kPAL Documentation

Release 2.1.1

LUMC, Jeroen F.J. Laros, Martijn Vermaat

August 14, 2015

Contents

kPAL is an analysis toolkit and Python programming library for *k*-mer profiles.

Note: Please cite the following paper if you use kPAL in your own work:

Anvar et al., Determining the quality and complexity of next-generation sequencing data without a reference genome. *Genome Biology* 2014, **15**:555. doi:10.1186/s13059-014-0555-3

CHAPTER 1

User's guide

1.1 Introduction

kPAL provides a command-line analysis toolkit for creating, analysing, and manipulating *k*-mer profiles. It is implemented in Python.

After following *Installation*, kPAL can be started by typing:

```
$ kpal
```

More information about the available commands and their arguments is printed by adding the *-h* argument.

For example, to count all 9-mers in a FASTA file, use the *count* command:

```
$ kpal count -k 9 example.fasta example.k9
```

Below, we provide an overview of all functions of kPAL that are available via the command-line interface:

Command	Description
count	Make a profile from a FASTA file.
merge	Merge two profiles.
balance	Balance a profile on the frequency of *k*-mers and their reverse complements.
showbalance	Calculate the balance of a profile.
meanstd	Show the mean and standard deviation of *k*-mer frequencies.
distr	Calculate the distribution of the frequencies in a profile.
info	Print basic statistics on a given profile.
getcount	Retrieve the count for a particular *k*-mer.
positive	Only keep counts that are positive in both profiles.
scale	Scale profiles such that the total number of *k*-mer frequencies is equal.
shrink	Shrink a profile, effectively reducing *k*-mer length.
shuffle	Randomise a profile.
smooth	Smooth two profiles by collapsing sub-profiles.
distance	Calculate the distance between two profiles.
matrix	Make a pairwise distance matrix for a series of *k*-mer profiles.
cat	Save profiles from several files to one file.

More information about the methods implemented by kPAL can be found in *Methodology*. Some examples of working with the toolkit are shown in *Tutorial*.

1.2 Installation

The kPAL source code is hosted on GitHub. Supported Python versions for running kPAL are 2.6, 2.7, 3.3, and 3.4. kPAL can be installed either via the Python Package Index (PyPI) or from the source code.

1.2.1 Dependencies

kPAL depends on the following Python libraries:

- NumPy
- h5py
- biopython

The easiest way to use kPAL is with the Anaconda distribution which comes with these libraries installed.

Alternatively, you can install them using their binary packages for your operating system.

Although all dependencies will also be automatically installed if they aren't yet when installing kPAL, you may still want to have them installed beforehand. Automatic installation requires compilation from source, which takes a lot of time and needs several compilers and development libraries to be available. The options noted above are often much more convenient.

1.2.2 Latest kPAL release

To install the latest release from PyPI using pip:

```
pip install kPAL
```

1.2.3 kPAL development version

You can also clone and use the latest development version directly from the GitHub repository:

```
git clone https://github.com/LUMC/kPAL.git
cd kPAL
pip install -e .
```

1.3 Methodology

Below we describe the methods implemented by kPAL.

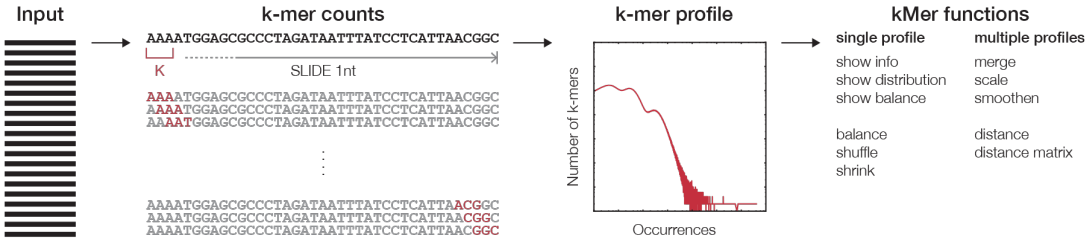

Figure 1

1.3.1 *k*-mer counting

The first step in any *k*-mer analysis is the generation of a profile (*Figure 1*), which is constructed by the *counting* algorithm. The efficiency of the algorithm is improved by encoding the DNA string in binary following this map:

Base	Binary
A	00
C	01
G	10
T	11

Subsequently, the binary encoded *k*-mers are used as the index of a count table. This can be achieved by the concatenation of the binary code for each nucleotide in a given DNA string. This procedure eliminates the need to store the actual *k*-mer sequences since they can be retrieved from decoding the offset in the count table. The binary code for each nucleotide is chosen in such a way that the complement of the nucleotide can be calculated using the binary *NOT* operator. The counting algorithm returns a profile that holds observed counts for all possible substrings of length *k* that can be stored for other analyses.

1.3.2 Distance metrics

Since the *k*-mer profile is in essence a vector of almost independent values, we can use any metric defined for vectors to calculate the *distance* between two profiles. We have implemented two metrics which are the standard Euclidian distance measure and the *multiset* distance measure (1.1). The last metric is parameterised by a function that reflects the distance between a pair. We have implemented two pairwise distance functions (1.2) and (1.3).

For a multiset X, let $S(X)$ denote its underlying set. For multisets X, Y with $S(X), S(Y) \subseteq \{1, 2, \ldots, n\}$ we define:

$$d_f = \frac{\sum_{i=1}^{n} f(x_i, y_i)}{|S(X) \cup S(Y)| + 1} \tag{1.1}$$

$$f_1(x, y) = \frac{|x - y|}{(x + 1)(y + 1)} \tag{1.2}$$

$$f_2(x, y) = \frac{|x - y|}{x + y + 1} \tag{1.3}$$

1.3.3 Strand balance

When analysing sequencing data, which frequently consist of reads from both strands (e.g., due to non strand-specific sample preparation or paired-end sequencing), we can assume that the chance of observing a fragment originating from the plus and minus strands are equal. Additionally, if the sequencing depth is high enough, we expect a *balance* between the frequencies of *k*-mers and their reverse complement in a given *k*-mer profile. Every type of NGS data has an expected balance (i.e., SAGE is not expected to yield a balanced profile while whole genome shotgun sequencing is expected to have a perfectly balanced frequency between *k*-mers and their reverse complement). Thus, *k*-mer balance can indicate the quality of NGS data in respect to over-amplification, insufficient number of reads, or poor capture performance in the case of whole exome sequencing.

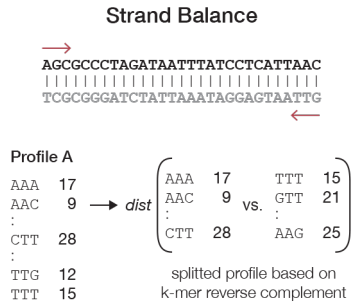

Figure 2

To calculate the balance, first we observe that every *k*-mer has a reverse complement. One of these is lexicographically smaller (or equal in the case of a palindrome) than the other. We first split a profile into two vectors, $A = (a_0, a_1, \ldots)$ and $B = (b_0, b_1, \ldots)$ and where b_i represents the reverse complement of a_i and vice versa. The distance between these vectors can be calculated in the same way as described for pairwise comparison of two full *k*-mer profiles (*Figure 2*).

Additionally, kPAL can forcefully balance the *k*-mer profiles (if desired) by adding the values of each *k*-mer to its reverse complement. This procedure can improve distance calculation if the sequencing depth is too low.

1.3.4 Profile shrinking

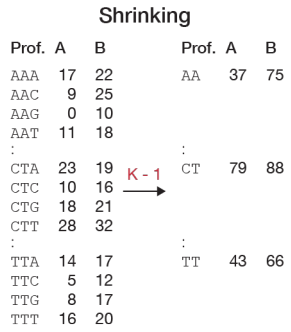

Figure 3

A profile for a certain *k*-mer length contains information about *k*-mers of smaller lengths. This can be seen from the fact that a word w over an alphabet $\mathcal{A}$ has $|\mathcal{A}|$ possible suffixes of length one. To calculate the number of occurrences of w, we simply need to calculate $\sum_{i \in \mathcal{A}} count(w.i)$. This only holds when the *k*-mer length is relatively small compared to the length of the original sequences. Indeed, if a sequence of length l is used for counting at length k, then $(l-k+1)$ *k*-mers are encountered per sequence. However, *shrinking* of a profile will yield $(l - k)$ *k*-mers. Usually, this border effect is small enough to ignore, but should be taken into consideration when counting in large amounts of small (approaching length k) sequences. Shrinking is useful when trying to estimate the best k for a particular purpose. One can start with choosing a relatively large k and then reuse the generated profile to construct a profile of smaller k sizes (*Figure 3*).

1.3.5 Scaling and smoothing

Ideally, the samples that are used to generate profiles are sequenced with the same sample preparation, on the same platform, and most importantly at sufficient depth. However, in practice, this is rarely the case. When two similar samples are sequenced at insufficient depth, it will be reflected in a *k*-mer profile by zero counts for *k*-mers that are not expected to be nullomers. While this is not a problem in itself, the fact that most sequencing procedures have a random selection of sequencing fragments will result in a random distribution of these zero counts. When comparing two profiles, the pairwise distances will be artificially large. *Scaling* the profiles can partially compensate for differences in the sequencing depth but cannot account for nullomers since no distinction can be made between true missing words and artificially missing words. An obvious solution would be to shrink the profile until nullomers

are removed. This method is valid as long as all zero counts reflect artificial nullomers. Otherwise, shrinking will reduce the specificity and does not reflect the true complexity of the sequenced genome. To deal with this problem, we have developed the *pairwise smoothing* function. This method locally shrinks a profile only whe necessary. In this way, we retain information if it is available in both profiles and discard missing data (*Figure 4*).

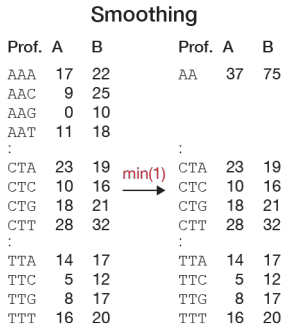

Figure 4

Let P and Q be sub-profiles of words over an alphabet $\mathcal{A}$ of length l (with l devidable by $|\mathcal{A}|$). Let t be a user-defined threshold and let f be a method of summarizing a profile. If $min(f(P), f(Q)) > t$ we divide the profiles in $|\mathcal{A}|$ equal parts and recursively repeat the procedure for each part. If this is not the case, we collapse both P and Q to one word. Implemented methods of summarizing are minimum, mean, and median. In *Figure 4* we show an example of how smoothing might work. We have chosen $f = min$ and $t = 0$ as default parameters.With this method, we can count with a large *k*-mer length k and retain the overall specificity of the profile since this method can automatically select the optimal choice of k locally.

1.4 Tutorial

Before following this tutorial, make sure kPAL is installed properly:

```
$ kpal -h
```

This should print a help message. If it does not, follow *Installation*.

We work with an artificial dataset consisting of 200 *read pairs* from four different samples. They are randomly generated so have no biological relevance.

Note: Download the data: `tutorial.zip`

Now unzip the file and go to the resulting directory:

```
$ unzip -q tutorial.zip
$ cd tutorial
$ ls
a_1.fa  a_2.fa  b_1.fa  b_2.fa  c_1.fa  c_2.fa  d_1.fa  d_2.fa
```

We'll create *k*-mer profiles for these samples and try to compare them.

1.4.1 *k*-mer counting

kPAL can count *k*-mers in any number of fasta files and store the results in one *k*-mer profile file. By default, the profiles in the file are named according to the original fasta filenames.

Let's count 8-mers in the first read for all samples and write the profiles to `reads_1.k8`:

```
$ kpal count -k 8 *_1.fa reads_1.k8
```

Using the *info* command, we can get an overview of our profiles:

```
$ kpal info reads_1.k8
File format version: 1.0.0
Produced by: kPAL 2.0.0

Profile: a_1
- k-mer length: 8 (65536 k-mers)
- Zero counts: 49395
- Non-zero counts: 16141
- Sum of counts: 18600
- Mean of counts: 0.284
- Median of counts: 0.000
- Standard deviation of counts: 0.535

Profile: b_1
- k-mer length: 8 (65536 k-mers)
- Zero counts: 49348
- Non-zero counts: 16188
- Sum of counts: 18600
- Mean of counts: 0.284
- Median of counts: 0.000
- Standard deviation of counts: 0.533

Profile: c_1
- k-mer length: 8 (65536 k-mers)
- Zero counts: 49388
- Non-zero counts: 16148
- Sum of counts: 18600
- Mean of counts: 0.284
- Median of counts: 0.000
- Standard deviation of counts: 0.534

Profile: d_1
- k-mer length: 8 (65536 k-mers)
- Zero counts: 49345
- Non-zero counts: 16191
- Sum of counts: 18600
- Mean of counts: 0.284
- Median of counts: 0.000
- Standard deviation of counts: 0.533
```

1.4.2 Merging profiles

For completeness, we also want to include *k*-mer counts for the second read in our analysis. We can do so using the *merge* command:

```
$ kpal count -k 8 *_2.fa reads_2.k8
$ kpal merge reads_1.k8 reads_2.k8 merged.k8
```

Note: Merging two *k*-mer profiles this way is equivalent to first concatenating both fasta files and counting in the result.

By default, profiles from both files are merged pairwise in alphabetical order. If you need another pairing, you can

provide profile names to use for both files. For example, the following is a more explicit version of the previous command:

```
$ kpal merge reads_1.k8 reads_2.k8 merged.k8 -l a_1 b_1 c_1 d_1 -r a_2 b_2 c_2 d_2
```

We can check that, indeed, the total *k*-mer count has doubled compared to our previous numbers:

```
$ kpal info merged.k8 -p c_1_c_2
File format version: 1.0.0
Produced by: kPAL 2.0.0

Profile: c_1_c_2
- k-mer length: 8 (65536 k-mers)
- Zero counts: 37138
- Non-zero counts: 28398
- Sum of counts: 37200
- Mean of counts: 0.568
- Median of counts: 0.000
- Standard deviation of counts: 0.753
```

1.4.3 Distance between profiles

We can compare two profiles by using a distance function. By default, *distance* uses the multiset distance parameterised by the *prod* pairwise distance function (f_2 in *Distance metrics*):

```
$ kpal distance reads_1.k8 reads_2.k8 -l c_1 -r c_2
c_1 c_2 0.456
```

All profiles in a file can be compared pairwise to produce a distance matrix with the *matrix* command. It first writes the number of profiles compared followed by their names, and then the distance matrix itself. Here we ask it to print the result to standard output (using − for the output filename):

```
$ kpal matrix merged.k8 -
4
a_1_a_2
b_1_b_2
c_1_c_2
d_1_d_2
0.415
0.416 0.416
0.414 0.413 0.414
```

1.4.4 Enforcing strand balance

Todo.

1.4.5 Custom merge functions

Todo.

1.5 Using the Python library

kPAL provides a light-weight Python library for creating, analysing, and manipulating *k*-mer profiles. It is implemented on top of NumPy.

This is a gentle introduction to the library. Consult the *API reference* for more detailed documentation.

1.5.1 *k*-mer profiles

The class `Profile` is the central object in kPAL. It encapsulates *k*-mer counts and provides operations on them.

Instead of using the `Profile` constructor directly, you should generally use one of the profile construction methods. One of those is `Profile.from_fasta()`. The following code creates a 6-mer profile by counting from a FASTA file:

```
>>> from kpal.klib import Profile
>>> p = Profile.from_fasta(open('a.fasta'), 6)
```

The profile object has several properties. For example, we can ask for the *k*-mer length (also known as *k*), the total *k*-mer count, or the median count per *k*-mer:

```
>>> p.length
6
>>> p.total
49995
>>> p.median
12.0
```

Counts are stored as a NumPy `ndarray` of integers, one for each possible *k*-mer, in alphabetical order:

```
>>> len(p.counts)
4096
>>> p.counts
array([ 8, 11,  5, ...,  7, 12, 13])
```

We can get the index in that array for a certain *k*-mer using the `dna_to_binary()` method:

```
>>> i = p.dna_to_binary('AATTAA')
>>> p.counts[i]
13
```

1.5.2 Storing *k*-mer profiles

Todo.

1.5.3 Differences between *k*-mer profiles

Todo.

CHAPTER 2

API reference

2.1 API reference

This part of the documentation covers the interfaces of kPAL's Python library.

2.1.1 *k*-mer profiles

class kpal.klib.**Profile**(*counts*, *name=None*)

A *k*-mer profile provides *k*-mer counts and operations on them.

Instead of using the *Profile* constructor directly, you should generally use one of the profile construction methods:

- *from_file()*
- *from_file_old_format()*
- *from_fasta()*

Parameters

- **counts** (*numpy.ndarray*) – Array of integers where each element is the count for a *k*-mer. Ordering is alphabetically by the *k*-mer.
- **name** (*str*) – Profile name.

balance()

Add the counts of the reverse complement of a *k*-mer to the *k*-mer and vice versa.

binary_to_dna(*number*)

Convert an integer to a DNA string.

Parameters **number** (*int*) – Binary representation of a DNA sequence.

Returns DNA string corresponding to *number*.

Return type str

copy()

Create a copy of the *k*-mer profile. This returns a deep copy, so modifying the copy's *k*-mer counts will not affect the original and vice versa.

Returns Deep copy of profile.

Return type *Profile*

dna_to_binary(*sequence*)
Convert a string of DNA to an integer.

Parameters **sequence** (*str*) – DNA sequence.

Returns Binary representation of *sequence*.

Return type int

classmethod from_fasta(*handle*, *length*, *name=None*)
Create a *k*-mer profile from a FASTA file by counting all *k*-mers in each line.

Parameters

- **handle** (*file-like object*) – Open readable FASTA file handle.
- **length** (*int*) – Length of the *k*-mers.
- **name** (*str*) – Profile name.

Returns A *k*-mer profile.

Return type *Profile*

classmethod from_fasta_by_record(*handle*, *length*, *prefix=None*)
Create *k*-mer profiles from a FASTA file by counting all *k*-mers per record. Profiles are named by the record names.

Parameters

- **handle** (*file-like object*) – Open readable FASTA file handle.
- **length** (*int*) – Length of the *k*-mers.
- **prefix** (*str*) – If provided, the names of the *k*-mer profiles are prefixed with this.

Returns A generator yielding the created *k*-mer profiles.

Return type iterator(Profile)

classmethod from_file(*handle*, *name=None*)
Load the *k*-mer profile from a file.

Parameters

- **handle** (*h5py.File*) – Open readable *k*-mer profile file handle.
- **name** (*str*) – Profile name.

Returns A *k*-mer profile.

Return type *Profile*

classmethod from_file_old_format(*handle*, *name=None*)
Load the *k*-mer profile from a file in the old plaintext format.

Parameters

- **handle** (*file-like object*) – Open readable *k*-mer profile file handle (old format).
- **name** (*str*) – Profile name.

Returns A *k*-mer profile.

Return type *Profile*

classmethod from_sequences(*sequences*, *length*, *name=None*)
Create a *k*-mer profile from *sequences* by counting all *k*-mers in each sequence.

Parameters

- **sequences** (*iterator(str)*) – An iterable of string sequences.
- **length** (*int*) – Length of the *k*-mers.
- **name** (*str*) – Profile name.

Returns A *k*-mer profile.

Return type *Profile*

mean
Mean of *k*-mer counts.

median
Median of *k*-mer counts.

merge(*profile*, *merger=<function <lambda>>*)
Merge two profiles.

Parameters

- **profile** (Profile) – Another *k*-mer profile.
- **merger** (*function*) – A pairwise merge function.

Note that *function* must be vectorized, i.e., it is called directly on NumPy arrays, instead of on their pairwise elements. If your function only works on individual elements, convert it to a NumPy ufunc first. For example:

```
>>> f = np.vectorize(f, otypes=['int64'])
```

name
Profile name.

non_zero
Number *k*-mers with a non-zero count.

number
Number of possible *k*-mers with this length.

print_counts()
Print the *k*-mer counts.

reverse_complement(*number*)
Calculate the reverse complement of a DNA sequence in a binary representation.

Parameters **number** (*int*) – Binary representation of a DNA sequence.

Returns Binary representation of the reverse complement of the sequence corresponding to *number*.

Return type int

save(*handle*, *name=None*)
Save the *k*-mer counts to a file.

Parameters

- **handle** (*h5py.File*) – Open writeable *k*-mer profile file handle.
- **name** (*str*) – Profile name in the file. If not provided, the current profile name is used, or the first available number from 1 consecutively if the profile has no name.

Returns Profile name in the file.

Return type str

shrink(*factor=1*)
Shrink the profile, effectively reducing the value of *k*.

Note that this operation may give slightly different values than counting at a lower *k* directly.

Parameters **factor** (*int*) – Shrinking factor.

shuffle()
Randomise the profile.

split()
Split the profile into two lists, every position in the first list has its reverse complement in the same position in the second list and vice versa. All counts are doubled, so we can equaly distribute palindrome counts over both lists.

Note that the returned counts are not *k*-mer profiles. They can be used to show the balance of the original profile by calculating the distance between them.

Returns The doubled forward and reverse complement counts.

Return type numpy.ndarray, numpy.ndarray

std
Standard deviation of *k*-mer counts.

total
Sum of *k*-mer counts.

2.1.2 *k*-mer profile distances

class kpal.kdistlib.**ProfileDistance**(*do_balance=False*, *do_positive=False*, *do_smooth=False*, *summary=<Mock id='139953457026000'>*, *threshold=0*, *do_scale=False*, *down=False*, *distance_function=None*, *pairwise=<function <lambda>>*)
Class of distance functions.

distance(*left*, *right*)
Calculate the distance between two *k*-mer profiles.

Parameters **left, right** (kpal.klib.Profile) – Profiles to calculate distance between.

Returns The distance between *left* and *right*.

Return type float

dynamic_smooth(*left*, *right*)
Smooth two profiles by collapsing sub-profiles that do not meet the requirements governed by the selected summary function and the threshold.

Parameters **left, right** (kpal.klib.Profile) – Profiles to smooth.

kpal.kdistlib.**distance_matrix**(*profiles*, *output*, *precision*, *dist*)
Make a distance matrix for any number of *k*-mer profiles.

Parameters

- **profiles** (*list(Profile)*) – List of profiles.
- **output** (*file-like object*) – Open writable file handle.
- **precision** (*int*) – Number of digits in the output.

- **dist** (kpal.kdistlib.ProfileDistance) – A distance functions object.

2.1.3 Metrics

General library containing metrics and helper functions.

kpal.metrics.**cosine_similarity**(*left*, *right*)
Calculate the Cosine similarity between two vectors.

Parameters **left, right** (*array_like*) – Vector.

Returns The Cosine similarity between *left* and *right*.

Return type float

kpal.metrics.**distribution**(*vector*)
Calculate the distribution of the values in a vector.

Parameters **vector** (*iterable(int)*) – A vector.

Returns A list of *(value, count)* pairs.

Return type list(int, int)

kpal.metrics.**euclidean**(*left*, *right*)
Calculate the Euclidean distance between two vectors.

Parameters **left, right** (*array_like*) – Vector.

Returns The Euclidean distance between *left* and *right*.

Return type float

kpal.metrics.**get_scale**(*left*, *right*)
Calculate scaling factors based upon total counts. One of the factors is always one (the other is either one or larger than one).

Parameters **left, right** (*array_like*) – A vector.

Returns A tuple of scaling factors.

Return type float, float

kpal.metrics.**mergers = {u'int': <function <lambda> at 0x7f49740ea500>, u'sum': <function <lambda> at 0x7f49740ea41**
Merge functions. Arguments should be of type *numpy.ndarray*.

kpal.metrics.**multiset**(*left*, *right*, *pairwise*)
Calculate the multiset distance between two vectors.

Parameters

- **left, right** (*array_like*) – Vector.
- **pairwise** (*function*) – A pairwise distance function.

Returns The multiset distance between *left* and *right*.

Return type float

Note that *function* must be vectorized, i.e., it is called directly on NumPy arrays, instead of on their pairwise elements. If your function only works on individual elements, convert it to a NumPy ufunc first. For example:

```
>>> f = np.vectorize(f, otypes=['float'])
```

kpal.metrics.**pairwise = {u'sum': <function <lambda> at 0x7f49740ea398>, u'prod': <function <lambda> at 0x7f49740e**
Pairwise distance functions. Arguments should be of type *numpy.ndarray*.

kpal.metrics.**positive**(*vector*, *mask*)
Set all zero positions in *mask* to zero in *vector*.

Parameters **vector, mask** (*array_like*) – Vector.

Returns *vector* with all zero positions in *mask* set to zero.

Return type numpy.ndarray

kpal.metrics.**scale_down**(*left*, *right*)
Normalise scaling factor between 0 and 1.

Parameters **left, right** (*float*) – Scaling factors.

Returns Tuple of normalised scaling factors.

Return type float, float

kpal.metrics.**summary = {u'average': <Mock id='139953456429392'>, u'median': <Mock id='139953456429328'>, u'min'**
Summary functions.

kpal.metrics.**vector_distance = {u'default': None, u'euclidean': <function euclidean at 0x7f49740ea230>, u'cosine': <**
Vector distance functions.

kpal.metrics.**vector_length**(*vector*)
Calculate the Euclidean length of a vector.

Parameters **vector** (*array_like*) – A vector.

Returns The length of *vector*.

Return type float

CHAPTER 3

Additional notes

3.1 Development

Development of kPAL happens on GitHub: https://github.com/LUMC/kPAL

3.1.1 Contributing

Contributions to kPAL are very welcome! They can be feature requests, bug reports, bug fixes, unit tests, documentation updates, or anything els you may come up with.

Start by installing all kPAL development dependencies:

```
$ pip install -r requirements.txt
```

This installs dependencies for building the documentation and running unit tests.

After that you'll want to install kPAL in *development mode*:

```
$ pip install -e .
```

Note: Instead of copying the source code to the installation directory, this only links from the installation directory to the source code such that any changes you make to it are directly available in the environment.

3.1.2 Documentation

The latest documentation with user guide and API reference is hosted at Read The Docs.

You can also compile the documentation directly from the source code by running `make html` from the `doc/` subdirectory. This requires Sphinx to be installed.

3.1.3 Unit tests

To run the unit tests with pytest, just run:

```
$ py.test
```

Use tox to run the unit tests in all supported Python environments automatically:

```
$ tox
```

3.1.4 Coding style

In general, try to follow the PEP 8 guidelines for Python code and PEP 257 for docstrings.

You can use the flake8 tool to assist in style and error checking.

3.1.5 Versioning

A normal version number takes the form X.Y.Z where X is the major version, Y is the minor version, and Z is the patch version. Development versions take the form X.Y.Z.dev where X.Y.Z is the closest future release version.

Note that this scheme is not 100% compatible with SemVer which would require X.Y.Z-dev instead of X.Y.Z.dev but compatibility with setuptools is more important for us. Other than that, version semantics are as described by SemVer.

Releases are published at PyPI and available from the git repository as tags.

Release procedure

Releasing a new version is done as follows:

1. Make sure the section in the `CHANGES.rst` file for this release is complete and there are no uncommitted changes.

   Note: Commits since release X.Y.Z can be listed with `git log vX.Y.Z..` for quick inspection.

2. Update the `CHANGES.rst` file to state the current date for this release and edit `kpal/__init__.py` by updating *__date__* and removing the `dev` value from *__version_info__*.

   Commit and tag the version update:

   ```
   git commit -am 'Bump version to X.Y.Z'
   git tag -a 'vX.Y.Z'
   git push --tags
   ```

3. Upload the package to PyPI:

   ```
   python setup.py sdist upload
   ```

4. Add a new entry at the top of the `CHANGES.rst` file like this:

   ```
   Version X.Y.Z+1
   ---------------

   Release date to be decided.
   ```

   Increment the patch version and add a `dev` value to *__version_info__* in `kpal/__init__.py` and commit these changes:

   ```
   git commit -am 'Open development for X.Y.Z+1'
   ```

3.2 *k*-mer profile file format

The file format kPAL uses to store *k*-mer profiles is HDF5. Here we describe the structure within a *k*-mer profile file.

3.2.1 Versioning

The file format is versioned roughly according to semantic versioning. Software designed to work with files in version *MAJOR.MINOR.PATCH* should be able to work with files in later versions with the same *MAJOR* version without modification.

3.2.2 Current version: 1.0.0

The HDF5 toplevel attributes are:

- **format** (*string*) – This is always set to `kMer`.
- **version** (*string*) – Currently `1.0.0`.
- **producer** (*string*) – Anything, for example `My k-mer program 1.2.1`.

Each *k*-mer profile is a dataset under the `/profiles` group, named `/profiles/<profile_name>`. The data is a one-dimensional array of integers of length 4^k (where k is the *k*-mer length) and is gzip compressed. This dataset has the following attributes:

- **length** (*integer*): *k*-mer length (also know as *k*).
- **total** (*integer*): Sum of *k*-mer counts.
- **non_zero** (*integer*): Number of *k*-mers with a non-zero count.
- **mean** (*float*): Mean of *k*-mer counts.
- **median** (*integer*): Median of *k*-mer counts.
- **std** (*float*): Standard deviation of *k*-mer counts.

Within one file, all profiles must have the same value for the *length* attribute.

All strings and object names in the file are unicode strings encoded as described in the h5py documentation.

3.2.3 Changes from older versions

None yet.

3.3 Changelog

This is a record of changes made between each kMer release.

3.3.1 Version 2.1.1

Released on August 14th, 2015.

- Option to create a *k*-mer profile per FASTA record instead of per FASTA file (use `kpal count --by-record` on the command line or *kpal.klib.Profile.from_fasta_by_record* in the Python API).

- GitHub project moved to LUMC/kPAL.
- Change default precision to 10 decimals.

3.3.2 Version 2.1.0

Released on November 21st, 2014.

- Save profiles from several files to one file (`cat` subcommand).

3.3.3 Version 2.0.1

Released on November 21st, 2014.

- Fixed a major bug that made the command line interface unable to start.

3.3.4 Version 2.0.0

Released on November 18th, 2014.

- Rename from kMer to kPAL (k-mer profile analysis library). The Python package is now *kpal* (was *k_mer*). The command line interface is now `kpal` (was `kMer`).

3.3.5 Version 1.0.1

Released on October 3rd, 2014.

- Fix typo in setuptools trove classifier which made it impossible to push to PyPI.

3.3.6 Version 1.0.0

Released on October 2nd, 2014.

- Also count *k*-mers if *k* equals the length of the string.
- Python 2.6 compatibility.
- Added unit tests and a tox configuration.
- Use a NumPy ndarray for storing *k*-mer counts.
- New multi-profile HDF5 file format (see *k-mer profile file format*).
- Fix splitting a profile for calculating balance. Palindromes were previously not taken into account when splitting a profile. We now double all counts, so palindrome counts can be evenly distributed over both sides (see GitLab #1).
- Our own implementation of a vector's median contained two bugs. Better to use a library for this.
- Fix Euclidean distance between two vectors. Don't add one to the sum of squares. The distance between two empty vectors should be 0, not 1.
- Rename *k_mer.klib.kMer* to *k_mer.klib.Profile*.
- Support Python 3.3 and 3.4 (*see below*).
- Generalize custom function arguments (*see below*).

- Travis CI configuration.
- Sphinx documentation including a user guide and API reference.
- Renamed the *index* command to *count*.
- Renamed the *diff* command to *distance* and the builtin pairwise distance functions from *diff-prod* and *diff-sum* to just *prod* and *sum*.

Support Python 3.3 and 3.4

TL;DR: kMer supports Python 2 and 3 and every module has the following line at the top:

```
>>> from __future__ import (absolute_import, division, print_function,
                            unicode_literals)
```

We now support Python versions 2.6, 2.7, 3.3, and 3.4 in a single codebase without using 2to3. We don't support Python 3.2 because BioPython does not.

We use the Python future package as a compatibility layer between Python 2 and Python 3. The goal is to use a single, clean Python 3.x-compatible codebase to support both Python 2 and Python 3 with minimal overhead.

Most changes are quite straightforward (e.g., absolute imports, print statement, division operator). The main painpoint is of course the bytestring versus unicode story. We now import unicode_literals in each module and maintain that all text in kMer is unicode (*unicode* in Python 2, *str* in Python 3).

Generalize custom function arguments

Custom function arguments in the command line interface can now be either a Python expression or importable name. For example, all commands accepting a summary function argument, also except a custom summary function argument which should be one of:

1. A Python expression over the NumPy ndarray *values* (e.g., `np.max(values)`).
2. An importable name (e.g., `package.module.summary`) that can be called with an ndarray as argument.

Likewise for custom merger and pairwise functions (here the expression is over the two NumPy ndarrays *left* and *right*).

3.3.7 Version 0.3.0

Released on July 3rd, 2014.

- Usage of the Euclidean distance is now handled differently, breaking backwards compatibility.
- Added Cosine similarity measure and generalised distance parameters.
- Fixed broken setup script.
- Added custom merging functionality.

3.3.8 Version 0.2.0

Released on March 23rd, 2014.

- New command line interface, using positional arguments for required parameters.
- Added checking for existing files to prevent overwriting them.

- Fixed a bug in the scale subcommand that prevented scaling.
- Added a version parameter.
- Updated the homepage.
- Made code PEP 8 compliant.
- Switched to Sphynx docstrings.
- Added keyword selection for distance and smoothing functions.
- Added support for custom distance and smoothing functions.
- Added CHANGELOG and README.

3.3.9 Version 0.1.0

Released on September 24th, 2013.

- Start of log.

3.4 Copyright

kPAL is licensed under the MIT License, meaning you can do whatever you want with it as long as all copies include these license terms. The full license text can be found below.

3.4.1 Authors

kPAL is written and maintained by Jeroen F.J. Laros at Leiden University Medical Center and includes contributions by Martijn Vermaat.

- Leiden University Medical Center <humgen@lumc.nl>
- Jeroen F.J. Laros <j.f.j.laros@lumc.nl>
- Martijn Vermaat <martijn@vermaat.name>

3.4.2 License

3.4.3 Citations

Please cite the following paper if you use kPAL in your own work:

Anvar et al., Determining the quality and complexity of next-generation sequencing data without a reference genome. *Genome Biology* 2014, **15**:555. doi:10.1186/s13059-014-0555-3

```
@Article{kpal2014,
  AUTHOR = {Anvar, Seyed and Khachatryan, Lusine and Vermaat, Martijn and
    van Galen, Michiel and Pulyakhina, Irina and Ariyurek, Yavuz and
    Kraaijeveld, Ken and den Dunnen, Johan and de Knijff, Peter and 't
    Hoen, Peter and Laros, Jeroen},
  TITLE = {Determining the quality and complexity of next-generation
    sequencing data without a reference genome},
  JOURNAL = {Genome Biology},
  VOLUME = {15},
  YEAR = {2014},
  NUMBER = {12},
  PAGES = {555},
  URL = {http://genomebiology.com/2014/15/12/555},
  DOI = {10.1186/s13059-014-0555-3},
  ISSN = {1465-6906},
  ABSTRACT = {We describe an open-source kPAL package that facilitates an
    alignment-free assessment of the quality and comparability of
    sequencing datasets by analyzing k-mer frequencies. We show that kPAL
    can detect technical artefacts such as high duplication rates, library
    chimeras, contamination and differences in library preparation
    protocols. kPAL also successfully captures the complexity and
    diversity of microbiomes and provides a powerful means to study
    changes in microbial communities. Together, these features make kPAL
    an attractive and broadly applicable tool to determine the quality and
    comparability of sequence libraries even in the absence of a reference
    sequence. kPAL is freely available at https://github.com/LUMC/kPAL
    webcite.},
}
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

k

B

C

D

E

F

G

K

M

N

P

R

S

T

V