(name, feature_entries)**
  Add a feature to the feature system

- **add_feature(feature[, default])**
  Add a feature to the feature system

- **add_segment(seg, feat_spec)**
  Add a segment with a feature specification to the feature system

- **categorize(seg)**
  Categorize a segment into consonant/vowel, place of articulation, manner of articulation, voicing, vowel height, vowel backness, and vowel rounding.

- **features_to_segments(feature_description)**
  Given a feature description, return the segments in the inventory

- **generate_generic()**

- **generate_generic_hayes()**

- **generate_generic_names()**

- **generate_generic_spe()**

- **seg_to_feat_line(symbol)**
  Get a list of feature values for a given segment in the order

- **valid_feature_strings()**
  Get all combinations of possible_values and features

- **validate()**
  Make sure that all segments in the matrix have all the features.

**add_feature** *(feature, default=None)*

Add a feature to the feature system

**Attributes**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>feature</td>
<td>(str) Name of the feature to add to the feature system</td>
</tr>
<tr>
<td>default</td>
<td>(str, optional) If specified, set the value for all segments to this value, otherwise use the FeatureMatrix's default_value</td>
</tr>
</tbody>
</table>
add_segment (seg, feat_spec)
   Add a segment with a feature specification to the feature system

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>seg</td>
<td>(str) Segment symbol to add to the feature system</td>
</tr>
<tr>
<td>feat_spec</td>
<td>(dictionary) Dictionary with features as keys and feature values as values</td>
</tr>
</tbody>
</table>

categorize (seg)
   Categorize a segment into consonant/vowel, place of articulation, manner of articulation, voicing, vowel height, vowel backness, and vowel rounding.
   For consonants, the category is of the format:
   (‘Consonant’, PLACE, MANNER, VOICING)
   For vowels, the category is of the format:
   (‘Vowel’, HEIGHT, BACKNESS, ROUNDED)
   Diphthongs are categorized differently:
   (‘Diphthong’, ‘Vowel’)

   Parameters seg : Segment
      
   Returns tuple or None
      Returns categories according to the formats above, if any are unable to be calculated, returns None in those places. Returns None if a category cannot be found.

features
   Get a list of features that are used in this feature system

   Returns list
      Sorted list of the names of all features in the matrix

features_to_segments (feature_description)
   Given a feature description, return the segments in the inventory that match that feature description
   Feature descriptions should be either lists, such as [’+feature1’, ’-feature2’] or strings that can be separated into lists by ’,’, such as ‘+feature1,-feature2’.

   Parameters feature_description : str, list, or dict
      Feature values that specify the segments, see above for format

   Returns list of Segments
      Segments that match the feature description

seg_to_feat_line (symbol)
   Get a list of feature values for a given segment in the order that features are return in get_feature_list
   Use for display purposes

   Returns list
      List of feature values for the symbol, as well as the symbol itself
Phonological CorpusTools Documentation, Release 1.0.0

Attributes

symbol (str) Segment symbol to look up

segments
Return a list of segment symbols that are specified in the feature system

Returns list
List of all the segments with feature specifications

valid_feature_strings()
Get all combinations of possible_values and features

Returns list
List of valid feature strings

validate()
Make sure that all segments in the matrix have all the features. If not, add an unspecified value for that feature to them.

21.1.5 Segment

class corpustools.corpus.classes.lexicon.Segment(symbol)
Class for segment symbols

Parameters symbol: str
Segment symbol

Attributes

features (dict) Feature specification for the segment

Methods

__init__(symbol)

feature_match(specification) Return true if segment matches specification, false otherwise.

minimal_difference(other, features) Check if this segment is a minimal feature difference with another

specify(feature_dict) Specify a segment with a new feature specification

feature_match(specification)
Return true if segment matches specification, false otherwise.

Parameters specification: object
Specification can be a single feature value ‘+feature’, a list of feature values
[‘+feature1’,’-feature2’], or a dictionary of features and values {‘feature1’: ‘+’, ‘feature2’: ‘-’}

Returns bool
True if this segment contains the feature values in the specification

minimal_difference(other, features)
Check if this segment is a minimal feature difference with another segment (ignoring some features)

**Parameters**
- **other**: Segment
  - Segment to compare with
- **features**: list
  - Features that are allowed to vary between the two segments

**Returns**
- bool
  - True if all features other than the specified ones match, False otherwise

**specify**(feature_dict)
Specify a segment with a new feature specification

**Parameters**
- **feature_dict**: dict
  - Feature specification

### 21.1.6 Transcription

**class**

```python
corpustools.corpus.classes.lexicon.Transcription(seg_list)
```

**Parameters**
- **seg_list**: list
  - List of segments that form the transcription. Elements in the list, can be Segments, strings, or BaseAnnotations

**Attributes**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_list</td>
<td>(list) List of strings representing segment symbols</td>
</tr>
<tr>
<td>stress_pattern:</td>
<td>Dictionary with keys of segment indices and values of the stress for that segment</td>
</tr>
<tr>
<td>boundaries</td>
<td>(dict) Possible keys of ‘morpheme’ or ‘tone’ that keeps track of where morpheme or tone boundaries are inserted</td>
</tr>
</tbody>
</table>

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__</code>(seg_list)</td>
<td></td>
</tr>
<tr>
<td><code>find</code>(environment)</td>
<td>Find instances of an EnvironmentFilter in the Transcription</td>
</tr>
<tr>
<td><code>find_nonmatch</code>(environment)</td>
<td>Find all instances of an EnvironmentFilter in the Transcription</td>
</tr>
<tr>
<td><code>match_segments</code>(segments)</td>
<td>Returns a matching segments from a list of segments</td>
</tr>
<tr>
<td><code>with_word_boundaries</code>()</td>
<td>Return the string of segments with word boundaries surrounding them</td>
</tr>
</tbody>
</table>

**find**(environment)
Find instances of an EnvironmentFilter in the Transcription

**Parameters**
- **environment**: EnvironmentFilter
  - EnvironmentFilter to search for

**Returns**
- list
  - List of Environments that fit the EnvironmentFilter

### 21.1. Lexicon classes
find_nonmatch (environment)
    Find all instances of an EnvironmentFilter in the Transcription that match in the middle segments, but don’t match on the sides

    Parameters environment : EnvironmentFilter
        EnvironmentFilter to search for

    Returns list
        List of Environments that fit the EnvironmentFilter’s middle but not the sides

match_segments (segments)
    Returns a matching segments from a list of segments

    Parameters segments : list
        List of Segments or strings to filter the Transcription

    Returns list
        List of segments (in their original order) that match the segment parameter

with_word_boundaries ()
    Return the string of segments with word boundaries surrounding them

    Returns list
        Transcription with word boundaries

21.1.7 Word

class corpustools.corpus.classes.lexicon.Word(**kwargs)
    An object representing a word in a corpus

    Information about the attributes are contained in the Corpus’ attributes.

    Attributes

    spelling (str) A representation of a word that lacks phonological information.
    transcription (Transcription) A representation of a word that includes phonological information.
    frequency (float) Token frequency in a corpus

    Methods

    __init__(**kwargs)
    add_abstract_tier(tier_name, tier_segments) Add an abstract tier to the Word
    add_attribute(tier_name, value) Add an arbitrary attribute to the Word
    add_tier(tier_name, tier_segments) Adds a new tier attribute to the Word
    remove_attribute(attribute_name) Deletes a tier attribute from a Word
    variants([sequence_type]) Get variants and frequencies for a Word

    add_abstract_tier (tier_name, tier_segments)
    Add an abstract tier to the Word

    Parameters tier_name : str
Attribute name
tier_segments: dict
Dictionary with keys of the abstract segments (i.e., ‘C’ or ‘V’) and values that are sets of segments

add_attribute(tier_name, value)
Add an arbitrary attribute to the Word

Parameters tier_name: str
Attribute name
value: object
Attribute value

add_tier(tier_name, tier_segments)
Adds a new tier attribute to the Word

Parameters tier_name: str
Name for the new tier
tier_segments: list of segments
Segments that count for inclusion in the tier

remove_attribute(attribute_name)
Deletes a tier attribute from a Word

Parameters attribute_name: str
Name of tier attribute to be deleted.

Notes
If attribute_name is not a valid attribute, this function does nothing. It does not raise an error.

variants(sequence_type='transcription')
Get variants and frequencies for a Word

Parameters sequence_type: str, optional
Tier name to get variants

Returns dict
Dictionary with keys of Transcriptions and values of their frequencies

21.1.8 EnvironmentFilter
class corpustools.corpus.classes.lexicon.EnvironmentFilter(middle_segments, 
lhs=None, rhs=None)
Filter to use for searching words to generate Environments that match

Parameters middle_segments: set
Set of segments to center environments

lhs: list, optional
List of set of segments on the left of the middle
**Methods**

```python
__init__(middle_segments[, lhs, rhs])
```

- `compile_re_pattern()`: Compile regular expression pattern.
- `is_applicable(sequence)`: Check whether the Environment filter is applicable to the sequence.
- `lhs_count()`: Get the number of elements on the left hand side.
- `rhs_count()`: Get the number of elements on the right hand side.
- `set_lhs(lhs)`: Set segments to the left of the middle.
- `set_rhs(rhs)`: Set segments to the right of the middle.

**is_applicable(sequence)**

Check whether the Environment filter is applicable to the sequence (i.e., the sequence must be greater or equal in length to the EnvironmentFilter).

- **Parameters**
  - `sequence`: list

- **Returns**
  - `bool`

**lhs_count()**: Get the number of elements on the left hand side.

- **Returns**
  - `int`

**rhs_count()**: Get the number of elements on the right hand side.

- **Returns**
  - `int`

## 21.1.9 Environment

**class** `corpustools.corpus.classes.lexicon.Environment(middle, position, lhs=None, rhs=None)`

Specific sequence of segments that was a match for an EnvironmentFilter.

- **Parameters**
  - `middle`: str
    - Middle segment
  - `position`: int
    - Position of the middle segment in the word (to differentiate between repetitions of an environment in the same word)
  - `lhs`: list, optional
    - Segments to the left of the middle segment
  - `rhs`: list, optional
Segments to the right of the middle segment

Methods

```
__init__(middle, position[, lhs, rhs])
```

## 21.2 Speech corpus classes

### 21.2.1 Discourse

```python
class corpustools.corpus.classes.spontaneous.Discourse(**kwargs)
```

Discourse objects are collections of linear text with word tokens

**Parameters**

- **name**: str
  - Identifier for the Discourse
- **speaker**: Speaker
  - Speaker producing the tokens/text (defaults to an empty Speaker)

**Attributes**

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
<td>(list of Attributes) The Discourse object tracks all of the attributes used by its WordToken objects</td>
</tr>
<tr>
<td>words</td>
<td>(dict of WordTokens) The keys are the beginning times of the WordTokens (or their place in a text if it's not a speech discourse) and the values are the WordTokens</td>
</tr>
</tbody>
</table>

**Methods**

- `__init__(**kwargs)`
- `add_attribute(attribute[, initialize_defaults])`: Add an Attribute of any type to the Discourse or replace an existing Attribute.
- `add_word(wordtoken)`: Adds a WordToken to the Discourse
- `create_lexicon()`: Create a Corpus object from the Discourse
- `find_wordtype(wordtype)`: Look up all WordTokens that are instances of a Word
- `keys()`: Returns a sorted list of keys for looking up WordTokens

**add_attribute** *(attribute, initialize_defaults=False)*

Add an Attribute of any type to the Discourse or replace an existing Attribute.

**Parameters**

- **attribute**: Attribute
  - Attribute to add or replace
initialize_defaults : bool
If True, word tokens will have this attribute set to the default_value of the attribute, defaults to False

```
add_word(wordtoken)
```

Adds a WordToken to the Discourse

**Parameters**

wordtoken : WordToken
WordToken to be added

```
create_lexicon()
```

Create a Corpus object from the Discourse

**Returns**

Corpus
Corpus with spelling and transcription from previous Corpus and token frequency from the Discourse

```
find_wordtype(wordtype)
```

Look up all WordTokens that are instances of a Word

**Parameters**

wordtype : Word
Word to look up

**Returns**

list of WordTokens
List of the given Word’s WordTokens in this Discourse

```
has_audio
```

Checks whether the Discourse is associated with a .wav file

**Returns**

bool
True if a .wav file is associated and if that file exists, False otherwise

```
keys()
```

Returns a sorted list of keys for looking up WordTokens

**Returns**

list
List of begin times or indices of WordTokens in the Discourse

### 21.2.2 Speaker

```
class corpustools.corpus.classes.spontaneous.Speaker(name, **kwargs)
```

Speaker objects contain information about the producers of WordTokens or Discourses

**Parameters**

name : string
Name to identify the Speaker

**Attributes**

<table>
<thead>
<tr>
<th>name</th>
<th>(string) Name of Speaker</th>
</tr>
</thead>
<tbody>
<tr>
<td>gender</td>
<td>(string) Gender of Speaker</td>
</tr>
<tr>
<td>age</td>
<td>(int or string) Age of Speaker</td>
</tr>
</tbody>
</table>
Methods

```python
__init__(name, **kwargs)
```

### 21.2.3 SpontaneousSpeechCorpus

**class corpustools.corpus.classes.spontaneous.SpontaneousSpeechCorpus (name, directory)**

SpontaneousSpeechCorpus objects are a collection of Discourse objects and Corpus objects for frequency information.

**Parameters**

- **name**: str
  - Name to identify the SpontaneousSpeechCorpus
- **directory**: str
  - Directory associated with the SpontaneousSpeechCorpus

**Attributes**

- **lexicon**: (Corpus) Corpus object with token frequencies from its Discourses
- **discourses**: (dict) Discourses of the SpontaneousSpeechCorpus indexed by the names of the Discourses

**Methods**

```python
__init__(name, directory)
add_discourse(discourse)
```

**add_discourse(discourse)**

Add a discourse to the SpontaneousSpeechCorpus

**Parameters**

- **discourse**: Discourse
  - Discourse to be added

### 21.2.4 WordToken

**class corpustools.corpus.classes.spontaneous.WordToken (**kwargs)**

WordToken objects are individual productions of Words

**Parameters**

- **word**: Word
  - Word that the WordToken is associated with
- **transcription**: iterable of str
  - Transcription for the WordToken (can be different than the transcription of the Word type). Defaults to None if not specified
- **spelling**: str
  - Spelling for the WordToken (can be different than the spelling of the Word type). Defaults to None if not specified

21.2. Speech corpus classes
begin : float or int
    Beginning of the WordToken (can be specified as either in seconds of time or in position
    from the beginning of the Discourse)
end : float or int
    End of the WordToken (can be specified as either in seconds of time or in position from
    the beginning of the Discourse)

previous_token : WordToken
    The preceding WordToken in the Discourse, defaults to None if not specified
following_token : WordToken
    The following WordToken in the Discourse, defaults to None if not specified
discourse : Discourse
    Parent Discourse object that the WordToken belongs to

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>transcription</td>
<td>(Transcription) The WordToken’s transcription, or the word type’s transcription if the</td>
</tr>
<tr>
<td></td>
<td>WordToken’s transcription is None</td>
</tr>
<tr>
<td>spelling</td>
<td>(str) The WordToken’s spelling, or the word type’s spelling if the WordToken’s spelling is None</td>
</tr>
<tr>
<td>previous_token</td>
<td>(WordToken) The previous WordToken in the Discourse</td>
</tr>
<tr>
<td>following_token</td>
<td>(WordToken) The following WordToken in the Discourse</td>
</tr>
<tr>
<td>duration</td>
<td>(float) The duration of the WordToken</td>
</tr>
</tbody>
</table>

Methods

```
__init__(**kwargs)
add_attribute(tier_name, default_value)
```

21.2.5 Corpus context managers

```
contextmanagers.BaseCorpusContext(corpus, ...)
contextmanagers.CanalonicVariantContext(...)
contextmanagers.MostFrequentVariantContext(...)
contextmanagers.SeparatedTokensVariantContext(...)
contextmanagers.WeightedVariantContext(...)
```
BaseCorpusContext

class corpustools.contextmanagers.BaseCorpusContext (corpus, sequence_type, type_or_token, attribute=None, frequency_threshold=0)

Abstract Corpus context class that all other contexts inherit from.

Parameters

**corpus** : Corpus
Corpus to form context from

**sequence_type** : str
Sequence type to evaluate algorithms on (i.e., 'transcription')

**type_or_token** : str
The type of frequency to use for calculations

**attribute** : Attribute, optional
Attribute to save results to for calculations involving all words in the Corpus

**frequency_threshold** : float, optional
If specified, ignore words below this token frequency

Methods

__init__(corpus, sequence_type, type_or_token)

get_frequency_base([gramsize, halve_edges, ...])
Generate (and cache) frequencies for each segment in the Corpus.

get_phone_probs([gramsize, probability, ...])
Generate (and cache) phonotactic probabilities for segments in the Corpus.

get_frequency_base (gramsize=1, halve_edges=False, probability=False)
Generate (and cache) frequencies for each segment in the Corpus.

Parameters

**halve_edges** : boolean
If True, word boundary symbols (‘#’) will only be counted once per word, rather than twice. Defaults to False.

**gramsize** : integer
Size of n-gram to use for getting frequency, defaults to 1 (unigram)

**probability** : boolean
If True, frequency counts will be normalized by total frequency, defaults to False

Returns

dict
Keys are segments (or sequences of segments) and values are their frequency in the Corpus

get_phone_probs (gramsize=1, probability=True, preserve_position=True, log_count=True)
Generate (and cache) phonotactic probabilities for segments in the Corpus.

Parameters

**gramsize** : integer
Size of n-gram to use for getting frequency, defaults to 1 (unigram)

**probability** : boolean
If True, frequency counts will be normalized by total frequency, defaults to False

`preserve_position` : boolean

If True, segments will in different positions in the transcription will not be collapsed, defaults to True

`log_count` : boolean

If True, token frequencies will be logarithmically-transformed prior to being summed

**Returns** dict

Keys are segments (or sequences of segments) and values are their phonotactic probability in the Corpus

---

**CanonicalVariantContext**

```python
class corpustools.contextmanagers.CannoticalVariantContext (corpus, sequence_type, type_or_token, attribute=None, frequency_threshold=0)
```

Corpus context that uses canonical forms for transcriptions and tiers

See the documentation of `BaseCorpusContext` for additional information

**Methods**

```python
__init__(corpus, sequence_type, type_or_token)
get_frequency_base([gramsize, halve_edges, ...])  Generate (and cache) frequencies for each segment in the Corpus.
get_phone_probs([gramsize, probability, ...])  Generate (and cache) phonotactic probabilities for segments in the Corpus.
```

**get_frequency_base** (gramsize=1, halve_edges=False, probability=False)

Generate (and cache) frequencies for each segment in the Corpus.

**Parameters**

- `halve_edges` : boolean
  - If True, word boundary symbols (‘#’) will only be counted once per word, rather than twice. Defaults to False.

- `gramsize` : integer
  - Size of n-gram to use for getting frequency, defaults to 1 (unigram)

- `probability` : boolean
  - If True, frequency counts will be normalized by total frequency, defaults to False

**Returns** dict

Keys are segments (or sequences of segments) and values are their frequency in the Corpus

**get_phone_probs** (gramsize=1, probability=True, preserve_position=True, log_count=True)

Generate (and cache) phonotactic probabilities for segments in the Corpus.

**Parameters**

- `gramsize` : integer
  - Size of n-gram to use for getting frequency, defaults to 1 (unigram)

- `probability` : boolean
If True, frequency counts will be normalized by total frequency, defaults to False

**preserve_position** : boolean

If True, segments will in different positions in the transcription will not be collapsed, defaults to True

**log_count** : boolean

If True, token frequencies will be logarithmically-transformed prior to being summed

**Returns**  
dict

Keys are segments (or sequences of segments) and values are their phonotactic probability in the Corpus

**MostFrequentVariantContext**

class corpustools.contextmanagers.MostFrequentVariantContext (corpus, sequence_type, type_or_token, attribute=None, frequency_threshold=0)

Corpus context that uses the most frequent pronunciation variants for transcriptions and tiers

See the documentation of *BaseCorpusContext* for additional information

**Methods**

**__init__**(corpus, sequence_type, type_or_token)

**get_frequency_base**(gramsize=1, halve_edges=False, probability=False)  
Generate (and cache) frequencies for each segment in the Corpus.

**get_phone_probs**(gramsize=1, probability=True, preserve_position=True, log_count=True)  
Generate (and cache) phonotactic probabilities for segments in the Corpus.

**Parameters** halve_edges : boolean

If True, word boundary symbols (‘#’) will only be counted once per word, rather than twice. Defaults to False.

gramsize : integer

Size of n-gram to use for getting frequency, defaults to 1 (unigram)

probability : boolean

If True, frequency counts will be normalized by total frequency, defaults to False

**Returns**  
dict

Keys are segments (or sequences of segments) and values are their frequency in the Corpus

**get_phone_probs**(gramsize=1, probability=True, preserve_position=True, log_count=True)  
Generate (and cache) phonotactic probabilities for segments in the Corpus.

**Parameters**  
gramsize : integer

Size of n-gram to use for getting frequency, defaults to 1 (unigram)
**probability**: boolean

If True, frequency counts will be normalized by total frequency, defaults to False

**preserve_position**: boolean

If True, segments will in different positions in the transcription will not be collapsed, defaults to True

**log_count**: boolean

If True, token frequencies will be logarithmically-transformed prior to being summed

**Returns**

*dict*

Keys are segments (or sequences of segments) and values are their phonotactic probability in the Corpus

### SeparatedTokensVariantContext

**class** `corpustools.contextmanagers.SeparatedTokensVariantContext` *(corpus, sequence_type, type_or_token, attribute=None, frequency_threshold=0)*

Corpus context that treats pronunciation variants as separate types for transcriptions and tiers

See the documentation of `BaseCorpusContext` for additional information

#### Methods

**__init__**(corpus, sequence_type, type_or_token)

**get_frequency_base**(gramsize, halve_edges, ...)

Generate (and cache) frequencies for each segment in the Corpus.

**get_phone_probs**(gramsize, probability, ...)

Generate (and cache) phonotactic probabilities for segments in the Corpus.

**get_frequency_base**(gramsize=1, halve_edges=False, probability=False)

Generate (and cache) frequencies for each segment in the Corpus.

**Parameters**

**halve_edges**: boolean

If True, word boundary symbols ('#') will only be counted once per word, rather than twice. Defaults to False.

**gramsize**: integer

Size of n-gram to use for getting frequency, defaults to 1 (unigram)

**probability**: boolean

If True, frequency counts will be normalized by total frequency, defaults to False

**Returns**

*dict*

Keys are segments (or sequences of segments) and values are their frequency in the Corpus

**get_phone_probs**(gramsize=1, probability=True, preserve_position=True, log_count=True)

Generate (and cache) phonotactic probabilities for segments in the Corpus.

**Parameters**

**gramsize**: integer
Size of n-gram to use for getting frequency, defaults to 1 (unigram)

**probability** : boolean

If True, frequency counts will be normalized by total frequency, defaults to False

**preserve_position** : boolean

If True, segments will in different positions in the transcription will not be collapsed, defaults to True

**log_count** : boolean

If True, token frequencies will be logarithmically-transformed prior to being summed

Returns dict

Keys are segments (or sequences of segments) and values are their phonotactic probability in the Corpus

### WeightedVariantContext

class corpustools.contextmanagers.WeightedVariantContext (corpus, sequence_type, type_or_token, attribute=None, frequency_threshold=0)

Corpus context that weights frequency of pronunciation variants by the number of variants or the token frequency for transcriptions and tiers

See the documentation of BaseCorpusContext for additional information

#### Methods

**__init__**(corpus, sequence_type, type_or_token)

**get_frequency_base**(gramsize=1, halve_edges=False, probability=False) Generate (and cache) frequencies for each segment in the Corpus.

**get_phone_probs**(gramsize=1, probability=True, preserve_position=True, log_count=True) Generate (and cache) phonotactic probabilities for segments in the Corpus.

21.2. Speech corpus classes
Parameters

**gramsize**: integer
Size of n-gram to use for getting frequency, defaults to 1 (unigram)

**probability**: boolean
If True, frequency counts will be normalized by total frequency, defaults to False

**preserve_position**: boolean
If True, segments will in different positions in the transcription will not be collapsed, defaults to True

**log_count**: boolean
If True, token frequencies will be logarithmically-transformed prior to being summed

Returns

dict
Keys are segments (or sequences of segments) and values are their phonotactic probability in the Corpus

### 21.2.6 Corpus IO functions

### 21.3 Corpus binaries

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>binary.download_binary(name, path[, call_back])</code></td>
<td>Download a binary file of example corpora and feature matrices.</td>
</tr>
<tr>
<td><code>binary.load_binary(path)</code></td>
<td>Unpickle a binary file</td>
</tr>
<tr>
<td><code>binary.save_binary(obj, path)</code></td>
<td>Pickle a Corpus or FeatureMatrix object for later loading</td>
</tr>
</tbody>
</table>

#### 21.3.1 download_binary

corpustools.corpus.io.binary.download_binary(name, path, call_back=None)
Download a binary file of example corpora and feature matrices.

Names of available corpora: ‘example’ and ‘iphod’


Parameters

**name**: str
Identifier of file to download

**path**: str
Full path for where to save downloaded file

**call_back**: callable
Function that can handle strings (text updates of progress), tuples of two integers (0, total number of steps) and an integer for updating progress out of the total set by a tuple

Returns

bool
True if file was successfully saved to the path specified, False otherwise
21.3.2 load_binary

corpustools.corpus.io.binary.load_binary(path)
    Unpickle a binary file
    Parameters path : str
        Full path of binary file to load
    Returns Object
        Object generated from the text file

21.3.3 save_binary

corpustools.corpus.io.binary.save_binary(obj, path)
    Pickle a Corpus or FeatureMatrix object for later loading
    Parameters obj : Corpus or FeatureMatrix
        Object to save
    path : str
        Full path for where to save object

21.4 Loading from CSV

<table>
<thead>
<tr>
<th>csv.load_corpus_csv(corpus_name, path, delimiter)</th>
<th>Load a corpus from a column-delimited text file</th>
</tr>
</thead>
<tbody>
<tr>
<td>csv.load_feature_matrix_csv(name, path, ...)</td>
<td>Load a FeatureMatrix from a column-delimited text file</td>
</tr>
</tbody>
</table>

21.4.1 load_corpus_csv

corpustools.corpus.io.csv.load_corpus_csv(corpus_name, path, delimiter, annotation_types=None, feature_system_path=None, stop_check=None, call_back=None)
    Load a corpus from a column-delimited text file
    Parameters corpus_name : str
        Informative identifier to refer to corpus
    path : str
        Full path to text file
    delimiter : str
        Character to use for splitting lines into columns
    annotation_types : list of AnnotationType, optional
        List of AnnotationType specifying how to parse text files
    feature_system_path : str
        Full path to pickled FeatureMatrix to use with the Corpus
    stop_check : callable, optional
Optional function to check whether to gracefully terminate early

call_back : callable, optional

Optional function to supply progress information during the function

Returns Corpus
Corpus object generated from the text file

### 21.4.2 load_feature_matrix_csv

corpuertools.corpus.io.csv.load_feature_matrix_csv(name, path, delimiter, stop_check=None, call_back=None)

Load a FeatureMatrix from a column-delimited text file

Parameters name : str
Informative identifier to refer to feature system

path : str
Full path to text file

delimiter : str
Character to use for splitting lines into columns

stop_check : callable, optional
Optional function to check whether to gracefully terminate early

call_back : callable, optional
Optional function to supply progress information during the function

Returns FeatureMatrix
FeatureMatrix generated from the text file

### 21.5 Export to CSV

csv.export_corpus_csv(corpus, path[, ...]) Save a corpus as a column-delimited text file
csv.export_feature_matrix_csv(...[, delimiter]) Save a FeatureMatrix as a column-delimited text file

### 21.5.1 export_corpus_csv

corpuertools.corpus.io.csv.export_corpus_csv(corpus, path, delimiter=’,’,
trans_delimiter=’ ‘, variant_behavior=None)

Save a corpus as a column-delimited text file

Parameters corpus : Corpus
Corpus to save to text file

path : str
Full path to write text file
**21.5.2 export_feature_matrix_csv**

corpustools.corpus.io.csv.export_feature_matrix_csv(feature_matrix, path, delimiter='

Save a FeatureMatrix as a column-delimited text file

**Parameters**

- `feature_matrix`: FeatureMatrix
  - FeatureMatrix to save to text file
- `path`: str
  - Full path to write text file
- `delimiter`: str
  - Character to mark boundaries between columns. Defaults to `'

---

**21.6 TextGrids**

- `textgrid.inspect_discourse_textgrid`
- `textgrid.load_discourse_textgrid`
- `textgrid.load_directory_textgrid`

---

**21.7 Running text**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>text_spelling.inspect_discourse_spelling(path)</code></td>
<td>Generate a list of AnnotationTypes for a specified text file</td>
</tr>
<tr>
<td><code>text_spelling.load_discourse_spelling(...[, ...])</code></td>
<td>Load a discourse from a text file containing running orthographic text</td>
</tr>
<tr>
<td><code>text_spelling.load_directory_spelling(...[, ...])</code></td>
<td>Loads a directory of orthographic texts</td>
</tr>
<tr>
<td><code>text_spelling.export_discourse_spelling(...)</code></td>
<td>Export an orthography discourse to a text file</td>
</tr>
<tr>
<td><code>text_transcription.inspect_discourse_transcription(path)</code></td>
<td>Generate a list of AnnotationTypes for a specified text file</td>
</tr>
<tr>
<td><code>text_transcription.load_discourse_transcription(...)</code></td>
<td>Load a discourse from a text file containing running transcribed text</td>
</tr>
<tr>
<td><code>text_transcription.load_directory_transcription(...)</code></td>
<td>Loads a directory of transcribed texts.</td>
</tr>
<tr>
<td><code>text_transcription.export_discourse_transcription(...)</code></td>
<td>Export a transcribed discourse to a text file</td>
</tr>
</tbody>
</table>
21.7.1 inspect_discourse_spelling

corpustools.corpus.io.text_spelling.inspect_discourse_spelling(path, support_corpus_path=None)

Generate a list of AnnotationTypes for a specified text file for parsing it as an orthographic text

Parameters
- **path**: str
  - Full path to text file

- **support_corpus_path**: str, optional
  - Full path to a corpus to look up transcriptions from spellings in the text

Returns
- list of AnnotationTypes
  - Autodetected AnnotationTypes for the text file

21.7.2 load_discourse_spelling

corpustools.corpus.io.text_spelling.load_discourse_spelling(corpus_name, path, annotation_types=None, lexicon=None, support_corpus_path=None, ignore_case=False, stop_check=None, call_back=None)

Load a discourse from a text file containing running text of orthography

Parameters
- **corpus_name**: str
  - Informative identifier to refer to corpus

- **path**: str
  - Full path to text file

- **annotation_types**: list of AnnotationType, optional
  - List of AnnotationType specifying how to parse text files

- **lexicon**: Corpus, optional
  - Corpus to store Discourse word information

- **support_corpus_path**: str, optional
  - Full path to a corpus to look up transcriptions from spellings in the text

- **ignore_case**: bool, optional
  - Specify whether to ignore case when using spellings in the text to look up transcriptions

- **stop_check**: callable, optional
  - Callable that returns a boolean for whether to exit before finishing full calculation

- **call_back**: callable, optional
  - Function that can handle strings (text updates of progress), tuples of two integers (0, total number of steps) and an integer for updating progress out of the total set by a tuple

Returns
- Discourse
Discourse object generated from the text file

### 21.7.3 load_directory_spelling

corpustools.corpus.io.text_spelling.load_directory_spelling(corpus_name, path, annotation_types=None, support_corpus_path=None, ignore_case=False, stop_check=None, call_back=None)

Loads a directory of orthographic texts

**Parameters**

- **corpus_name**: str
  Name of corpus
- **path**: str
  Path to directory of text files
- **annotation_types**: list of AnnotationType, optional
  List of AnnotationType specifying how to parse text files
- **support_corpus_path**: str, optional
  File path of corpus binary to load transcriptions from
- **ignore_case**: bool, optional
  Specifies whether lookups in the support corpus should ignore case
- **stop_check**: callable, optional
  Optional function to check whether to gracefully terminate early
- **call_back**: callable, optional
  Optional function to supply progress information during the function

**Returns**

SpontaneousSpeechCorpus

Corpus containing Discourses corresponding to the text files

### 21.7.4 export_discourse_spelling

corpustools.corpus.io.text_spelling.export_discourse_spelling(discourse, path, single_line=False)

Export an orthography discourse to a text file

**Parameters**

- **discourse**: Discourse
  Discourse object to export
- **path**: str
  Path to export to
- **single_line**: bool, optional
Flag to enforce all text to be on a single line, defaults to False. If False, lines are 10
words long.

### 21.7.5 inspect_discourse_transcription

```
corpustools.corpus.io.text_transcription.inspect_discourse_transcription(path)
```
Generate a list of AnnotationTypes for a specified text file for parsing it as a transcribed text

**Parameters**
- `path`: str
  - Full path to text file

**Returns**
- list of AnnotationTypes
  - Autodetected AnnotationTypes for the text file

### 21.7.6 load_discourse_transcription

```
corpustools.corpus.io.text_transcription.load_discourse_transcription(corpus_name, path,
annotation_types=None, lexicon=None, feature_system_path=None, stop_check=None,
call_back=None)
```
Load a discourse from a text file containing running transcribed text

**Parameters**
- `corpus_name`: str
  - Informative identifier to refer to corpus
- `path`: str
  - Full path to text file
- `annotation_types`: list of AnnotationType, optional
  - List of AnnotationType specifying how to parse text files
- `lexicon`: Corpus, optional
  - Corpus to store Discourse word information
- `feature_system_path`: str, optional
  - Full path to pickled FeatureMatrix to use with the Corpus
- `stop_check`: callable, optional
  - Optional function to check whether to gracefully terminate early
- `call_back`: callable, optional
  - Optional function to supply progress information during the loading

**Returns**
- Discourse
  - Discourse object generated from the text file
21.7.7 load_directory_transcription

```
corpustools.corpus.io.text_transcription.load_directory_transcription(
corpus_name, path,
annotation_types=None, feature_system_path=None, stop_check=None,
call_back=None)
```

Loads a directory of transcribed texts.

**Parameters**
- **corpus_name** : str
  Name of corpus

- **path** : str
  Path to directory of text files

- **annotation_types** : list of AnnotationType, optional
  List of AnnotationType specifying how to parse text files

- **feature_system_path** : str, optional
  File path of FeatureMatrix binary to specify segments

- **stop_check** : callable, optional
  Optional function to check whether to gracefully terminate early

- **call_back** : callable, optional
  Optional function to supply progress information during the loading

**Returns**
SpontaneousSpeechCorpus

Corpus containing Discourses corresponding to the text files

21.7.8 export_discourse_transcription

```
corpustools.corpus.io.text_transcription.export_discourse_transcription(
discourse, path,
trans_delim='.', single_line=False)
```

Export an transcribed discourse to a text file

**Parameters**
- **discourse** : Discourse
  Discourse object to export

- **path** : str
  Path to export to

- **trans_delim** : str, optional
  Delimiter for segments, defaults to .

- **single_line** : bool, optional
  Indicates single line export
Flag to enforce all text to be on a single line, defaults to False. If False, lines are 10 words long.

## 21.8 Interlinear gloss text

### 21.8.1 inspect_discourse_ilg

corpustools.corpus.io.text_ilg.inspect_discourse_ilg(path[, number])
Generate a list of AnnotationTypes for a specified text file for parsing it as an interlinear gloss text file

**Parameters**
- **path**: str
  - Full path to text file
- **number**: int, optional
  - Number of lines per gloss, if not supplied, it is auto-detected

**Returns**
- list of AnnotationTypes
  - Autodetected AnnotationTypes for the text file

### 21.8.2 load_discourse_ilg

corpustools.corpus.io.text_ilg.load_discourse_ilg(corpus_name, path, annotation_types, lexicon=None, feature_system_path=None, stop_check=None, call_back=None)
Load a discourse from a text file containing interlinear glosses

**Parameters**
- **corpus_name**: str
  - Informative identifier to refer to corpus
- **path**: str
  - Full path to text file
- **annotation_types**: list of AnnotationType
  - List of AnnotationType specifying how to parse the glosses. Can be generated through inspect_discourse_ilg.
- **lexicon**: Corpus, optional
  - Corpus to store Discourse word information
- **feature_system_path**: str
  - Full path to pickled FeatureMatrix to use with the Corpus
- **stop_check**: callable or None
  - Stop function to check for stop words
- **call_back**: function or None
  - Function to call back in the process of parsing
Optional function to check whether to gracefully terminate early

**call_back**: callable or None

Optional function to supply progress information during the loading

**Returns**  Discourse

Discourse object generated from the text file

### 21.8.3 load_directory_ilg

```python
corpustools.corpus.io.text_ilg.load_directory_ilg(corpus_name, path, annotation_types, feature_system_path=None, stop_check=None, call_back=None)
```

Loads a directory of interlinear gloss text files

**Parameters**

- **corpus_name**: str
  - Name of corpus
- **path**: str
  - Path to directory of text files
- **annotation_types**: list of AnnotationType
  - List of AnnotationType specifying how to parse the glosses. Can be generated through `inspect_discourse_ilg`.
- **feature_system_path**: str, optional
  - File path of FeatureMatrix binary to specify segments
- **stop_check**: callable or None
  - Optional function to check whether to gracefully terminate early
- **call_back**: callable or None
  - Optional function to supply progress information during the loading

**Returns**  SpontaneousSpeechCorpus

Corpus containing Discourses corresponding to the text files

### 21.8.4 export_discourse_ilg

```python
corpustools.corpus.io.text_ilg.export_discourse_ilg(discourse, path, trans_delim='.')
```

Export a discourse to an interlinear gloss text file, with a maximal line size of 10 words

**Parameters**

- **discourse**: Discourse
  - Discourse object to export
- **path**: str
  - Path to export to
- **trans_delim**: str, optional
  - Delimiter for segments, defaults to .

21.8. Interlinear gloss text
21.9 Other standards

**21.9.1 inspect_discourse_multiple_files**

corpustools.corpus.io.multiple_files.inspect_discourse_multiple_files(word_path, dialect)

Generate a list of AnnotationTypes for a specified dialect

Parameters

- **word_path**: str
  - Full path to text file
- **dialect**: str
  - Either ‘buckeye’ or ‘timit’

Returns

- list of AnnotationTypes
  - Autodetected AnnotationTypes for the dialect

**21.9.2 load_discourse_multiple_files**

corpustools.corpus.io.multiple_files.load_discourse_multiple_files(corpus_name, word_path, phone_path, dialect, annotation_types=None, lexicon=None, feature_system_path=None, stop_check=None, call_back=None)

Load a discourse from a text file containing interlinear glosses

Parameters

- **corpus_name**: str
  - Informative identifier to refer to corpus
- **word_path**: str
  - Full path to words text file
- **phone_path**: str
  - Full path to phones text file
- **dialect**: str
  - One of ‘buckeye’ or ‘timit’
- **annotation_types**: list of AnnotationType, optional
List of AnnotationType specifying how to parse the glosses. Auto-generated based on dialect.

**lexicon**: Corpus, optional
Corpus to store Discourse word information

**feature_system_path**: str
Full path to pickled FeatureMatrix to use with the Corpus

**stop_check**: callable or None
Optional function to check whether to gracefully terminate early

**call_back**: callable or None
Optional function to supply progress information during the loading

**Returns**
Discourse
Discourse object generated from the text file

### 21.9.3 load_directory_multiple_files

corpustools.corpus.io.multiple_files.load_directory_multiple_files(corpus_name, path, dialect, annotation_types=None, feature_system_path=None, stop_check=None, call_back=None)

Loads a directory of corpus standard files (separated into words files and phones files)

**Parameters**

**corpus_name**: str
Name of corpus

**path**: str
Path to directory of text files

**dialect**: str
One of ‘buckeye’ or ‘timit’

**annotation_types**: list of AnnotationType, optional
List of AnnotationType specifying how to parse the glosses. Auto-generated based on dialect.

**feature_system_path**: str, optional
File path of FeatureMatrix binary to specify segments

**stop_check**: callable or None
Optional function to check whether to gracefully terminate early

**call_back**: callable or None
Optional function to supply progress information during the loading

**Returns**
SpontaneousSpeechCorpus
Corpus containing Discourses corresponding to the text files

### 21.9.4 Analysis functions

#### 21.10 Frequency of alternation

**freq_of_alt.calc_freq_of_alt**

Returns a double that is a measure of the frequency of alternation of two sounds in a given corpus

**Parameters**

- **corpus_context**: CorpusContext
  - Context manager for a corpus
- **seg1**: char
  - A sound segment, e.g. ‘s’, ‘t’
- **seg2**: char
  - A sound segment
- **algorithm**: string
  - The string similarity algorithm
- **max_rel**: double
  - Filters out all words that are higher than max_rel from a relatedness measure
- **min_rel**: double
  - Filters out all words that are lower than min_rel from a relatedness measure
- **phono_align**: boolean (1 or 0), optional
  - 1 means ‘only count alternations that are likely phonologically aligned,’ defaults to not force phonological alignment
- **min_pairs_okay**: bool, optional
  - True means allow minimal pairs (e.g. in English, ‘s’ and ‘t’ do not alternate in minimal pairs, so allowing minimal pairs may skew results)
- **stop_check**: callable, optional
  - Optional function to check whether to gracefully terminate early
- **call_back**: callable, optional
  - Optional function to supply progress information during the function
Returns double
The frequency of alternation of two sounds in a given corpus

21.11 Functional load

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>functional_load.minpair_fl</code></td>
<td>Calculate the functional load of the contrast between two segments as a count of minimal pairs.</td>
</tr>
<tr>
<td><code>functional_load.deltah_fl</code></td>
<td>Calculate the functional load of the contrast between two segments as the decrease in corpus entropy caused by a merger.</td>
</tr>
<tr>
<td><code>functional_load.relative_minpair_fl</code></td>
<td>Calculate the average functional load of the contrasts between a segment and all other segments, as a count of minimal pairs.</td>
</tr>
<tr>
<td><code>functional_load.relative_deltah_fl</code></td>
<td>Calculate the average functional load of the contrasts between a segment and all other segments, as the decrease in corpus entropy caused by a merger.</td>
</tr>
</tbody>
</table>

21.11.1 minpair_fl

corpustools.funcload.functional_load.minpair_fl(corpus_context, segment_pairs, relative_count=True, distinguish_homophones=False, stop_check=None, call_back=None)

Calculate the functional load of the contrast between two segments as a count of minimal pairs.

Parameters corpus_context : CorpusContext
Context manager for a corpus
segment_pairs : list of length-2 tuples of str
The pairs of segments to be conflated.
relative_count : bool, optional
If True, divide the number of minimal pairs by the total count by the total number of words that contain either of the two segments.
distinguish_homophones : bool, optional
If False, then you’ll count sock~shock (sock=clothing) and sock~shock (sock=punch) as just one minimal pair; but if True, you’ll overcount alternative spellings of the same word, e.g. axel~actual and axle~actual. False is the value used by Wedel et al.
stop_check : callable, optional
Optional function to check whether to gracefully terminate early
call_back : callable, optional
Optional function to supply progress information during the function

Returns tuple(int or float, list)
Tuple of: 0. if relative_count‘==False, an int of the raw number of minimal pairs; if ‘relative_count‘==True, a float of that count divided by the total number of words in the corpus that include either ’s1 or s2; and 1. list of minimal pairs.

21.11.2 deltat_h_fl

corpustools.funcload.functional_load.deltah_fl(corpus_context, segment_pairs, stop_check=None, call_back=None)

Calculate the functional load of the contrast between two segments as the decrease in corpus entropy caused by a merger.
**Parameters**

- **corpus_context**: CorpusContext
  - Context manager for a corpus
- **segment_pairs**: list of length-2 tuples of str
  - The pairs of segments to be conflated.
- **stop_check**: callable, optional
  - Optional function to check whether to gracefully terminate early
- **call_back**: callable, optional
  - Optional function to supply progress information during the function

**Returns**

float

The difference between a) the entropy of the choice among non-homophonous words in the corpus before a merger of \( s_1 \) and \( s_2 \) and b) the entropy of that choice after the merger.

### 21.11.3 relative_minpair_fl

corpustools.funcload.functional_load.relative_minpair_fl(corpus_context, segment, relative_count=True, distinguish_homophones=False, output_filename=None, stop_check=None, call_back=None)

Calculate the average functional load of the contrasts between a segment and all other segments, as a count of minimal pairs.

**Parameters**

- **corpus_context**: CorpusContext
  - Context manager for a corpus
- **segment**: str
  - The target segment.
- **relative_count**: bool, optional
  - If True, divide the number of minimal pairs by the total count by the total number of words that contain either of the two segments.
- **distinguish_homophones**: bool, optional
  - If False, then you’ll count sock~shock (sock=clothing) and sock~shock (sock=punch) as just one minimal pair; but if True, you’ll overcount alternative spellings of the same word, e.g. axel~actual and axle~actual. False is the value used by Wedel et al.
- **stop_check**: callable, optional
  - Optional function to check whether to gracefully terminate early
- **call_back**: callable, optional
  - Optional function to supply progress information during the function

**Returns**

int or float
If \( \text{relative\_count} \neq \text{False} \), returns an int of the raw number of minimal pairs. If \( \text{relative\_count} \neq \text{True} \), returns a float of that count divided by the total number of words in the corpus that include either \( s1 \) or \( s2 \).

### 21.11.4 \texttt{relative\_deltah\_fl}

\texttt{corpustools.funcload.functional\_load:relative\_deltah\_fl}(corpus\_context, segment, stop\_check=None, call\_back=None)

Calculate the average functional load of the contrasts between a segment and all other segments, as the decrease in corpus entropy caused by a merger.

- **Parameters**
  - \texttt{corpus\_context} : CorpusContext
    - Context manager for a corpus
  - \texttt{segment} : str
    - The target segment.
  - \texttt{stop\_check} : callable, optional
    - Optional function to check whether to gracefully terminate early
  - \texttt{call\_back} : callable, optional
    - Optional function to supply progress information during the function

- **Returns**
  - float
  - The difference between a) the entropy of the choice among non-homophonous words in the corpus before a merger of \( s1 \) and \( s2 \) and b) the entropy of that choice after the merger.

### 21.12 Kullback-Leibler divergence

\texttt{kl.KullbackLeibler}(corpus\_context, \text{seg1}, ...)  
Calculates KL distances between two Phoneme objects in some context, either the left or right-hand side.

#### 21.12.1 KullbackLeibler

\texttt{corpustools.kl.kl.KullbackLeibler}(corpus\_context, \text{seg1, seg2}, side, outfile=None, stop\_check=False, call\_back=False)

Calculates KL distances between two Phoneme objects in some context, either the left or right-hand side. Segments with identical distributions (ie. \( \text{seg1} = \text{seg2} \)) have a KL of zero. Segments with similar distributions therefore have low numbers, so high numbers indicate possible allophones.

- **Parameters**
  - \texttt{corpus\_context} : CorpusContext
    - Context manager for a corpus
  - \texttt{seg1} : str
    - First segment
  - \texttt{seg2} : str
    - Second segment
  - \texttt{side} : str
One of ‘right’, ‘left’ or ‘both’

outfile : str
    Full path to save output
stop_check : callable or None
    Optional function to check whether to gracefully terminate early
call_back : callable or None
    Optional function to supply progress information during the function

21.13 Mutual information

    mutual_information.pointwise_mi(...[, ...])  Calculate the mutual information for a bigram.

21.13.1 pointwise_mi

corpustools.mutualinfo.mutual_information.pointwise_mi(corpus_context, query, halve_edges=False, in_word=False, stop_check=None, call_back=None)

    Calculate the mutual information for a bigram.

    Parameters corpus_context : CorpusContext
        Context manager for a corpus

    query : tuple
        Tuple of two strings, each a segment/letter

    halve_edges : bool
        Flag whether to only count word boundaries once per word rather than twice, defaults to False

    in_word : bool
        Flag to calculate non-local, non-ordered mutual information, defaults to False

    stop_check : callable or None
        Optional function to check whether to gracefully terminate early

    call_back : callable or None
        Optional function to supply progress information during the function

    Returns float
        Mutual information of the bigram

21.14 Neighborhood density
21.14.1 neighborhood_density

`corpustools.neighdens.neighborhood_density.neighborhood_density`(...)

Calculate the neighborhood density of a particular word in the corpus.

**Parameters**

- `corpus_context`: CorpusContext
  - Context manager for a corpus
- `query`: Word
  - The word whose neighborhood density to calculate.
- `algorithm`: str
  - The algorithm used to determine distance
- `max_distance`: float, optional
  - Maximum edit distance from the queried word to consider a word a neighbor.
- `stop_check`: callable, optional
  - Optional function to check whether to gracefully terminate early
- `call_back`: callable, optional
  - Optional function to supply progress information during the function

**Returns**
tuple(int, set)
- Tuple of the number of neighbors and the set of neighbor Words.

21.14.2 find_mutation_minpairs

`corpustools.neighdens.neighborhood_density.find_mutation_minpairs`(...)

Find all minimal pairs of the query word based only on segment mutations (not deletions/insertions)

**Parameters**

- `corpus_context`: CorpusContext
  - Context manager for a corpus
- `query`: Word
  - The word whose minimal pairs to find
- `stop_check`: callable or None
  - Optional function to check whether to gracefully terminate early
- `call_back`: callable or None
  - Optional function to supply progress information during the function
21.15 Phonotactic probability

phonotactic_probability.phonotactic_probability_vitevitch(...)  Calculate the phonotactic_probability of a particular word using the Vitevitch & Luce algorithm

Parameters
- **corpus_context**: CorpusContext
  - Context manager for a corpus
- **query**: Word
  - The word whose neighborhood density to calculate.
- **probability_type**: str
  - Either ‘unigram’ or ‘bigram’ probability
- **stop_check**: callable or None
  - Optional function to check whether to gracefully terminate early
- **call_back**: callable or None
  - Optional function to supply progress information during the function

Returns
- float
  - Phonotactic probability of the word

21.16 Predictability of distribution

pred_of_dist.calc_prod_all_envs(...[, ...])  Main function for calculating predictability of distribution for two segments

pred_of_dist.calc_prod(...[, ...])  Main function for calculating predictability of distribution for two segments
### 21.16.1 `calc_prod_all_envs`

```python
corpustools.prod.pred_of_dist.calc_prod_all_envs(corporus_context, seg1, seg2, all_info=False, stop_check=None, call_back=None)
```

Main function for calculating predictability of distribution for two segments over a corpus, regardless of environment.

**Parameters**

- **corpus_context**: CorpusContext
  - Context manager for a corpus
- **seg1**: str
  - The first segment
- **seg2**: str
  - The second segment
- **all_info**: bool
  - If true, all the intermediate numbers for calculating predictability of distribution will be returned. If false, only the final entropy will be returned. Defaults to False.
- **stop_check**: callable, optional
  - Optional function to check whether to gracefully terminate early
- **call_back**: callable, optional
  - Optional function to supply progress information during the function

**Returns**

- float or list
  - A list of [entropy, frequency of environment, frequency of seg1, frequency of seg2] if all_info is True, or just entropy if all_info is False.

### 21.16.2 `calc_prod`

```python
corpustools.prod.pred_of_dist.calc_prod(corporus_context, envs, strict=True, all_info=False, stop_check=None, call_back=None)
```

Main function for calculating predictability of distribution for two segments over specified environments in a corpus.

**Parameters**

- **corpus_context**: CorpusContext
  - Context manager for a corpus
- **envs**: list of EnvironmentFilter
  - List of EnvironmentFilter objects that specify environments
- **strict**: bool
  - If true, exceptions will be raised for non-exhaustive environments and non-unique environments. If false, only warnings will be shown. Defaults to True.
- **all_info**: bool
  - If true, all the intermediate numbers for calculating predictability of distribution will be returned. If false, only the final entropy will be returned. Defaults to False.
- **stop_check**: callable, optional

21.16. Predictability of distribution
Optional function to check whether to gracefully terminate early

**call_back**: callable, optional

Optional function to supply progress information during the function

**Returns**

dict

Keys are the environments specified and values are either a list of [entropy, frequency of environment, frequency of seg1, frequency of seg2] if all_info is True, or just entropy if all_info is False.

## 21.17 Symbol similarity

string_similarity.string_similarity(...) This function computes similarity of pairs of words across a corpus.

### 21.17.1 string_similarity

corpustoolssymbolsim.string_similarity.string_similarity(corpus_context, query, algorithm, **kwargs)

This function computes similarity of pairs of words across a corpus.

**Parameters**

**corpus_context**: CorpusContext

Context manager for a corpus

**query**: string, tuple, or list of tuples

If this is a string, every word in the corpus will be compared to it, if this is a tuple with two strings, those words will be compared to each other, if this is a list of tuples, each tuple’s strings will be compared to each other.

**algorithm**: string

The algorithm of string similarity to be used, currently supports ‘khorsi’, ‘edit_distance’, and ‘phono_edit_distance’

**max_rel**: double

Filters out all words that are higher than max_rel from a relatedness measure

**min_rel**: double

Filters out all words that are lower than min_rel from a relatedness measure

**stop_check**: callable or None

Optional function to check whether to gracefully terminate early

**call_back**: callable or None

Optional function to supply progress information during the function

**Returns**

list of tuples:

The first two elements of the tuple are the words that were compared and the final element is their relatedness score

edit_distance.edit_distance(word1, word2, ...) Returns the Levenshtein edit distance between a string from two words
21.17.2 edit_distance

corpustools.symbolsim.edit_distance.edit_distance(word1, word2, sequence_type, max_distance=None)

Returns the Levenshtein edit distance between a string from two words word1 and word2, code drawn from http://en.wikibooks.org/wiki/Algorithm_Implementation/Strings/Levenshtein_distance#Python. The number is the number of operations needed to transform word1 into word2, three operations are possible: insert, delete, substitute.

Parameters

word1: Word
    the first word object to be compared

word2: Word
    the second word object to be compared

string_type: string
    String specifying what attribute of the Word objects to compare, can be “spelling”, “transcription” or a tier

Returns

int:
    the edit distance between two words

khorsi.khorsi(word1, word2, freq_base, ...)

Calculate the string similarity of two words given a set of

21.17.3 khorsi

corpustools.symbolsim.khorsi.khorsi(word1, word2, freq_base, sequence_type, max_distance=None)

Calculate the string similarity of two words given a set of characters and their frequencies in a corpus based on Khorsi (2012)

Parameters

word1: Word
    First Word object to compare

word2: Word
    Second Word object to compare

freq_base: dictionary
    a dictionary where each segment is mapped to its frequency of occurrence in a corpus

sequence_type: string
    The type of segments to be used (‘spelling’ = Roman letters, ‘transcription’ = IPA symbols)

Returns

float:
    A number representing the relatedness of two words based on Khorsi (2012)

phono_edit_distance.phono_edit_distance(...) Returns an analogue to Levenshtein edit distance but uses
21.17.4 phono_edit_distance

corpustools.symbolsim.phono_edit_distance.phono_edit_distance (word1, word2, sequence_type, features)

Returns an analogue to Levenshtein edit distance but uses phonological features instead of characters

**Parameters**

word1: Word

Word object containing transcription tiers which will be compared to another word containing transcription tiers

word2: Word

The other word containing transcription tiers to which word1 will be compared

sequence_type: string

Name of the sequence type (transcription or a tier) to use for comparisons

features: FeatureMatrix

FeatureMatrix that contains all the segments in both transcriptions to be compared

**Returns**

float

the phonological edit distance between two words
22.1 CorpusTools 1.0.1 Release Notes

This is primarily a bugfix release in the 1.0.x series

22.1.1 New features

• Implemented the ability to check for updates to PCT from the executable versions through the help menu of the main window

22.1.2 Functional load

• Fixed a bug in functional load calculations that undercounted the number of minimal pairs found if homophones were present

22.1.3 Corpora

• Numeric filters for subsetting corpora should be working as intended now

22.1.4 TextGrid support

• Improved importing of TextGrids by allowing users to specify what the labels for orthography and transcription tiers are
• Fixed a bug in TextGrid loading where the last segment from the previous word’s transcription was duplicated in the following word’s transcription
• Fixed a bug where loading TextGrids resulted in an empty segment inventory

22.1.5 GUI

• Improved error messages
• Fixed a bug that blocked subsetting a corpus
• Fix for running text window not getting cleared when switching corpora
• Segments should now correctly default to a grid layout when the inventory is displayed and the feature system is missing some Hayes- or SPE-like features

22.2 CorpusTools 1.1.0 Release Notes

This is a major version release for Phonological CorpusTools.

22.2.1 Importing corpora

• Importing corpora functionality in the GUI received a large overhaul
• All types of corpora are imported through a single dialog
• PCT should autodetect many settings based on selected files or directories
• Autodetected settings can be edited and refined by the user
• Basic logging support saves parsing details entered by the user (i.e., multicharacter segments)
• Numbers in transcriptions can be parsed as stress, tone, or as a normal character (note that tone and stress are currently not supported in functions or phonological search)

22.2.2 Pronunciation variants

• All algorithms that analyze segments support four strategies for dealing with pronunciation variants: canonical forms, most frequent variants, separated tokens as types, and tokens weighted by their relative frequencies
• Algorithms that analyze words support two strategies for pronunciation variants: canonical forms and most frequent variants
• Exporting corpora can now export pronunciation variants (and their frequencies)

22.2.3 Functional load

• Added support for finding the average functional load of single segments

22.2.4 Phonotactic probability

• Fixed an issue where calculating biphone probabilities on single segment words would cause errors; now assigns a probability of 0 to those words

22.2.5 Kullback-Leibler divergence

• Added options to bring KL divergence in line with the other functions
• Added command line script for calculating KL divergence
22.2.6 GUI

- Added a dialog to the “View/change feature system” dialog to edit the categorization of segments into a coherent segment chart via features.
- Features can be used as input to the analysis functions, i.e. functional load of voice in the corpus (segments that are +voice compared to segments that are -voice).

Segment selection

- Segment selection has been redone.
- Segments can be selected via the inventory.
- Features can be typed into the filter field, which will highlight segments that will be included with that feature selection.
- Once a feature specification has been entered, that segment set can be locked in.

Environments

- Environment creation has been revamped.
- Users can select a set of center segments.
- Right hand and left hand can be added, with multiple sets of segments on each side.

Known issues

- Help pages for the Mac binary require internet connection to view, due to issues including .html files in the .app binary.
Indices and tables

• genindex
• modindex
• search


Symbols

- `algorithm ALGORITHM`
  command line option, 74, 102
- `all_pairwise_fl`
  command line option, 74
- `all_pairwise_mis`
  command line option, 113
- `context_type CONTEXT_TYPE`
  command line option, 74, 90, 102, 113
- `count_what COUNT_WHAT`
  command line option, 102
- `delimiter DELIMITER`
  command line option, 24
- `distinguish_homophones`
  command line option, 74
- `find_mutation_minpairs`
  command line option, 102
- `frequency_cutoff FREQUENCY_CUTOFF`
  command line option, 74
- `help`
  command line option, 24, 74, 90, 102, 113
- `max_distance MAX_DISTANCE`
  command line option, 102
- `outfile OUTFILE`
  command line option, 75, 90, 102, 113
- `pairs_file_name_or_segment`
  command line option, 74
- `QUERY QUERY`
  command line option, 113
- `relative_fl RELATIVE_FL`
  command line option, 74
- `sequence_type SEQUENCE_TYPE`
  command line option, 75, 90, 102, 113
- `trans_delimiter TRANS_DELIMITER`
  command line option, 24
- `type_or_token TYPE_OR_TOKEN`
  command line option, 74, 90
- `-a ALGORITHM`
  command line option, 74, 102
- `-c CONTEXT_TYPE`
  command line option, 74, 90, 102, 113
- `-d DELIMITER`
  command line option, 24
- `-d DISTINGUISH_HOMOPHONES`
  command line option, 74
- `-d MAX_DISTANCE`
  command line option, 102
- `-e RELATIVE_FL`
  command line option, 74
- `-f FREQUENCY_CUTOFF`
  command line option, 74
- `-h`
  command line option, 24, 74, 90, 102, 113
- `-i`
  command line option, 74, 113
- `-m`
  command line option, 102
- `-o OUTFILE`
  command line option, 75, 90, 102, 113
- `-p PAIRS_FILE_NAME_OR_SEGMENT`
  command line option, 74
- `-q QUERY`
  command line option, 113
- `-s SEQUENCE_TYPE`
  command line option, 75, 90, 102, 113
- `-t TRANS_DELIMITER`
  command line option, 24
- `-t TYPE_OR_TOKEN`
  command line option, 24
- `-w COUNT_WHAT`
  command line option, 74, 90

A

`add_abstract_tier()`
  (corpus-tools.corpus.classes.lexicon.Corpus-method), 127
`add_abstract_tier()`
  (corpus-tools.corpus.classes.lexicon.Word-method), 138
add_attribute() (corpus-tools.corpus.classes.lexicon.Corpus method), 128
add_attribute() (corpustools.corpus.classes.lexicon.Word method), 139
add_attribute() (corpustools.corpus.classes.spontaneous.Discourse method), 141
add_count_attribute() (corpustools.corpus.classes.lexicon.Corpus method), 128
add_discourse() (corpustools.corpus.classes.spontaneous.SpontaneousSpeechCorpus method), 143
add_feature() (corpustools.corpus.classes.lexicon.FeatureMatrix method), 134
add_segment() (corpustools.corpus.classes.lexicon.FeatureMatrix method), 134
add_tier() (corpustools.corpus.classes.lexicon.Corpus method), 128
add_tier() (corpustools.corpus.classes.lexicon.Word method), 139
add_word() (corpustools.corpus.classes.lexicon.Corpus method), 128
add_word() (corpustools.corpus.classes.spontaneous.Discourse method), 142
Attribute (class in corpustools.corpus.classes.lexicon), 125

B
BaseCorpusContext (class in corpustools.contextmanagers), 145

C
calc_freq_of_alt() (in module corpustools.freqalt.freq_of_alt), 162
calc_prod() (in module corpustools.prod.pred_of_dist), 169
calc_prod_all_envs() (in module corpustools.prod.pred_of_dist), 169
CanonicalVariantContext (class in corpustools.contextmanagers), 146
categorize() (corpustools.corpus.classes.lexicon.FeatureMatrix method), 135
categorize() (corpustools.corpus.classes.lexicon.Inventory method), 132
check_coverage() (corpustools.corpus.classes.lexicon.Corpus method), 129
command line option
-algorithm ALGORITHM, 74, 102
-all_pairwise_fl, 74
-all_pairwise_mis, 113
deltah_fl() (in module corpustools.funcload.functional_load), 163
Discourse (class in corpustools.corpus.classes.spontaneous), 141
download_binary() (in module corpustools.corpus.io.binary), 150
E
edit_distance() (in module corpus-tools.symbolsim.edit_distance), 171
Environment (class in corpus-tools.corpus.classes.lexicon), 140
EnvironmentFilter (class in corpus-tools.corpus.classes.lexicon), 139
export_corpus_csv() (in module corpus-tools.corpus.io.csv), 152
export_discourse_ilg() (in module corpus-tools.corpus.io.text_ilg), 159
export_discourse_spelling() (in module corpus-tools.corpus.io.text_spelling), 155
export_discourse_transcription() (in module corpus-tools.corpus.io.text_transcription), 157
export_feature_matrix_csv() (in module corpus-tools.corpus.io.csv), 153

F
feature_match() (corpus-tools.corpus.classes.lexicon.Segment method), 136
FeatureMatrix (class in corpus-tools.corpus.classes.lexicon), 133
features (corpus-tools.corpus.classes.lexicon.FeatureMatrix attribute), 135
features_to_segments() (corpus-tools.corpus.classes.lexicon.Corpus method), 129
features_to_segments() (corpus-tools.corpus.classes.lexicon.FeatureMatrix method), 135
features_to_segments() (corpus-tools.corpus.classes.lexicon.Inventory method), 132
find() (corpus-tools.corpus.classes.lexicon.Corpus method), 129
find() (corpus-tools.corpus.classes.lexicon.Transcription method), 137
find_all() (corpus-tools.corpus.classes.lexicon.Corpus method), 129
find_min_feature_pairs() (corpus-tools.corpus.classes.lexicon.Inventory method), 133
find_mutation_minpairs() (in module corpus-tools.neighdens.neighborhood_density), 167
find_nonmatch() (corpus-tools.corpus.classes.lexicon.Transcription method), 137
find_wordtype() (corpus-tools.corpus.classes.spontaneous.Discourse method), 142

G
get_features() (corpus-tools.corpus.classes.lexicon.Corpus method), 129
get_frequency_base() (corpus-tools.contextmanagers.BaseCorpusContext method), 145
guess_type() (corpus-tools.corpus.classes.lexicon.Attribute static method), 126
has_audio (corpus-tools.corpus.classes.spontaneous.Discourse attribute), 142
inspect_discourse_ilg() (in module corpus-tools.corpus.io.text_ilg), 158
inspect_discourse_multiple_files() (in module corpus-tools.corpus.io.multiple_files), 160

Index 185
seg_to_feat_line() (corpus-
tools.corpus.classes.lexicon.FeatureMatrix
method), 135
Segment (class in corpustools.corpus.classes.lexicon), 136
segment_to_features() (corpus-
tools.corpus.classes.lexicon.Corpus method), 130
segments (corpustools.corpus.classes.lexicon.FeatureMatrix
attribute), 136
SeparatedTokensVariantContext (class in corpus-
tools.contextmanagers), 148
set_feature_matrix() (corpus-
tools.corpus.classes.lexicon.Corpus method), 131

side
command line option, 90
Speaker (class in corpus-
tools.corpus.classes.spontaneous), 142
specify() (corpustools.corpus.classes.lexicon.Inventory
method), 133
specify() (corpustools.corpus.classes.lexicon.Segment
method), 137
SpontaneousSpeechCorpus (class in corpus-
tools.corpus.classes.spontaneous), 143
string_similarity() (in module corpus-
tools.symbolsim.string_similarity), 170
subset() (corpustools.corpus.classes.lexicon.Corpus
method), 131

T
Transcription (class in corpus-
tools.corpus.classes.lexicon), 137

U
update_inventory() (corpus-
tools.corpus.classes.lexicon.Corpus method), 131
update_range() (corpus-
tools.corpus.classes.lexicon.Attribute method), 126

V
valid_feature_strings() (corpus-
tools.corpus.classes.lexicon.FeatureMatrix
method), 136
valid_feature_strings() (corpus-
tools.corpus.classes.lexicon.Inventory method), 133
validate() (corpustools.corpus.classes.lexicon.FeatureMatrix
method), 136
variants() (corpustools.corpus.classes.lexicon.Word
method), 139

W
WeightedVariantContext (class in corpus-
tools.contextmanagers), 149
with_word_boundaries() (corpus-
tools.corpus.classes.lexicon.Transcription
method), 138
Word (class in corpustools.corpus.classes.lexicon), 138
WordToken (class in corpus-
tools.corpus.classes.spontaneous), 143