attelo Documentation

Release 0.2

IRIT MELODI team

February 24, 2017

Contents

Contents:

CHAPTER 1

User manual

Unfortunately we do not have much of a user manual at the time of this writing, but we hope to have enough of skeleton to enable us to write one over time.

Getting started

Attelo is mostly a parsing library with a couple of helper command line tools on the side.

The bulk of attelo usage goes through the API. Below is an example showing how you might run a simple attelo decoding cross-fold validation experiment (This is *doc/quickstart.py* in the attelo source tree)

```
"""
Example minature attelo evaluation for a dataset
"""

from __future__ import print_function

from os import path as fp
import os
import sys

from sklearn.linear_model import (LogisticRegression)

from attelo.decoding.mst import (MstDecoder,
                                 MstRootStrategy)
from attelo.decoding.util import (prediction_to_triples)

from attelo.learning.local import (SklearnAttachClassifier,
                                   SklearnLabelClassifier)
from attelo.parser.full import (JointPipeline)

from attelo.fold import (make_n_fold,
                         select_testing,
                         select_training)
from attelo.io import (load_multipack,
                       write_predictions_output)
from attelo.report import (CombinedReport,
                           EdgeReport)
from attelo.score import (score_edges)
from attelo.table import (DataPack)
from attelo.util import (mk_rng, Team)
```

```
# pylint: disable=invalid-name

WORKING_DIR = 'doc/example-corpus'
PREFIX = fp.join(WORKING_DIR, 'tiny')
TMP_OUTPUT = '/tmp/mini-evaluate'
if not fp.exists(TMP_OUTPUT):
    os.makedirs(TMP_OUTPUT)

# load the data
mpack = load_multipack(PREFIX + '.edus',
                       PREFIX + '.pairings',
                       PREFIX + '.features.sparse',
                       PREFIX + '.features.sparse.vocab',
                       verbose=True)

# divide the dataset into folds
num_folds = min((10, len(mpack)))
fold_dict = make_n_fold(mpack, num_folds, mk_rng())

# select a decoder and a learner team
decoder = MstDecoder(root_strategy=MstRootStrategy.fake_root)
learners = Team(attach=SklearnAttachClassifier(LogisticRegression()),
                label=SklearnLabelClassifier(LogisticRegression()))

# put them together as a parser
parser = JointPipeline(learner_attach=learners.attach,
                       learner_label=learners.label,
                       decoder=decoder)

# run cross-fold evaluation
scores = []
for fold in range(num_folds):
    print(">>> doing fold ", fold + 1, file=sys.stderr)
    print("training ... ", file=sys.stderr)
    # learn a model for the training data for this fold
    train_packs = select_training(mpack, fold_dict, fold).values()
    parser.fit(train_packs,
               [x.target for x in train_packs])

    fold_predictions = []
    # decode each document separately
    test_pack = select_testing(mpack, fold_dict, fold)
    for onedoc, dpack in test_pack.items():
        print("decoding on file : ", onedoc, file=sys.stderr)
        dpack = parser.transform(dpack)
        prediction = prediction_to_triples(dpack)
        # print("Predictions: ", prediction)
        # record the prediction score
        scores.append(score_edges(dpack, prediction))
        # optional: save the predictions for further inspection
        fold_predictions.extend(prediction)

    # optional: write predictions for this fold
    output_file = fp.join(TMP_OUTPUT, 'fold-%d' % fold)
    print("writing: %s" % output_file, file=sys.stderr)
    write_predictions_output(DataPack.vstack(test_pack.values()),
                             fold_predictions, output_file)
```

```
report = EdgeReport(scores)

# a combined report provides scores for multiple configurations
# here, we are only using it for the single config
combined_report = CombinedReport(EdgeReport,
                                 {('maxent', 'mst'): report})
print(combined_report.table())
```

Input format

Input to attelo consists of three files two of which are aligned:

- an EDU input file with one line per discourse unit
- a pairings file with one line per EDU *pair*
- a features file also with one line per EDU *pair*

EDU inputs

- global id: used by your application, arbitrary string? (NB: *ROOT* is a special name: no EDU should be named that, but all EDUs can have ROOT as a potential parent)
- text: essentially for debugging purposes, used by attelo graph to provide a visualisation of parses
- grouping (eg. file name, dialogue id): edus are only ever connected with edus in the same group. Also, folds are built on the basis of EDU groupings
- subgrouping (eg. sentence id): any common subunit that can hold multiple EDUs (use the EDU id itself if there is no useful notion of subgrouping). Some decoders may try to treat links between EDUs in the same subgrouping differently from the general case
- span start: (int): used by decoders to order EDUs and determine their adjacency
- span end: (int): see span start

```
d1_492      sheep for wood? dialogue_1      sent1   0       15
d1_493      nope, not me    dialogue_1      sent2   16      28
d1_494      not me either   dialogue_1      sent2   29      42
```

Pairings

The pairings file is a tab-delimited list of (parent, child) pairs, with each element being either an EDU global id (from the EDU inputs), or the distinguished label ROOT. Each row in this file is corresponds with a row in the feature files

```
ROOT        d1_492
d1_493      d1_492
d1_494      d1_492
ROOT        d1_493
d1_492      d1_493
d1_494      d1_493
ROOT        d1_494
d1_492      d1_494
d1_493      d1_494
```

Note that attelo can also accept pairings files with a third column (which it ignores)

Features

Features and labels are supplied as in (multiclass) libsvm/svmlight format.

Relation labels

You should supply a single comment at the very beginning of the file, which attelo can use to associate relation labels with string values

```
# labels: <space delimited list of labels>
```

The labels 'UNRELATED' must exist and be used for any edu pairs which are not related/attached. For example, in the below, the second and fourth EDU pairs are not considered to be related

```
# labels: elaboration narration continuation UNRELATED ROOT
1 1:1 2:1
4 1:2
2 1:3 3:1
4 1:1
3 1:2
```

Also, if intersentential learning/decoding is used, the label 'ROOT' must also be exist and be used for links from the ROOT edu.

Note that labels are assumed to start from 1.

Categorical features

Attelo no longer provides direct support for categorical features, that is, features whose possible values are members of a set (eg. POS tag). You should perform one hot encoding on any categorical features you have. Luckily, with the svmlight sparse format, this can be done with no additional cost in space and also opens the door for more straightforward filtering on your part.

Other notes on features

Don't forget that the order that features appear in must correspond to the order that pairings appear in the EDU file

Output format

The output format is similar to the EDU pairings format. It is a tab-delimited text file divided into rows and columns. The columns are

- parent EDU id
- child EDU id
- relation label (or UNRELATED if no link between the two)

```
ROOT    d1_492  ROOT
d1_493  d1_492  UNRELATED
d1_494  d1_492  UNRELATED
ROOT    d1_493  UNRELATED
d1_492  d1_493  elaboration
d1_494  d1_493  result
ROOT    d1_494  UNRELATED
```

```
d1_492  d1_494  narration
d1_493  d1_494  UNRELATED
```

The output above corresponds to the graph below

```
ROOT
  |
  | ROOT
  V
d1_492 ------------------+
  |                      |
  | elaboration          | narration
  V                      V
d1_493 <---[result]-- d1_494
```

You can visualise the results with the *attelo report* (see Evaluation with attelo report) and attelo graph commands

Learning

In what follows,

- A refers to the attachment task: given an edu pair, is there a link between the two edus in any direction?
- D refers to the direction task: given an edu pair with a link between them, is the link from the textually earlier edu to the later on or vice-versa?
- L refers to the labelling task: given a directed linked edu pair, what is the label on edges between them?
- We use a '.' character to denote the grouping of the tasks into models, so for example, an 'AD.L' scheme is one in which we use one model for predicting attachement and directions together; and a separate model for predicting labels

AD.L scheme

In the AD.L scheme, we learn

- a binary attachment/direction model on all edu pairs — (e1, e2) and (e2, e1) are considered to be different pairs here
- a multiclass label model on only the edu pairs that are have an edge between them in the training data

See decoding for details on how these models are used on decoding time

Probabilities

In the general case both attachment and labelling scores are probabilities, and so the resulting score is also a probability; however, this is not always appropriate for all classifiers.

For example, see this blog post on the implications of using a hinge loss function as opposed to the proper loss. If you are using a non-probability-based learner, you should also set *–non-prob-scores* to false on decoding time

Sometimes classifiers may not naturally support probabilities but can provide conversion mechanisms to compute them from scores. These methods may come with various downsides (eg. be expensive to compute, and more worryingly, inconsistent with the scores), so it may be best to stick with non-prob decoding for them too. See the note in the scikit manual for details.

Developers' note: if you are developing classifiers for attelo, and your classifier does not return probabilties, it should implement *decision_function* instead

Decoding

Joint decoding mode (AD.L and ADL)

Joint decoding mode works with both the AD.L and the ADL schemes (latter is not yet implemented at the time of this writing 2015-02-16).

In the AD.L scheme, we query the attachment model for an attachment probability and the relation labelling model for its best labelling probability. We then multiply these into a single probability score for the decoder.

In the ADL scheme (ie. with only one model that does everything), we merely retrieve the highest probability score for each given instance.

Note that joint decoding mode cannot be used with models that cannot supply probabilities (for example, the perceptron). Post-label mode must be used instead. (See learning for details)

Post-label decoding mode (AD.L and ADL)

In post-label mode we retrieve just the probability of attachment (from the AD model in the AD.L case, and *1-P(UNRELATED)* in the ADL case) and feed this to the decoder (along with a dummy *UNKNOWN* label).

For each edge in the decoder output, we then retrieve the best label possible from the labeling model (or the best non-UNRELATED label in the ADL case) and apply that to the decoder outputs

Evaluation with attelo report

The `attelo report` command generates a set of evaluation reports by comparing `attelo decode` results against a gold standard. So far it creates:

- global precision/recall reports
- confusion matrices

There are two ways to use attelo report. You have it report scores on a single predictions file (produced by *attelo decode*); or you can have it report on a full set of predictions generated by a harness over multiple folds.

Mode A: predictions file

For one-off tests on *attelo decode* results, use the predictions file mode (*–prediction* <*FILE*>).

(NB: if your test was on a particular fold of the data you can also supply the *–fold* and *–fold-file* arguments to slice the data)

Mode B: harness mode

Harness-level reporting is only available programmatically via the attelo API

CHAPTER 2

Tutorial

Note: if you have downloaded the attelo source code, the tutorial is available as iPython notebooks in the doc directory

Datapacks and multipacks

Attelo reads its input files into "datapacks". Generally speaking, we have one datapack per document, so when reading a corpus in, we would be reading multiple datapacks (we read a multipack, ie. a dictionary of datapacks, or perhaps a fancier structure in future attelo versions)

```
from __future__ import print_function

from os import path as fp
from attelo.io import (load_multipack)

CORPUS_DIR = 'example-corpus'
PREFIX = fp.join(CORPUS_DIR, 'tiny')

# load the data into a multipack
mpack = load_multipack(PREFIX + '.edus',
                       PREFIX + '.pairings',
                       PREFIX + '.features.sparse',
                       PREFIX + '.features.sparse.vocab',
                       verbose=True)
```

```
Reading edus and pairings... done [0 ms]
Reading features... done [2 ms]
Build data packs... done [0 ms]
```

As we can see below, multipacks are dictionaries from document names to dpacks.

```
for dname, dpack in mpack.items():
    about = ("Doc: {name} |"
             " edus: {edus}, pairs: {pairs},"
             " features: {feats}")
    print(about.format(name=dname,
                       edus=len(dpack.edus),
                       pairs=len(dpack),
                       feats=dpack.data.shape))
```

```
Doc: d2 | edus: 4, pairs: 9, features: (9, 7)
Doc: d3 | edus: 3, pairs: 4, features: (4, 7)
Doc: d1 | edus: 4, pairs: 9, features: (9, 7)
```

Datapacks store everything we know about a document:

- edus: edus and their and their metadata
- pairings: factors to learn on
- data: feature array
- target: predicted label for each instance

```
dpack = mpack.values()[0] # pick an arbitrary pack
print("LABELS ({num}): {lbls}".format(num=len(dpack.labels),
                                      lbls=", ".join(dpack.labels)))
print()
# note that attelo will by convention insert __UNK__ into the list of
# labels, at position 0.  It also requires that UNRELATED and ROOT be
# in the list of available labels

for edu in dpack.edus[:3]:
    print(edu)
print("...\n")

for i, (edu1, edu2) in enumerate(dpack.pairings[:3]):
    lnum = dpack.target[i]
    lbl = dpack.get_label(lnum)
    feats = dpack.data[i,:].toarray()[0]
    print('PAIR', i, edu1.id, edu2.id, '\t|', lbl, '\t|', feats)
print("...\n")

for j, vocab in enumerate(dpack.vocab[:3]):
    print('FEATURE', j, vocab)
print("...\n")
```

```
LABELS (6): __UNK__, elaboration, narration, continuation, UNRELATED, ROOT

EDU ROOT: (0, 0) from None [None]
EDU d2_e2: (0, 27) from d2 [s3]     anybody want sheep for wood?
EDU d2_e3: (28, 40) from d2 [s4]    nope, not me
...

PAIR 0 ROOT d2_e2   | elaboration   | [ 0.  0.  0.  0.  0.  0.  0.]
PAIR 1 d2_e3 d2_e2  | narration     | [ 1.  1.  0.  0.  0.  0.  0.]
PAIR 2 d2_e4 d2_e2  | UNRELATED     | [ 2.  0.  1.  0.  0.  0.  0.]
...

FEATURE 0 sentence_id_EDU2=1
FEATURE 1 offset_diff_div3=0
FEATURE 2 num_tokens_EDU2=19
...
```

There are a couple of datapack variants to be aware of:

- *weighted* datapacks are parsed or partially parsed datapacks. They have a `graph` entry. We will explore weighted datapacks in the parser tutorial.
- *stacked* datapacks: are formed by combining datapacks from different documents into one. Some parts of the attelo API (namely scoring and reporting) work with stacked datapacks. In the future (now: 2015-05-06), they may evolve to deal with multipacks, in which case the notion of stack datapacks may dissapear

Conclusion

This concludes our tour of attelo datapacks. In other tutorials we will explore some of the uses of datapacks, namely as the input/output of our parsers.

Parsers

An attelo parser converts "documents" (here: EDUs with some metadata) into graphs (with EDUs as nodes and relation labels between them). In API terms, a parser is something that enriches datapacks, progressively adding or stripping away information until we get a full graph.

Parsers follow the scikit-learn estimator and transformer conventions, ie. with a `fit` function to learn some model from training data and a `transform` function to convert (in our case) datapacks to enriched datapacks.

Preliminaries

To begin our exploration of attelo parsers, let's load up a tiny multipack of sample data.

```
from __future__ import print_function

from os import path as fp
from attelo.io import (load_multipack)

CORPUS_DIR = 'example-corpus'
PREFIX = fp.join(CORPUS_DIR, 'tiny')

# load the data into a multipack
mpack = load_multipack(PREFIX + '.edus',
                       PREFIX + '.pairings',
                       PREFIX + '.features.sparse',
                       PREFIX + '.features.sparse.vocab',
                       verbose=True)
```

```
Reading edus and pairings... done [1 ms]
Reading features... done [1 ms]
Build data packs... done [0 ms]
```

We'll set aside one of the datapacks to test with, leaving the other two for training. We do this by hand for this simple example, but you may prefer to use the helper functions in attelo.fold when working with real data

```
test_dpack = mpack.values()[0]
train_mpack = {k: mpack[k] for k in mpack.keys()[1:]}

print('multipack entries:', len(mpack))
print('train entries:', len(train_mpack))
```

```
multipack entries: 3
train entries: 2
```

Trying a parser out 1 (attach)

Now that we have our training and test data, we can try feeding them to a simple parser. Before doing this, we'll take a quick detour to define a helper function to visualise our parse results.

```
def print_results(dpack):
    'summarise parser results'
    for i, (edu1, edu2) in enumerate(dpack.pairings):
        wanted = dpack.get_label(dpack.target[i])
        got = dpack.get_label(dpack.graph.prediction[i])
        print(i, edu1.id, edu2.id, '\t|', got, '\twanted:', wanted)
```

As for parsing, we'll start with the attachment pipeline. It combines a learner with a decoder

```
from attelo.decoding.baseline import (LastBaseline)
from attelo.learning import (SklearnAttachClassifier)
from attelo.parser.attach import (AttachPipeline)
from sklearn.linear_model import (LogisticRegression)

learner = SklearnAttachClassifier(LogisticRegression())
decoder = LastBaseline()
parser1 = AttachPipeline(learner=learner,
                         decoder=decoder)

# train the parser
train_dpacks = train_mpack.values()
train_targets = [x.target for x in train_dpacks]
parser1.fit(train_dpacks, train_targets)

# now run on a test pack
dpack = parser1.transform(test_dpack)
print_results(dpack)
```

```
0 ROOT d2_e2        | __UNK__       wanted: elaboration
1 d2_e3 d2_e2       | UNRELATED     wanted: narration
2 d2_e4 d2_e2       | UNRELATED     wanted: UNRELATED
3 ROOT d2_e3        | UNRELATED     wanted: continuation
4 d2_e2 d2_e3       | __UNK__       wanted: narration
5 d2_e4 d2_e3       | UNRELATED     wanted: narration
6 ROOT d2_e4        | UNRELATED     wanted: UNRELATED
7 d2_e3 d2_e4       | __UNK__       wanted: elaboration
8 d2_e2 d2_e4       | UNRELATED     wanted: UNRELATED
```

Trying a parser out 2 (label)

In the output above, our predictions for every edge are either `__UNK__` or `UNRELATED`. The attachment pipeline only predicts if edges will be attached or not. What we need is to be able to predict their labels.

```
from attelo.learning import (SklearnLabelClassifier)
from attelo.parser.label import (SimpleLabeller)
from sklearn.linear_model import (LogisticRegression)

learner = SklearnLabelClassifier(LogisticRegression())
parser2 = SimpleLabeller(learner=learner)

# train the parser
parser2.fit(train_dpacks, train_targets)

# now run on a test pack
dpack = parser2.transform(test_dpack)
print_results(dpack)
```

```
0 ROOT d2_e2         | elaboration   wanted: elaboration
1 d2_e3 d2_e2        | elaboration   wanted: narration
2 d2_e4 d2_e2        | narration     wanted: UNRELATED
3 ROOT d2_e3         | elaboration   wanted: continuation
4 d2_e2 d2_e3        | elaboration   wanted: narration
5 d2_e4 d2_e3        | narration     wanted: narration
6 ROOT d2_e4         | elaboration   wanted: UNRELATED
7 d2_e3 d2_e4        | elaboration   wanted: elaboration
8 d2_e2 d2_e4        | narration     wanted: UNRELATED
```

That doesn't quite look right. Now we have labels, but none of our edges are `UNRELATED`. But this is because the simple labeller will apply labels on all unknown edges. What we need is to be able to combine the attach and label parsers in a parsing pipeline

Parsing pipeline

A parsing pipeline is a parser that combines other parsers in sequence. For purposes of learning/fitting, the individual steps can be thought of as being run in parallel (in practice, they are fitted in sequnce). For transforming though, they are run in order. A pipeline thus refines a datapack over the course of multiple parsers.

```
from attelo.parser.pipeline import (Pipeline)

# this is actually attelo.parser.full.PostlabelPipeline
parser3 = Pipeline(steps=[('attach', parser1),
                          ('label', parser2)])

parser3.fit(train_dpacks, train_targets)
dpack = parser3.transform(test_dpack)
print_results(dpack)
```

```
0 ROOT d2_e2         | elaboration   wanted: elaboration
1 d2_e3 d2_e2        | UNRELATED     wanted: narration
2 d2_e4 d2_e2        | UNRELATED     wanted: UNRELATED
3 ROOT d2_e3         | UNRELATED     wanted: continuation
4 d2_e2 d2_e3        | elaboration   wanted: narration
5 d2_e4 d2_e3        | UNRELATED     wanted: narration
6 ROOT d2_e4         | UNRELATED     wanted: UNRELATED
7 d2_e3 d2_e4        | elaboration   wanted: elaboration
8 d2_e2 d2_e4        | UNRELATED     wanted: UNRELATED
```

Conclusion (for now)

We have now seen some basic attelo parsers, how they use the scikit-learn fit/transform idiom, and we can combine them with pipelines. In future tutorials we'll break some of the parsers down into their constituent parts (notice the attach pipeline is itself a pipeline) and explore the process of writing parsers of our own.

Parsers (part 2)

In the previous tutorial, we saw a couple of basic parsers, and also introduced the notion of a pipeline parser. It turns out that some of the parsers we introduced and had taken for granted are themselves pipelines. In this tutorial we will break these pipelines down and explore some of finer grained tasks that a parser can do.

Preliminaries

We begin with the same multipacks and the same breakdown into a training and test set

```
from __future__ import print_function

from os import path as fp
from attelo.io import (load_multipack)

CORPUS_DIR = 'example-corpus'
PREFIX = fp.join(CORPUS_DIR, 'tiny')

# load the data into a multipack
mpack = load_multipack(PREFIX + '.edus',
                       PREFIX + '.pairings',
                       PREFIX + '.features.sparse',
                       PREFIX + '.features.sparse.vocab',
                       verbose=True)

test_dpack = mpack.values()[0]
train_mpack = {k: mpack[k] for k in mpack.keys()[1:]}
train_dpacks = train_mpack.values()
train_targets = [x.target for x in train_dpacks]

def print_results(dpack):
    'summarise parser results'
    for i, (edu1, edu2) in enumerate(dpack.pairings):
        wanted = dpack.get_label(dpack.target[i])
        got = dpack.get_label(dpack.graph.prediction[i])
        print(i, edu1.id, edu2.id, '\t|', got, '\twanted:', wanted)
```

```
Reading edus and pairings... done [1 ms]
Reading features... done [1 ms]
Build data packs... done [0 ms]
```

Breaking a parser down (attach)

If we examine the source code for the attach pipeline, we can see that it is in fact a two step pipeline combining the attach classifier wrapper and a decoder. So let's see what happens when we run the attach classifier by itself.

```
import numpy as np
from attelo.learning import (SklearnAttachClassifier)
from attelo.parser.attach import (AttachClassifierWrapper)
from sklearn.linear_model import (LogisticRegression)

def print_results_verbose(dpack):
    """Print detailed parse results"""
    for i, (edu1, edu2) in enumerate(dpack.pairings):
        attach = "{:.2f}".format(dpack.graph.attach[i])
        label = np.around(dpack.graph.label[i,:], decimals=2)
        got = dpack.get_label(dpack.graph.prediction[i])
        print(i, edu1.id, edu2.id, '\t|', attach, label, got)

learner = SklearnAttachClassifier(LogisticRegression())
parser1a = AttachClassifierWrapper(learner)
parser1a.fit(train_dpacks, train_targets)
```

```
dpack = parser1a.transform(test_dpack)
print_results_verbose(dpack)
```

```
0 ROOT d2_e2         | 0.44 [ 1.  1.  1.  1.  1.  1.] __UNK__
1 d2_e3 d2_e2        | 0.43 [ 1.  1.  1.  1.  1.  1.] __UNK__
2 d2_e4 d2_e2        | 0.43 [ 1.  1.  1.  1.  1.  1.] __UNK__
3 ROOT d2_e3         | 0.44 [ 1.  1.  1.  1.  1.  1.] __UNK__
4 d2_e2 d2_e3        | 0.97 [ 1.  1.  1.  1.  1.  1.] __UNK__
5 d2_e4 d2_e3        | 0.39 [ 1.  1.  1.  1.  1.  1.] __UNK__
6 ROOT d2_e4         | 0.01 [ 1.  1.  1.  1.  1.  1.] __UNK__
7 d2_e3 d2_e4        | 0.42 [ 1.  1.  1.  1.  1.  1.] __UNK__
8 d2_e2 d2_e4        | 0.39 [ 1.  1.  1.  1.  1.  1.] __UNK__
```

Parsers and weighted datapacks

In the output above, we have dug a little bit deeper into our datapacks. Recall above that a parser translates datapacks to datapacks. The output of a parser is always a *weighted datapack*., ie. a datapack whose 'graph' attribute is set to a record containing

- attachment weights
- label weights
- predictions (like target values)

So called "standalone" parsers will take an unweighted datapack (`graph == None`) and produce a weighted datapack with predictions set. But some parsers tend to be more useful as part of a pipeline:

- the attach classfier wrapper fills the attachment weights
- likewise the label classifier wrapper assigns label weights
- a decoder assigns predictions from weights

We see the first case in the above output. Notice that the attachments have been set to values from a model, but the label weights and predictions are assigned default values.

NB: all parsers should do "something sensible" in the face of all inputs. This typically consists of assuming the default weight of 1.0 for unweighted datapacks.

Decoders

Having now transformed a datapack with the attach classifier wrapper, let's now pass its results to a decoder. In fact, let's try a couple of different decoders and compare the output.

```
from attelo.decoding.baseline import (LocalBaseline)

decoder = LocalBaseline(threshold=0.4)
dpack2 = decoder.transform(dpack)
print_results_verbose(dpack2)
```

```
0 ROOT d2_e2         | 0.44 [ 1.  1.  1.  1.  1.  1.] __UNK__
1 d2_e3 d2_e2        | 0.43 [ 1.  1.  1.  1.  1.  1.] __UNK__
2 d2_e4 d2_e2        | 0.43 [ 1.  1.  1.  1.  1.  1.] __UNK__
3 ROOT d2_e3         | 0.44 [ 1.  1.  1.  1.  1.  1.] __UNK__
4 d2_e2 d2_e3        | 0.97 [ 1.  1.  1.  1.  1.  1.] __UNK__
5 d2_e4 d2_e3        | 0.39 [ 1.  1.  1.  1.  1.  1.] UNRELATED
6 ROOT d2_e4         | 0.01 [ 1.  1.  1.  1.  1.  1.] UNRELATED
```

```
7 d2_e3 d2_e4        | 0.42 [ 1.  1.  1.  1.  1.  1.] __UNK__
8 d2_e2 d2_e4        | 0.39 [ 1.  1.  1.  1.  1.  1.] UNRELATED
```

The result above is what we get if we run a decoder on the output of the attach classifier wrapper. This is in fact, the the same thing as running the attachment pipeline. We can define a similar pipeline below.

```
from attelo.parser.pipeline import (Pipeline)

# this is basically attelo.parser.attach.AttachPipeline
parser1 = Pipeline(steps=[('attach weights', parser1a),
                          ('decoder', decoder)])
parser1.fit(train_dpacks, train_targets)
print_results_verbose(parser1.transform(test_dpack))
```

```
0 ROOT d2_e2         | 0.44 [ 1.  1.  1.  1.  1.  1.] __UNK__
1 d2_e3 d2_e2        | 0.43 [ 1.  1.  1.  1.  1.  1.] UNRELATED
2 d2_e4 d2_e2        | 0.43 [ 1.  1.  1.  1.  1.  1.] UNRELATED
3 ROOT d2_e3         | 0.44 [ 1.  1.  1.  1.  1.  1.] UNRELATED
4 d2_e2 d2_e3        | 0.97 [ 1.  1.  1.  1.  1.  1.] __UNK__
5 d2_e4 d2_e3        | 0.39 [ 1.  1.  1.  1.  1.  1.] UNRELATED
6 ROOT d2_e4         | 0.01 [ 1.  1.  1.  1.  1.  1.] UNRELATED
7 d2_e3 d2_e4        | 0.42 [ 1.  1.  1.  1.  1.  1.] __UNK__
8 d2_e2 d2_e4        | 0.39 [ 1.  1.  1.  1.  1.  1.] UNRELATED
```

Mixing and matching

Being able to break parsing down to this level of granularity lets us experiment with parsing techniques by composing different parsing substeps in different ways. For example, below, we write two slightly different pipelines, one which sets labels separately from decoding, and one which combines attach and label scores before handing them off to a decoder.

```
from attelo.learning.local import (SklearnLabelClassifier)
from attelo.parser.label import (LabelClassifierWrapper,
                                 SimpleLabeller)
from attelo.parser.full import (AttachTimesBestLabel)

learner_l = SklearnLabelClassifier(LogisticRegression())

print("Post-labelling")
print("--------------")
parser = Pipeline(steps=[('attach weights', parser1a),
                         ('decoder', decoder),
                         ('labels', SimpleLabeller(learner_l))])
parser.fit(train_dpacks, train_targets)
print_results_verbose(parser.transform(test_dpack))

print()
print("Joint")
print("-----")
parser = Pipeline(steps=[('attach weights', parser1a),
                         ('label weights', LabelClassifierWrapper(learner_l)),
                         ('attach times label', AttachTimesBestLabel()),
                         ('decoder', decoder)])
parser.fit(train_dpacks, train_targets)
print_results_verbose(parser.transform(test_dpack))
```

```
Post-labelling
--------------
0 ROOT d2_e2         | 0.44 [ 0.    0.45  0.28  0.28  0.    0.  ] elaboration
1 d2_e3 d2_e2        | 0.43 [ 0.    0.4   0.34  0.25  0.    0.  ] elaboration
2 d2_e4 d2_e2        | 0.43 [ 0.    0.3   0.53  0.17  0.    0.  ] narration
3 ROOT d2_e3         | 0.44 [ 0.    0.45  0.28  0.28  0.    0.  ] elaboration
4 d2_e2 d2_e3        | 0.97 [ 0.    0.52  0.03  0.45  0.    0.  ] elaboration
5 d2_e4 d2_e3        | 0.39 [ 0.    0.37  0.43  0.2   0.    0.  ] UNRELATED
6 ROOT d2_e4         | 0.01 [ 0.    0.45  0.28  0.28  0.    0.  ] UNRELATED
7 d2_e3 d2_e4        | 0.42 [ 0.    0.41  0.35  0.24  0.    0.  ] elaboration
8 d2_e2 d2_e4        | 0.39 [ 0.    0.37  0.43  0.2   0.    0.  ] UNRELATED

Joint
-----
0 ROOT d2_e2         | 0.19 [ 0.    0.45  0.28  0.28  0.    0.  ] UNRELATED
1 d2_e3 d2_e2        | 0.17 [ 0.    0.4   0.34  0.25  0.    0.  ] UNRELATED
2 d2_e4 d2_e2        | 0.23 [ 0.    0.3   0.53  0.17  0.    0.  ] UNRELATED
3 ROOT d2_e3         | 0.19 [ 0.    0.45  0.28  0.28  0.    0.  ] UNRELATED
4 d2_e2 d2_e3        | 0.50 [ 0.    0.52  0.03  0.45  0.    0.  ] elaboration
5 d2_e4 d2_e3        | 0.17 [ 0.    0.37  0.43  0.2   0.    0.  ] UNRELATED
6 ROOT d2_e4         | 0.00 [ 0.    0.45  0.28  0.28  0.    0.  ] UNRELATED
7 d2_e3 d2_e4        | 0.17 [ 0.    0.41  0.35  0.24  0.    0.  ] UNRELATED
8 d2_e2 d2_e4        | 0.17 [ 0.    0.37  0.43  0.2   0.    0.  ] UNRELATED
```

Conclusion

Thinking of parsers as transformers from weighted datapacks to weighted datapacks should allow for some interesting parsing experiments, parsers that

- divide the work using different strategies on different subtypes of input (eg. intra vs intersentential links), or
- work in multiple stages, maybe modifying past decisions along the way, or
- influence future parsing stages by tweaking the weights they might see, or
- prune out undesirable edges (by setting their weights to zero), or
- apply some global constraint satisfaction algorithm across the possible weights

With a notion of a parsing pipeline, you should also be able to build parsers that combine different experiments that you want to try simultaneously

Harnesses

In the previous tutorials, we introduced the notion of parsers, broke them down into their constituent parts, and very briefly touched upon the idea of mixing and matching parsers to form more interesting combinations.

If you find yourself in a situation where you have several parsing ideas that you would like to explore, you may find it helpful to create an experimental harness. A harness can be useful for

1. [reliability, convenience] bundling all the evaluation steps into a single easy-to-remember command (this eliminates the risk of omitting a crucial step)
2. [convenience] consistently generating an detailed report including confusion matrices, discriminating features, some visual samples of the output
3. [performance] caching shareable results to save evaluation time (both horizontally, for example, across parsers that can share models, and vertically, perhaps across different versions of a decoder but using the same model)

4. [performance] managing concurrency and distributed evaluation, which may be attractive if you have access to a compute cluster

The *attelo.harness* provides a basic framework for defining such harnesses. You would need to implement the *Harness* class, specifying

- the data to read
- a list of parsers to run (wrapped in *attelo.harness.config.EvaluationConfig*)
- some functions for assigning filenames to intermediary results
- and a variety of reporting options (for example, which evaluations you would like to generate extra reports on)

Have a look at the example harness to get started, and perhaps also the irit-rst-dt to see how this might be used in a real experimental setting.

Caching

Attelo's caching mechanism uses the *cache* keyword argument in *attelo.parser.Parser.fit* (cache is an attelo-ism, and is not standard to the scikit estimator/transformer idiom). The idea is for parsers to accept a dictionary from simple cache keywords (eg. 'attach') to paths. Parsers could interact with the cache in different ways. In the simplest case, they might look for a particular keyword to determine if there is a cache entry that it could load (or should save to). Alternatively, if multiple parsers are composed of parsers that they have in commone, they can avoid repeating work on their constituent parts by simply passing their cache dictionaries down (NB: it is up to parser authors to ensure that cache keys do not conflict; parsers should document their cache keys in the API)

The *attelo.harness.Harness.model_paths* function implemented by your harness should return exactly such a dictionary, as we might see in the example below

```
def model_paths(self, rconf, fold):
    if fold is None:
        parent_dir = self.combined_dir_path()
    else:
        parent_dir = self.fold_dir_path(fold)

    def _eval_model_path(mtype):
        "Model for a given loop/eval config and fold"
        bname = self._model_basename(rconf, mtype, 'model')
        return fp.join(parent_dir, bname)

    return {'attach': _eval_model_path("attach"),
            'label': _eval_model_path("label")}
```

Cluster mode: parallel and distributed

The attelo harness provides some crude support on a cluster:

- decoding is split into one decoding job per document/grouping; as each parser is learned [fit] (sequentially), the harness adds its decoding jobs [transform] to a pool of jobs in progress.
- each fold is self-contained, and can be run concurrently. If you are on a cluster with multiple machines reading from a shared filesystem, you can farm the folds out to separate machines (nb: the harness itself does not do this for you, so you would need to write eg. a shell script that does this parceling out of folds, but it can be broken down in a way that facilitates this usage, ie. with "initialise", "run folds 1 and 2", "run folds 3 and 4", ... "gather the results" as discrete steps)

CHAPTER 3

attelo API

attelo package

Attelo is a statistical discourse parser. The API provides

- decoders which you should be able to call in a standalone way
- machine learning infrastructure wrapping around a library like sci-kit learn
- support for building experimental harnesses around the parser

Subpackages

attelo.decoding package

Decoding in attelo consists in building discourse graphs from a set of attachment/labelling predictions.

Submodules

attelo.decoding.astar module

module for building discourse graphs from probability distribution and respecting some constraints, using Astar heuristics based search and variants (beam, b&b)

TODO: unlabelled evaluation seems to bug on RF decoding (relation is of type orange.value -> go see in decoding.py)

class attelo.decoding.astar.**AstarArgs**
Bases: *attelo.decoding.astar.AstarArgs*

Configuration options for the A* decoder

Parameters

- **heuristics** (*Heuristic*) – an a* heuristic funtion (estimate the cost of what has not been explored yet)
- **use_prob** (*bool*) – indicates if previous scores are probabilities in [0,1] (to be mapped to -log) or arbitrary scores (untouched)
- **beam** (*int or None*) – size of the beam-search (if None: vanilla astar)
- **rfc** (RfcConstraint) – what sort of right frontier constraint to apply

class attelo.decoding.astar.**AstarDecoder**(*astar_args*)
Bases: *attelo.decoding.interface.Decoder*

wrapper for astar decoder to be used by processing pipeline returns the best structure

decode(*dpack*)

class attelo.decoding.astar.**DiscData**(*parent=None*, *accessible=None*, *tolink=None*)
Bases: object

Natural reading order decoding: incremental building of tree in order of text (one edu at a time)

Basic discourse data for a state: chosen links between edus at that stage + right-frontier state. To save space, only new links are stored. the complete solution will be built with backpointers via the parent field

RF: right frontier, = admissible attachment point of current discourse unit

Parameters

- **parent** – parent state (previous decision)
- **link** (*(string, string, string)*) – current decision (a triplet: target edu, source edu, relation)
- **tolink** (*[string]*) – remaining unattached discourse units

accessible()
return the list of edus that are on the right frontier

Return type [string]

final()
return *True* if there are no more links to be made

last_link()
return the link that was made to get to this state, if any

link(*to_edu*, *from_edu*, *relation*, *rfc=<RfcConstraint.full: 2>*)
rfc = "full": use the distinction coord/subord rfc = "simple": consider everything as subord rfc = "none" no constraint on attachment

tobedone()
return the list of edus to be linked

Return type [string]

class attelo.decoding.astar.**DiscourseBeamSearch**(*heuristic=<function <lambda>>*, *shared=None*, *queue_size=10*)
Bases: *attelo.decoding.astar.DiscourseSearch*, *attelo.optimisation.astar.BeamSearch*

class attelo.decoding.astar.**DiscourseSearch**(*heuristic=<function <lambda>>*, *shared=None*, *queue_size=None*)
Bases: *attelo.optimisation.astar.Search*

subtype of astar search for discourse: should be the same for every astar decoder, provided the discourse state is a subclass of DiscourseState

recover solution should be as is, provided a state has at least the following info: - parent: parent state - _link: the actual prediction made at this stage (1 state = 1 relation = (du1, du2, relation)

new_state(*data*)

recover_solution(*endstate*)
follow back pointers to collect list of chosen relations on edus.

class attelo.decoding.astar.**DiscourseState**(*data*, *heuristics*, *shared*)
Bases: *attelo.optimisation.astar.State*

Natural reading order decoding: incremental building of tree in order of text (one edu at a time)

instance of discourse graph with probability for each attachement+relation on a subset of edges.

implements the State interface to be used by Search

strategy: at each step of exploration choose a relation between two edus related by probability distribution, reading order a.k.a NRO "natural reading order", cf Bramsen et al., 2006. in temporal processing.

'data' is set of instantiated relations (typically nothing at the beginning, but could be started with a few chosen relations)

'shared' points to shared data between states (here proba distribution between considered pairs of edus at least, but also can include precomputed info for heuristics)

h_average()
return the average probability possible when n nodes still need to be attached assuming the best overall prob in the distrib

h_best()
return the best probability possible when n nodes still need to be attached assuming the best overall prob in the distrib

h_best_overall()
return the best probability possible when n nodes still need to be attached assuming the best overall prob in the distrib

h_zero()
always 0

is_solution()

next_states()
must return a state and a cost TODO: adapt to disc parse, according to choice made for data -> especially update to RFC

proba(*edu_pair*)
return the label and probability that an edu pair are attached, or *("no", None)* if we don't have a prediction for the pair

Return type (string, float or None)

shared()
information shared between states

strategy()
full or not, if the RFC is applied to labelled edu pairs

class attelo.decoding.astar.**Heuristic**
Bases: enum.Enum

Heuristic cost to guide A* search with

- zero: see DiscourseState.h_zero
- max: see DiscourseState.h_best_overall
- best: see DiscourseState.h_best
- average: see DiscourseState.h_average

average = <Heuristic.average: 3>

best = <Heuristic.best: 2>

max = <Heuristic.max: 1>

zero = <Heuristic.zero: 0>

class attelo.decoding.astar.**RfcConstraint**
Bases: enum.Enum

What sort of right frontier constraint to apply during decoding:

- simple: every relation is treated as subordinating
- full: (falls back to simple in case of unlabelled prediction)

full = <RfcConstraint.full: 2>

none = <RfcConstraint.none: 3>

simple = <RfcConstraint.simple: 1>

class attelo.decoding.astar.**TwoStageNRO**
Bases: *attelo.decoding.astar.DiscourseState*

similar as above with different handling of inter-sentence and intra-sentence relations

next_states()
must return a state and a cost

same_sentence(*edu1*, *edu2*)
not implemented: will always return False TODO: this should go in preprocessing before launching astar ?? would it be easier to have access to all edu pair features ?? (certainly for that one)

class attelo.decoding.astar.**TwoStageNROData**(*parent=None*, *accessible=None*, *tolink=None*)
Bases: *attelo.decoding.astar.DiscData*

similar as above with different handling of inter-sentence and intra-sentence relations

accessible is list of starting edus (only one for now)

accessible()
wip:

link(*to_edu*, *from_edu*, *relation*)
WIP

update_mode()
switch between intra/inter-sentential parsing mode

attelo.decoding.astar.**preprocess_heuristics**(*cands*)

precompute a set of useful information used by heuristics, such as

- best probability
- table of best probability when attaching a node, indexed on that node

format of cands is format given in main decoder: a list of (arg1,arg2,proba,best_relation)

attelo.decoding.baseline module

Baseline decoders

class attelo.decoding.baseline.**LastBaseline**
Bases: *attelo.decoding.interface.Decoder*

attach to last, always

decode(*dpack*, *nonfixed_pairs=None*)

class attelo.decoding.baseline.**LocalBaseline**(*threshold*, *use_prob=True*)
Bases: *attelo.decoding.interface.Decoder*

just attach locally if prob is > threshold

decode(*dpack*, *nonfixed_pairs=None*)

attelo.decoding.greedy module

Implementation of the locally greedy approach similar with DuVerle & Predinger (2009, 2010) (but adapted for SDRT, where the notion of adjacency includes embedded segments)

July 2012

@author: stergos

class attelo.decoding.greedy.**LocallyGreedy**
Bases: *attelo.decoding.interface.Decoder*

The locally greedy decoder

decode(*dpack*)

class attelo.decoding.greedy.**LocallyGreedyState**(*instances*)
Bases: object

the mutable parts of the locally greedy algorithm

decode()
Run the decoder

:rtype [(EDU, EDU, string)]

attelo.decoding.greedy.**are_strictly_adjacent**(*one*, *two*, *edus*)
returns True in the following cases

```
[one] [two]
[two] [one]
```

in the rest of the cases (when there is an edu between one and two) it returns False

attelo.decoding.greedy.**get_neighbours**(*edus*)
Return a mapping from each EDU to its neighbours

Return type Dict Edu [Edu]

attelo.decoding.greedy.**is_embedded**(*one*, *two*)
returns True when one is embedded in two, that is

```
[two ... [one] ... ]
```

returns False in all other cases

attelo.decoding.interface module

Common interface that all decoders must implement

class attelo.decoding.interface.**Decoder**
Bases: *attelo.parser.interface.Parser*

A decoder is a function which given a probability distribution (see below) and some control parameters, returns a sequence of predictions.

Most decoders only really return one prediction in practice, but some, like the A* decoder might have able to return a ranked sequence of the "N best" predictions it can find

We have a few informal types to consider here:

- a **link** (*(string, string, string)*) represents a link between a pair of EDUs. The first two items are their identifiers, and the third is the link label
- a **candidate link** (or candidate, to be short, *(EDU, EDU, float, string)*) is a link with a probability attached
- a **prediction** is morally a set (in practice a list) of links
- a **distribution** is morally a set of proposed links

Note that a decoder could also be seen/used as a sort of crude parser (with a fit function is a no-op). You'll likely want to prefix it with a parser that extracts weights from datapacks lest you work with the somewhat unformative 1.0s everywhere.

decode(*dpack*)
Return the N-best predictions in the form of a datapack per prediction.

fit(*dpacks*, *targets*, *nonfixed_pairs=None*, *cache=None*)

transform(*dpack*, *nonfixed_pairs=None*)

attelo.decoding.local module

Local decoders make decisions for each edge independently.

class attelo.decoding.local.**AsManyDecoder**
Bases: *attelo.decoding.interface.Decoder*

Greedy decoder that picks as many edges as there are real EDUs.

The output structure is a graph that has the same number of edges as a spanning tree over the EDUs. It can be non-connex, contain cycles and re-entrancies.

decode(*dpack*)
Return the set of top N edges

class attelo.decoding.local.**BestIncomingDecoder**
Bases: *attelo.decoding.interface.Decoder*

Greedy decoder that picks the best incoming edge for each EDU.

The output structure is a graph that contains exactly one incoming edge for each EDU, thus it has the same number of edges as a spanning tree over the EDUs. It can be non-connex or contain cycles, but no re-entrancy.

decode(*dpack*)
Return the best incoming edge for each EDU

attelo.decoding.mst module

Created on Jun 27, 2012

@author: stergos, jrmyp

class attelo.decoding.mst.**MsdagDecoder**(*root_strategy*, *use_prob=True*)
Bases: *attelo.decoding.mst.MstDecoder*

Attach according to MSDAG (subgraph of original)

decode(*dpack*, *nonfixed_pairs=None*)

class attelo.decoding.mst.**MstDecoder**(*root_strategy*, *use_prob=True*)
Bases: *attelo.decoding.interface.Decoder*

Attach in such a way that the resulting subgraph is a maximum spanning tree of the original

decode(*dpack*, *nonfixed_pairs=None*)

class attelo.decoding.mst.**MstRootStrategy**
Bases: *attelo.util.ArgparserEnum*

How we declare the MST root node

fake_root = <MstRootStrategy.fake_root: 1>

leftmost = <MstRootStrategy.leftmost: 2>

attelo.decoding.util module

Utility classes functions shared by decoders

exception attelo.decoding.util.**DecoderException**
Bases: exceptions.Exception

Exceptions that arise during the decoding process

attelo.decoding.util.**cap_score**(*score*)
Cap a real-valued score between *MIN_SCORE* and *MAX_SCORE*.

The current default values for *MIN_SCORE* and *MAX_SCORE* follow the requirements from the decoders: * The MST decoder uses the depparse package whose MST implementation has a hardcoded minimum score of *-1e100* ; Feeding it lower weights crashes the algorithm. Combined scores can't reach the limit unless we have more than 1e10 nodes. * The Eisner decoder internally uses float64 scores.

Parameters **score** (*float*) – Original score.

Returns **bounded_score** – Score bounded to [MIN_SCORE, MAX_SCORE].

Return type float

attelo.decoding.util.**convert_prediction**(*dpack*, *triples*)
Populate a datapack prediction array from a list of triples

Parameters **prediction** (*[(string, string, string)]*) – List of EDU id, EDU id, label triples

Returns **dpack** – A copy of the original DataPack with predictions set

Return type *DataPack*

attelo.decoding.util.**get_prob_map**(*instances*)
Reformat a probability distribution as a dictionary from edu id pairs to a (relation, probability) tuples

:rtype dict (string, string) (string, float)

attelo.decoding.util.**get_sorted_edus**(*instances*)
Return a list of EDUs, using the following as sort key in order of

- starting position (earliest edu first)
- ending position (narrowest edu first)

Note that there may be EDU pairs with the same spans (particularly in case of annotation error). In case of ties, the order should be considered arbitrary

attelo.decoding.util.**prediction_to_triples**(*dpack*)

Returns

triples –

List of EDU id, EDU id, label triples omitting the unrelated triples

Return type prediction: [(string, string, string)]

attelo.decoding.util.**simple_candidates**(*dpack*)
Translate the links into a list of (EDU, EDU, float, string) quadruplets representing the attachment probability and the the best label for each EDU pair. This is often good enough for simplistic decoders

attelo.decoding.window module

A "pruning" decoder that pre-processes candidate edges and prunes them away if they are separated by more than a certain number of EDUs

class attelo.decoding.window.**WindowPruner**(*window*)
Bases: *attelo.decoding.interface.Decoder*

Notes

We assume that the datapack includes every EDU in its grouping.

If there are any gaps, the window will be a bit messed up

As decoders are parsers like any other, if you just want to apply this as preprocessing to a decoder, you could construct a mini pipeline consisting of this plus the decoder. Alternatively, if you already have a larger pipeline of which the decoder is already part, you can just insert this before the decoder.

decode(*dpack*)

attelo.harness package

attelo experimental harness helpers

The modules here are meant to help with building your own test harnesses around attelo. They provide opinionated support for experiment layout and interfacing with attelo

Submodules

attelo.harness.config module

Configuring the harness

class attelo.harness.config.**ClusterStage**
Bases: enum.Enum

What stage of cluster usage we are at

This is used when you want to distribute the evaluation across multiple nodes of a cluster.

The idea is that you would run the harness in separate stages:

•a single "start" stage, then

•in parallel * nodes running "main" stages for some folds * a node running a "combined_model" stage

•finally, a single "end" stage

combined_models = <ClusterStage.combined_models: 3>

end = <ClusterStage.end: 4>

main = <ClusterStage.main: 2>

start = <ClusterStage.start: 1>

class attelo.harness.config.**DataConfig**
Bases: *attelo.harness.config.DataConfig*

Data tables read during harness evaluation

This class may be folded into HarnessConfig eventually

class attelo.harness.config.**EvaluationConfig**
Bases: *attelo.harness.config.EvaluationConfig*

Combination of learners, decoders and decoder settings for an attelo evaluation

The settings can really be of type that has a 'key' field; but you should have a way of extracting at least a DecodingMode from it

Parameters

- **learner** (*Keyed learnercfg*) – Some sort of keyed learner configuration. This is usually of type *LearnerConfig* but there are cases where you have fancier objects in place
- **parser** (*Keyed (learnercfg -> Parser)*) – A (keyed) function that builds a parser from whatever learner configuration you used in *learner*
- **settings** (*Keyed (???)*) –

classmethod simple_key(*learner*, *decoder*)
generate a short unique name for a learner/decoder combo

class attelo.harness.config.**Keyed**
Bases: *attelo.harness.config.Keyed*

A keyed object is just any object that is attached with a short unique (mnemonic) identifier.

Keys often appear in filenames so it's best to avoid whitespace, fancy characters, and for portability reasons, anything non-ASCII.

class attelo.harness.config.**LearnerConfig**
Bases: *attelo.util.Team*

Combination of an attachment and a relation learner variant

class attelo.harness.config.**RuntimeConfig**
Bases: *attelo.harness.config.RuntimeConfig*

Harness runtime options.

These are mostly relevant to when using the harness on a cluster.

Parameters

- **mode** (*string ('resume' or 'jumpstart') or None*) –
  - jumpstart: copy model and fold files from a previous evaluation
  - resume: pick an evaluation up from where it left off
- **folds** (*[int] or None*) – Which folds to run the harness on. None to run on all folds
- **n_jobs** (*int (-1 or natural)*) – Number of parallel jobs to run (-1 for max cores). See joblib doc for details
- **stage** (*ClusterStage or None*) – Which evaluation stage to run

classmethod empty()
Empty configuration

attelo.harness.evaluate module

attelo.harness.example module

attelo.harness.graph module

attelo.harness.interface module

Basic interface that all test harnesses should respect

class attelo.harness.interface.**Harness**(*dataset*, *testset*)
Bases: object

Test harness configuration.

Among other things, this is about defining conventions for filepaths.

Notes

You should have a method that calls *load*. It should be invoked once before running the harness. A natural idiom may be to implement a single *run* function that does this.

combined_dir_path()
Return path to directory where combined/global models should be stored

This would be for all training data, ie. without paying attention to folds

Returns

Return type filepath

config_files
Files needed to reproduce the configuration behind a particular set of scores.

Will be copied into the provenance section of the report.

Some harnesses have parameter files that should be saved in case there is any need to reproduce results much futher into the future. Specifying them here gives you some extra insurance in case you neglect to put them under version control.

create_folds(*mpack*)
Generate the folds dictionary for the given multipack, optionally caching them to disk

In some harness configurations, it may make sense to have a fixed set of folds rather than generating them on the fly

Returns **fold_dict** – dictionary from document names to fold

Return type dict(string, int)

decode_output_path(*econf*, *fold*)
Return path to output graph for given fold and config

detailed_evaluations
Set of evaluations for which we would like detailed reporting

eval_dir
Directory to store evaluation results.

Basically anything that should be considered as important for long-term archiving and reproducibility

evaluations
List of evaluations to use on the training data

fold_dir_path(*fold*)
Return path to working directory for a given fold

Parameters **fold** (*int*) –

Returns

Return type filepath

fold_file
Path to the fold allocation dictionary

graph_docs
List of document names for which we would like to generate graphs

load(*runcfg*, *eval_dir*, *scratch_dir*)

Parameters

- **eval_dir** (*filepath*) – Directory to store evaluation results, basically anything that should be considered as important for long-term archiving and reproducibility
- **scratch_dir** (*filepath*) – Directory for relatively emphemeral intermediary results. One would be more inclined to delete scratch than eval
- **runcfg** (*RuntimeConfig or None*) – Runtime configuration. None for default options

See also:

`See()`

metrics
Selection of metrics to compute in reports.

model_paths (*rconf*, *fold*, *parser*)
Return attelo model paths in dictionary form

Parameters

- **rconf** (LearnerConfig) –
- **fold** (*int*) –

Returns

Return type Dictionary from attelo parser cache keys to paths

mpack_paths (*test_data*, *stripped=False*)
Return a dict of paths needed to read a datapack.

Usual keys are: * edu_input * pairings * features * vocab

Parameters

- **test_data** (*bool*) – If True, it's test data we wanted.
- **stripped** (*bool, defaults to False*) – If True, return path for a "stripped" version of the data (faster loading, but only useful for scoring).

Returns **res** – Paths to files that enable to read a datapack.

Return type dict

report_digits
Number of digits to display floats in reports.

report_dir_path (*test_data*, *fold=None*, *is_tmp=True*)
Path to a directory containing reports.

Parameters

- **test_data** (*bool*) – If True, the report is about the test set, otherwise the (usually, training) dataset.
- **fold** (*int, optional*) – Number of the fold under scrutiny ; if None, all folds.
- **is_tmp** (*bool, defaults to True*) – If True, only return the path to a provisional report in progress.

runcfg
Runtime configuration settings for the harness

scratch_dir
Directory for relatively emphemeral intermediary results.

One would be more inclined to delete scratch than eval

test_evaluation
The test evaluation for this harness, or None if it's unset

exception attelo.harness.interface.**HarnessException**
Bases: exceptions.Exception

Things that go wrong in the test harness itself.

attelo.harness.parse module

attelo.harness.report module

attelo.harness.util module

Miscellaneous utility functions

attelo.harness.util.**call**(*args*, ***kwargs*)
Execute a command and die prettily if it fails

attelo.harness.util.**force_symlink**(*source*, *link_name*, ***kwargs*)
Symlink from source to *link_name*, removing any prexisting file at *link_name*

attelo.harness.util.**makedirs**(*path*, ***kwargs*)
Create a directory and its parents if it does not already exist

attelo.harness.util.**md5sum_dir**(*path*, *blocksize=65536*)
Read a dir and return its md5 sum

attelo.harness.util.**md5sum_file**(*path*, *blocksize=65536*)
Read a file and return its md5 sum

attelo.harness.util.**subdirs**(*parent*)
Return all subdirectories within the parent dir (with combined path, ie. *parent/subdir*)

attelo.harness.util.**timestamp**()
Current date/time to minute resolution in an ISO format.

attelo.learning package

Submodules

attelo.learning.interface module

attelo.learning.local module

attelo.learning.oracle module

attelo.learning.perceptron module

attelo.learning.util module

attelo.metrics package

Submodules

attelo.metrics.tree module

Metrics to assess performance on tree-structured predictions.

Functions named as `*_loss` return a scalar value to minimize: the lower the better.

attelo.metrics.tree.**labelled_tree_loss**(*ref_tree*, *pred_tree*)
Compute the labelled tree loss.

The labelled tree loss is the fraction of edges that are incorrectly predicted, with a lesser penalty for edges with the correct attachment but the wrong label.

Parameters

- **ref_tree** (*list of edges (source, target, label)*) – reference tree
- **pred_tree** (*list of edges (source, target, label)*) – predicted tree

Returns **loss** – Return the tree loss between edges of `ref_tree` and `pred_tree`.

Return type float

See also:

tree_loss()

Notes

The labelled tree loss counts only half of the penalty for edges with the right attachment but the wrong label.

attelo.metrics.tree.**tree_loss**(*ref_tree*, *pred_tree*)
Compute the tree loss.

The tree loss is the fraction of edges that are incorrectly predicted.

Parameters

- **ref_tree** (*list of edges (source, target, label)*) – reference tree
- **pred_tree** (*list of edges (source, target, label)*) – predicted tree

Returns **loss** – Return the tree loss between edges of `ref_tree` and `pred_tree`.

Return type float

See also:

labelled_tree_loss()

Notes

For labelled trees, the tree loss checks for strict correspondence: it does not differentiate between incorrectly attached edges and correctly attached but incorrectly labelled edges.

attelo.optimisation package

Submodules

attelo.optimisation.astar module

Various search algorithms for combinatorial problems:

- [OK] Astar (shortest path with heuristics), and variants:
  - [OK] beam search (astar with size limit on waiting queue)

- **[OK] nbest solutions: implies storing solutions and a counter, and changing** return values (actually most search will make use of a recover_solution(s) to reconstruct desired data)
- branch and bound (astar with forward lookahead)

class attelo.optimisation.astar.**BeamSearch**(*heuristic=<function <lambda>>*, *shared=None*, *queue_size=10*)

Bases: *attelo.optimisation.astar.Search*

search with heuristics but limited size waiting queue (restrict to p-best solutions at each iteration)

add_queue(*items*, *ancestor_cost*)

new_state(*data*)

class attelo.optimisation.astar.**Search**(*heuristic=<function <lambda>>*, *shared=None*, *queue_size=None*)

Bases: object

abstract class for search each state to be explored must have methods

- `next_states()` - successor states + costs
- `is_solution()` - is the state a valid solution
- `cost()` - cost of the state so far (must be additive)

default is astar search (search the minimum cost from init state to a solution

Parameters

- **heuristic** – heuristics guiding the search (applies to state-specific data(), see *State*)
- **shared** – other data shared by all nodes (eg. for heuristic computation)
- **queue_size** – limited beam-size to store states. (commented out, pending tests)

add_queue(*items*, *ancestor_cost*)
Add a set of succesors to the search queue

:type items [(data, float)]

add_seen(*state*)
Mark a state as seen

has_empty_queue()
Return *True* if the search queue is empty

is_already_seen(*state*)
Return *True* if the given search state has already been seen

launch(*init_state*, *verbose=False*, *norepeat=False*)
launch search from initital state value

Param norepeat: there's no need for an "already seen states" datastructure

new_state(*data*)
Build a new state from the given data

pop_best()
Return and remove the lowest cost item from the search queue

reset_queue()
Clear out the search queue

reset_seen()
Mark every state as not yet seen

shared()
Information that can be shared across states

class attelo.optimisation.astar.**State**(*data*, *cost=0*, *future_cost=0*)
Bases: object

state for state space exploration with search

(contains at least state info and cost)

Note the functions *is_solution* and *next_states* which must be implemented

cost()
past path cost

data()
actual distinguishing contents of a state

future_cost()
future cost

is_solution()
return *True* if the state is a valid solution

next_states()
return the successor states and their costs

total_cost()
past and future cost

update_cost(*value*)
add to the current cost

attelo.parser package

Attelo is essentially a toolkit for producing parsers: parsers are black boxes that take EDUS as inputs and produce graphs as output.

Parsers follow the scikit fit/transform idiom. They are learned from some training data via the *fit()* function (this usually results in some model that the parser remembers; but a hypothetical purely rule-based parser might have a no-op fit function). Once fitted to the training data, they can be set loose on anything you might want to parse: the *transform* function will produce graphs from the EDUs.

Submodules

attelo.parser.attach module

A parser that only decides on the attachment task (whether this is directed or not depends on the underlying datapack and decoder). You could also combine this with the label parser

class attelo.parser.attach.**AttachClassifierWrapper**(*learner_attach*)
Bases: *attelo.parser.interface.Parser*

Parser that extracts attachments weights from an attachment classifier.

This parser is really meant to be used in conjunction with other parsers downstream that make use of these weights.

If you use it in standalone mode, it will just provide the standard unknown prediction everywhere

Notes

Cache keys

- attach: attachment model path

fit (*dpacks*, *targets*, *nonfixed_pairs=None*, *cache=None*)
Extract whatever models or other information from the multipack that is necessary to make the parser operational

Parameters **mpack** (*MultiPack*) –

transform (*dpack*, *nonfixed_pairs=None*)

class attelo.parser.attach.**AttachPipeline** (*learner*, *decoder*)
Bases: *attelo.parser.pipeline.Pipeline*

Parser that performs the attachment task.

Attachments may be directed or undirected depending on the datapack and models.

For the moment, this assumes AD models, but perhaps over time could be generalised to A.D models too.

This can work as a standalone parser: if the datapack is unweighted it will initalise it from the classifier. Also, if there are pre-existing weights, they will be multiplied with the new weights.

Notes

fit() and transform() have a 'cache' parameter that is a dict with expected keys: * attach: attachment model path

attelo.parser.full module

A 'full' parser does the attach, direction, and labelling tasks

class attelo.parser.full.**AttachTimesBestLabel**
Bases: *attelo.parser.interface.Parser*

Intermediary parser that adjusts the attachment weight by multiplying the best label weight with it.

This is most useful in the middle of a parsing pipeline: we need something upstream to assign initial attachment and label weights (otherwise we get the default 1.0 everywhere), and something downstream to make predictions (otherwise it's UNKNOWN everywhere)

fit (*dpacks*, *targets*, *nonfixed_pairs=None*, *cache=None*)

transform (*dpack*, *nonfixed_pairs=None*)

class attelo.parser.full.**JointPipeline** (*learner_attach*, *learner_label*, *decoder*)
Bases: *attelo.parser.pipeline.Pipeline*

Parser that performs attach, direction, and labelling tasks.

For the moment, this assumes AD.L models, but we hope to explore possible generalisations of this idea over time.

In our working shorthand, this would be an AD.L:adl parser, ie. one that has separate attach-direct model and label model (AD.L); but which treats decoding as a joint-prediction task.

Notes

fit() and transform() have a *cache* parameter, it should be a dict with keys: * 'attach': attach model path * 'label': label model path

class attelo.parser.full.**PostlabelPipeline**(*learner_attach*, *learner_label*, *decoder*)
Bases: *attelo.parser.pipeline.Pipeline*

Parser that perform the attachment task (may be directed or undirected depending on datapack and models), and then the labelling task in a second step

For the moment, this assumes AD models, but perhaps over time could be generalised to A.D models too

This can work as a standalone parser: if the datapack is unweighted it will initalise it from the classifier. Also, if there are pre-existing weights, they will be multiplied with the new weights

Notes

fit() and transform() have a 'cache' parameter that is a dict with expected keys: * 'attach': attach model path * 'label': label model path

attelo.parser.interface module

Basic interface that all parsers should respect

class attelo.parser.interface.**Parser**
Bases: object

Parsers follow the scikit fit/transform idiom. They are learned from some training data via the *fit()* function. Once fitted to the training data, they can be set loose on anything you might want to parse: the *transform* function will produce graphs from the EDUs.

If the learning process is expensive, it would make sense to offer the ability to initialise a parser from a cached model

static deselect(*dpack*, *idxes*)
Common parsing pattern: mark all edges at the given indices as unrelated with attachment score of 0. This should normally exclude them from attachment by a decoder.

Warning: assumes a weighted datapack

This is often a better bet than using something like *DataPack.selected* because it keeps the unwanted edges in the datapack

static dzip(*fun*, *dpacks*, *targets*)
Apply a function on each datapack and the corresponding target block

Parameters

- **((a, b) -> (a, b))** (*fun*) –
- **[a]** (*dpacks*) –
- **[b]** (*targets*) –

Returns

Return type [a], [b]

fit(*dpacks*, *targets*, *cache=None*)
Extract whatever models or other information from the multipack that is necessary to make the parser operational

Parameters

- **dpacks** (*[DataPack]*) –
- **targets** (*[array(int)]*) – A block of labels for each datapack. Each block should have the same length as its corresponding datapack
- **cache** (*dict(string, object), optional*) – Paths to submodels. If set, this dictionary associates submodel names with filenames. The submodel names are arbitrary strings like "attach" or "label" (check the documentation for the parser itself to see what submodels it recognises) with some sort of cache.

  This usage is necessarily loose. The parser should be prepared to ignore a key if it does not exist in the cache. The typical cache value is a filepath containing a pickle to load or dump; but other objects may sometimes be used depending on the parser (eg. other caches if it's a parser that somehow combines other parsers together)

static **multiply**(*dpack*, *attach=None*, *label=None*)
If the datapack is weighted, multiply its existing probabilities by the given ones, otherwise set them

Parameters

- **(array(float), optional)** (attach) – If unset will default to ones
- **(2D array(float), optional)** (label) – If unset will default to ones

Returns

Return type The modified datapack

static **select**(*dpack*, *idxes*)
Mark any pairs except the ones indicated as unrelated

See also:

Parser.deselect

transform(*dpack*)
Refine the parse for a single document: given a document and a graph (for the same document), add or remove edges from the graph (mostly remove).

A standalone parser should be able to start from an unweighted datapack (a fully connected graph with all labels equally liekly) and pare it down with to a much more useful graph with one best label per edge.

Standalone parsers ought to also do something sensible with weighted datapacks (partially instantiated graphs), but in practice they may ignore them.

Not all parsers may necessarily standalone. Some may only be designed to refine already existing parses. Or may require further processing.

Parameters **dpack** (DataPack) – the graph to refine (can be unweighted for standalone parsers, MUST be weighted for other parsers)

Returns

predictions – the best graph/prediction for this document

(TODO: support n-best)

Return type *DataPack*

attelo.parser.intra module

Document-level parsers that first do sentence-level parsing.

An IntraInterParser applies separate parsers on edges within a sentence and then on edges across sentences.

class attelo.parser.intra.**FrontierToHeadParser**(*parsers*, *sel_inter='inter'*, *verbose=False*)
Bases: *attelo.parser.intra.IntraInterParser*

Intra/inter parser in which sentence recombination consists of parsing with edges from the frontier of sentential subtree to sentence head.

[ ] write and integrate an oracle that replaces lost gold edges (from non-head to head) with the closest alternative ; here this probably happens on leaky sentences and I still have to figure out what an oracle should look like.

class attelo.parser.intra.**HeadToHeadParser**(*parsers*, *sel_inter='inter'*, *verbose=False*)
Bases: *attelo.parser.intra.IntraInterParser*

Intra/inter parser in which sentence recombination consists of parsing with only sentence heads.

[ ] write and integrate an oracle that replaces lost gold edges (from non-head to head) with the closest alternative, here moving edges up the intra subtrees so they link the (recursive) heads of their original nodes.

class attelo.parser.intra.**IntraInterPair**
Bases: *attelo.parser.intra.IntraInterPair*

Any pair of the same sort of thing, but with one meant for intra-sentential decoding, and the other meant for intersentential

fmap(*fun*)
Return the result of applying a function on both intra/inter

Parameters **fun** (*a -> b*) –

Returns

Return type IntraInterPair(b)

class attelo.parser.intra.**IntraInterParser**(*parsers*, *sel_inter='inter'*, *verbose=False*)
Bases: *attelo.parser.interface.Parser*

Parser that performs attach, direction, and labelling tasks; but in two phases:

1. by separately parsing edges within the same sentence
2. and then combining the results to form a document

This is an abstract class

Notes

/Cache keys/: Same as whatever included parsers would use. This parser will divide the dictionary into keys that have an 'intra:' prefix or not. The intra prefixed keys will be passed onto the intrasentential parser (with the prefix stripped). The other keys will be passed onto the intersentential parser

fit(*dpacks*, *targets*, *cache=None*)

transform(*dpack*)

class attelo.parser.intra.**SentOnlyParser**(*parsers*, *sel_inter='inter'*, *verbose=False*)
Bases: *attelo.parser.intra.IntraInterParser*

Intra/inter parser with no sentence recombination. We also chop off any fakeroot connections

class attelo.parser.intra.**SoftParser**(*parsers*, *sel_inter='inter'*, *verbose=False*)
Bases: *attelo.parser.intra.IntraInterParser*

Intra/inter parser in which sentence recombination consists of

1.passing intra-sentential edges through but

2.marking 1.0 attachment probabilities if they are attached and 1.0 label probabilities on the resulting edge

Notes

In its current implementation, this parser needs a global model, i.e. one fit on the whole dataset, so that it can correctly score intra-sentential edges. Different, alternative implementations could probably solve or work around this.

attelo.parser.intra.**edu_id2num**(*edu_id*)
Get the number of an EDU

attelo.parser.intra.**for_intra**(*dpack*, *target*)
Adapt a datapack to intrasentential decoding.

An intrasentential datapack is almost identical to its original, except that we set the label for each ('ROOT', edu) pairing to 'ROOT' if that edu is a subgrouping head (if it has no parents other than 'ROOT' within its subgrouping).

This should be done before either *for_labelling* or *for_attachment*

Returns

- **dpack** (*DataPack*)
- **target** (*array(int)*)

attelo.parser.intra.**partition_subgroupings**(*dpack*)
Partition the pairings of a datapack along (grouping, subgrouping).

Parameters **dpack** (DataPack) – Datapack to partition

Returns **groups** – Map each (grouping, subgrouping) to the list of indices of pairings within the same subgrouping.

Return type dict from (string, string) to list of integers

Notes

- **(FAKE_ROOT, x) pairings are included in the group defined by** (grouping(x), subgrouping(x)).
- **This function is a tiny wrapper around** *attelo.table.grouped_intra_pairings*.

attelo.parser.label module

Labelling

class attelo.parser.label.**LabelClassifierWrapper**(*learner*)
Bases: *attelo.parser.interface.Parser*

Parser that extracts label weights from a label classifier.

This parser is really meant to be used in conjunction with other parsers downstream that make use of these weights.

If you use it in standalone mode, it will just provide the standard unknown prediction everywhere.

Notes

fit() and transform() have a 'cache' argument that is a dict with expected keys: * 'label': label model path

fit(*dpacks*, *targets*, *nonfixed_pairs=None*, *cache=None*)
Extract whatever models or other information from the multipack that is necessary to make the labeller operational.

Returns self

Return type object

transform(*dpack*, *nonfixed_pairs=None*)

class attelo.parser.label.**SimpleLabeller**(*learner*)
Bases: *attelo.parser.label.LabelClassifierWrapper*

A simple parser that assigns the best label to any edges with unknown labels.

This can be used as a standalone parser if the underlying classifier predicts UNRELATED.

Notes

fit() and transform() have a 'cache' parameter that is a dict with expected keys: * 'label': label model path

transform(*dpack*, *nonfixed_pairs=None*)

attelo.parser.pipeline module

Parser made by sequencing other parsers.

Ideally, we'd like to use sklearn.pipeline.Pipeline but our previous attempts have failed. The current trend is to try and slowly converge.

class attelo.parser.pipeline.**Pipeline**(*steps*)
Bases: *attelo.parser.interface.Parser*

Apply a sequence of parsers.

NB. For now we assume that these parsers can be fitted independently of each other.

Steps should be a tuple of names and parsers, just like in sklearn.

Parameters **steps** (*list*) – List of (name, parser) tuples that are chained.

named_steps
dict

Read-only attribute to access any step parameter by user given name. Keys are step names and values are step parameters.

fit(*dpacks*, *targets*, *nonfixed_pairs=None*, *cache=None*)
Fit.

named_steps

transform(*dpack*, *nonfixed_pairs=None*)
Transform.

Submodules

attelo.args module

Managing command line arguments

attelo.args.**add_common_args**(*psr*)
add usual attelo args to subcommand parser

attelo.args.**add_fold_choice_args**(*psr*)
ability to select a subset of the data according to a fold

attelo.args.**add_model_read_args**(*psr*, *help_*)
models files we can read in

Parameters **help** (*string*) – python format string for help *{}* will have a word (eg. 'attachment') plugged in

attelo.args.**add_report_args**(*psr*)
add args to scoring/evaluation

attelo.args.**validate_fold_choice_args**(*wrapped*)
Given a function that accepts an argparsed object, check the fold arguments before carrying on.

The idea here is that –fold and –fold-file are meant to be used together (xnor)

This is meant to be used as a decorator, eg.:

```
@validate_fold_choice_args
def main(args):
    blah
```

attelo.edu module

Uniquely identifying information for an EDU

class attelo.edu.**EDU**
Bases: *attelo.edu.EDU*

a class representing the EDU (id, span start and end, grouping, subgrouping)

span()
Starting and ending position of the EDU as an integer pair

attelo.edu.**FAKE_ROOT = EDU(id='ROOT', text='', start=0, end=0, grouping=None, subgrouping=None)**
a distinguished fake root EDU which simultaneously appears in all groupings

attelo.fold module

Group-aware n-fold evaluation.

Attelo uses a variant of n-fold evaluation, where we (still) andomly partition the dataset into a set of folds of roughly even size, but respecting the additional constraint that any two data entries belonging in the same "group" (determined a single distiguished feature, eg. the document id, the dialogue id, etc) are always in the same fold. Note that this makes it a bit harder to have perfectly evenly sized folds

Created on Jun 20, 2012

@author: stergos

contribs: phil

attelo.fold.**fold_groupings**(*fold_dict*, *fold*)
Return the set of groupings that belong in a fold. Raise an exception if the fold is not in the fold dictionary

:rtype frozenset(int)

attelo.fold.**make_n_fold**(*groupings*, *folds*, *rng*)
Given a set of groupings and a desired number of folds, return a fold selection dictionary assigning a fold number to each each grouping (see *attelo.edu.EDU*).

Parameters **rng** (*:py:class:random.Random:*) – random number generator (hint: the random module will be just fine if you don't mind shared state)

:rtype dict(string, int)

attelo.fold.**select_testing**(*mpack*, *fold_dict*, *fold*)
Given a division into folds and a fold number, return only the test items for that fold

Return type Multipack

attelo.fold.**select_training**(*mpack*, *fold_dict*, *fold*)
Given a division into folds and a fold number, return only the training items for that fold

Return type Multipack

attelo.graph module

graph visualisation

exception attelo.graph.**Alarm**
Bases: exceptions.Exception

Exception to raise on signal timeout

class attelo.graph.**GraphSettings**
Bases: *attelo.graph.GraphSettings*

Parameters

- **hide** (*string or None*) – 'intra' to hide links between EDUs in the same subgrouping; 'inter' to hide links across subgroupings; None to show all links
- **select** (*[string] or None*) – EDU groupings to graph (if None, all groupings will be graphed unless)
- **unrelated** (*bool*) – show unrelated links
- **timeout** (*int*) – number of seconds to allow graphviz to run before it times out
- **quiet** (*bool*) – suppress informational messages

attelo.graph.**alarm_handler**(*_*, *frame*)
Raise Alarm on signal

attelo.graph.**diff_all**(*edus*, *src_predictions*, *tgt_predictions*, *settings*, *output_dir*)
Generate graphs for all the given predictions. Each grouping will have its own graph, saved in the output directory

attelo.graph.**graph_all**(*edus*, *predictions*, *settings*, *output_dir*)
Generate graphs for all the given predictions. Each grouping will have its own graph, saved in the output directory

attelo.graph.**mk_diff_graph**(*title*, *edus*, *src_links*, *tgt_links*, *settings*)
Convert attelo predictions to a graphviz graph diplaying differences between two predictions

Predictions here consist of an EDU followed by a list of (parent name, relation label) tuples

Parameters **tgt_links** – if present, we generate a graph that represents a difference between the links and tgt_links (by highlighting links that only occur in one or the other)

attelo.graph.**mk_single_graph**(*title*, *edus*, *links*, *settings*)
Convert single set of attelo predictions to a graphviz graph

attelo.graph.**select_links**(*edus*, *links*, *settings*)
Given a set of edus and of edu id pairs, return only the pairs whose ids appear in the edu list

Parameters

- **intra** – if True, in addition to the constraints above, only return links that are in the same subgrouping
- **inter** – if True, only return links between subgroupings

attelo.graph.**write_dot_graph**(*filename*, *dot_graph*, *run_graphviz=True*, *quiet=False*, *timeout=30*)
Write a dot graph and possibly run graphviz on it

attelo.io module

attelo.report module

attelo.score module

attelo.table module

Manipulating data tables (taking slices, etc)

class attelo.table.**DataPack**
Bases: *attelo.table.DataPack*

A set of data that can be said to belong together.

A typical use of the datapack would be to group together data for a single document/grouping. But in cases where this distinction does not matter, it can also be convenient to combine data from multiple documents into a single pack.

Notes

A datapack is said to be

•single document (the usual case) it corresponds to a single document or "stacked" if it is made by joining multiple datapacks together. Some functions may only behave correctly on single-document datapacks

•weighted if the graphs tuple is set. You should never see weighted datapacks outside of a learner or decoder

Parameters

- **(EDU)** (*edus*) – effectively a set of edus
- **([(EDU, EDU)])** (*pairings*) – edu pairs

- **2D array(float)** (*data*) – sparse matrix of features, each row corresponding to a pairing
- **1D array (should be int, really)** (*target*) – array of predictions for each pairing
- **ctarget** (*dict from string to objects*) – Mapping from grouping name to structured target
- **([string])** (*vocab*) – list of relation labels (NB: by convention label zero is always the unknown label)
- **([string])** – feature names (corresponds to the feature indices) in data
- **(None or Graph)** (*graph*) – if set, arrays representing the probabilities (or confidence scores) of attachment and labelling

get_label(*i*)
Return the class label for the given target value.

Parameters **(int, less than len(self.labels))** (*i*) – a target value

See also:

label_number

label_number(*label*)
Return the numerical label that corresponnds to the given string label

Useful idiom: *unrelated = dpack.label_number(UNRELATED)*

Parameters **(string in self.labels)** (*label*) – a label string

See also:

get_label

classmethod load(*edus*, *pairings*, *data*, *target*, *ctarget*, *labels*, *vocab*)
Build a data pack and run some sanity checks (see :py:method:sanity_check') (recommended if reading from disk)

Return type *DataPack*

sanity_check()
Raising *DataPackException* if anything about this datapack seems wrong, for example if the number of rows in one table is not the same as in another

selected(*indices*)
Return only the items in the specified rows

set_graph(*graph*)
Return a copy of the datapack with weights set

classmethod vstack(*dpacks*)
Combine several datapacks into one.

The labels and vocabulary for all packs must be the same

exception attelo.table.**DataPackException**(*msg*)
Bases: exceptions.Exception

An exception which arises when worknig with an attelo data pack

class attelo.table.**Graph**
Bases: *attelo.table.Graph*

A graph can only be interpreted in light of a datapack.

It has predictions and attach/label weights. Predictions work like *DataPack.target*. The weights are useful within parsing pipelines, where it is sometimes useful for an intermediary parser to manipulate the weight vectors that a parser may calculate downstream.

See the parser interface for more details.

Parameters

- **prediction** (*array(int)*) – label for each edge (each cell corresponds to edge)
- **attach** (*array(float)*) – attachment weights (each cell corresponds to an edge)
- **label** (*2D array(float)*) – label attachment weights (edge by label)

Notes

Predictions are always labels; however, datapack targets may also be -1/0/1 when adapted to binary attachment task

selected(*indices*)
Return a subset of the links indicated by the list/array of indices

tweak(*prediction=None*, *attach=None*, *label=None*)
Return a variant of the current graph with some values changed.

Parameters

- **prediction** (*1D array of int16*) – Predicted label for each pair of EDUs
- **attach** (*1D array of float*) – Attachment scores for each pair of EDUs
- **label** (*2D array of float*) – Score of each label for each pair of EDUs

Returns **g_copy** – Copy of self with prediction, attach or label overridden with the values passed as arguments.

Return type *Graph*

Notes

This returns a copy of *self* with graph changed, because "[EYK] superstitiously believes that datapacks and graphs should be immutable as much as possible, and that mutability in the parsing pipeline would lead to confusion; hence this and namedtuples instead of simple getting and setting".

classmethod vstack(*graphs*)
Combine several graphs into one.

class attelo.table.**Multipack**
Bases: dict

A multipack is a mapping from groupings to datapacks

This class exists purely for documentation purposes; in practice, a dictionary of string to Datapack will do just fine

attelo.table.**UNKNOWN = '__UNK__'**
distinguished internal value for post-labelling mode

attelo.table.**UNRELATED = 'UNRELATED'**
distinguished value for unrelated relation labels

attelo.table.**attached_only**(*dpack*, *target*)
Return only the instances which are labelled as attached (ie. this would presumably return an empty pack on completely unseen data)

Parameters

- **dpack** (DataPack) – Original datapack
- **target** (*array(int)*) – Original targets

Returns

- **dpack** (*DataPack*) – Transformed datapack, with binary labels
- **target** (*array(int)*) – Transformed targets, with binary labels

attelo.table.**for_attachment**(*dpack*, *target*)
Adapt a datapack to the attachment task.

This could involve: * selecting some of the features (all for now, but may change in the future) * modifying the features/labels in some way: we currently binarise labels to {-1 ; 1} for UNRELATED and not-UNRELATED respectively.

Parameters

- **dpack** (DataPack) – Original datapack
- **target** (*array(int)*) – Original targets

Returns

- **dpack** (*DataPack*) – Transformed datapack, with binary labels
- **target** (*array(int)*) – Transformed targets, with binary labels

attelo.table.**for_labelling**(*dpack*, *target*)
Adapt a datapack to the relation labelling task (currently a no-op).

This could involve * selecting some of the features (all for now, but may change in the future) * modifying the features/labels in some way (in practice no change)

Parameters

- **dpack** (DataPack) – Original datapack
- **target** (*array(int)*) – Original targets

Returns

- **dpack** (*DataPack*) – Transformed datapack, with binary labels
- **target** (*array(int)*) – Transformed targets, with binary labels

attelo.table.**get_label_string**(*labels*, *i*)
Return the class label for the given target value.

attelo.table.**grouped_intra_pairings**(*dpack*, *include_fake_root=False*)
Retrieve intra pairings from a datapack, grouped by subgrouping.

Parameters

- **dpack** (DataPack) – The datapack under scrutiny.
- **include_fake_root** (*boolean, optional*) – If True, (FAKE_ROOT_ID, x) pairings are included in the group defined by (grouping(x), subgrouping(x)).

Returns **groups** – Map each (grouping, subgrouping) to the list of pairing indices within the same subgrouping.

Return type dict from (string, string) to list of integers

Notes

The result roughly corresponds to a hypothetical *dpack.pairings['intra'].groupby(['grouping', 'subgrouping']).groups*.

attelo.table.**groupings**(*pairings*)
Given a list of EDU pairings, return a dictionary mapping grouping names to list of rows within the pairings.

Return type dict(string, [int])

attelo.table.**idxes_attached**(*dpack*, *target*)
Indices of attached pairings from dpack, according to target.

Parameters

- **dpack** (DataPack) – Datapack
- **target** (*list of integers*) – Label for each pairings of dpack

Returns

- **indices** (*array of integers*) – Indices of attached pairings.
- *TODO*
- —-
- *Try and apply widely, especially for parser.intra ;*
- *search for e.g. "target != unrelated" and "target[i] != unrelated".*

attelo.table.**idxes_fakeroot**(*dpack*)
Return datapack indices only the pairings which involve the fakeroot node

attelo.table.**idxes_inter**(*dpack*, *include_fake_root=False*)
Return indices of pairings from different subgroupings.

Parameters

- **dpack** (DataPack) – Datapack under scrutiny
- **include_fake_root** (*boolean, optional*) – If True, pairings of the form (FAKE_ROOT_ID, x) are included.

Returns **idxes** – Indices of the inter pairings.

Return type list of int

attelo.table.**idxes_intra**(*dpack*, *include_fake_root=False*)
Return indices of pairings from same subgrouping, inside a datapack.

Parameters

- **dpack** (DataPack) – Datapack under scrutiny
- **include_fake_root** (*boolean, optional*) – If True, pairings of the form (FAKE_ROOT_ID, x) are included.

Returns **idxes** – Indices of the intra pairings.

Return type list of int

attelo.table.**locate_in_subpacks**(*dpack*, *subpacks*)
Given a datapack and some of its subpacks, return a list of tuples identifying for each pair, its subpack and index in that subpack.

If a pair is not found in the list of subpacks, we return None instead of tuple

Returns

Return type [None or (DataPack, float)]

attelo.table.**mpack_pairing_distances**(*mpack*)
Return for each target value (label) in the multipack. See *pairing_distances()* for details

:rtype dict(int, (int, int))

attelo.table.**pairing_distances**(*dpack*)
Return for each target value (label) in the datapack, the left and right maximum distances of edu pairings (in number of EDUs, so adjacent EDUs have distance of 0)

Note that we assume a single-document datapack. If you give this a stacked datapack, you may get very large distances to the fake root

:rtype dict(int, (int, int))

attelo.table.**select_window**(*dpack*, *window*)
Select only EDU pairs that are at most *window* EDUs apart from each other (adjacent EDUs would be considered *0* apart)

Note that if the window is *None*, we simply return the original datapack

Note that will only work correctly on single-document datapacks

attelo.util module

General-purpose classes and functions

class attelo.util.**ArgparserEnum**
Bases: enum.Enum

An enumeration whose values we spit out as choices to argparser

classmethod choices_str()
available choices in this enumeration

classmethod from_string(*string*)
from command line arg

classmethod help_suffix(*default*)
help text suffix showing choices and default

class attelo.util.**Team**
Bases: *attelo.util.Team*

Any collection where we have the same thing but duplicated for each attelo subtask (eg. models, learners,)

fmap(*func*)
Apply a function to each member of the collection

attelo.util.**concat_i**(*iters*)
Merge an iterable of iterables into a single iterable

attelo.util.**concat_l**(*iters*)
Merge an iterable of iterables into a list

attelo.util.**mk_rng**(*shuffle=False*, *default_seed=None*)
Return a random number generator instance, hard-seeded unless we ask for shuffling to be enabled

(note: if shuffle mode is enable, the rng in question will just be the system generator)

attelo.util.**truncate**(*text*, *width*)
Truncate a string and append an ellipsis if truncated

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

a

A

B

C

D

E

F

G

H

S

T

U

V

W

Z