SKLEARN-based classifiers

THE RESULTS REPORTED HERE MAY BE DIFFERENT FROM THE RESULTS IN THE BOOK: YOU GET DIFFERENT SCORES WITH DIFFERENT SETS OF PARAMETERS, AND THE VALUES FOR THE PARAMETERS USED HERE MAY BE DIFFERENT FROM THOSE IN THE BOOK. THAT'S REALLY THE WHOLE POINT OF THIS BOOK!

The classifiers in Chapters 5 to 8 and 10 are all based on standard learning algorithms, using the SKLEARN implementations of those algorithms. Much of what we want to do with these is fairly generic – apply classifier C to dataset D and see what happens, and maybe look under the bonnet to see some of the key datastructures. We will therefore given a generic introduction to how we run these classifiers before looking in detail at interesting aspects of individual classifiers.

We start by importing the package **sklclassifiers.baseclassifiers**. You can do this either by doing (remember, you should always run these programs from the top-level directory unless we explicitly say you should run them from somewhere else)

```
>>> from sklclassifiers.baseclassifiers import *
```

or

```
>>> from sklclassifiers import baseclassifiers
```

or

```
>>> import sklclassifiers.baseclassifiers
```

The first of these means we don't have to type the module prefix for things we want to use from **sklclassifiers.baseclassifiers**, so it's a bit more convenient, but it does mean that it would in principle be possible to muddle up things from here with code from other modules. If you want to be careful, use one of the others – we will use the version where we just import everything from **sklclassifiers.baseclassifiers** into the top-level of the interpreter.

We assume that you will have installed the various datasets as described in *datasets.docx*. If you have, then the classes WASSA, SEM4, SEM11, KWT, CARER, IMDB should all contain pointers to where the datasets are kept:

```
>>> WASSA.PATH
'/Library/WebServer/CGI-Executables/SENTIMENTS/CORPORA/TWEETS/WASSA/EN'
>>> SEM4.PATH
'/Library/WebServer/CGI-Executables/SENTIMENTS/CORPORA/TWEETS/SEM4'
…
```

(you will probably have set **CORPORA** in **basics/corpora.py** to something different from **/Library/WebServer/CGI-Executables/SENTIMENTS/CORPORA**). We will need these locations when we tell the classifiers what datasets we want to apply them to.
We also need the classifiers themselves. We can import these from **py**. Once we have imported **baseclassifiers** and **allclassifiers** then we can apply a classifier to a dataset. We start by applying a LEXCLASSIFIER (which is our first classifier in Chapter 5) to the English SEM4 datasets. This produces quite a lot of output.

- The first thing it prints is a list of parameters. There are a huge number of parameters for controlling the behaviour of the classifiers, with different classifiers requiring different parameters to be set. We will want to run experiments where we compare the performance of different classifiers on the same datasets, and it would be tedious to specify all the parameters that we might need as optional arguments of everything. We therefore make a table of default values for arguments that we might need. These can be overridden when we call everything. So the first thing that gets printed is the list of parameters. The format is quite like .json, but there are a few things that are not easily serialisable, so you can't read it directly as if it were .json, but you can copy bits of it to use (or to edit and then use). We will look at specified parameters as they become relevant – there are lots that are not relevant for particular classifiers, e.g. **LEXCLASSIFIERS** do not need a value for the number of hidden layers, but as noted it is more convenient to set values for all of them even if we do not need them all for a given classifier. The significant ones for the current example are highlighted in pink.

- The program then prints out progress information – how long is it taking to read and convert the dataset, which fold are we currently processing (everything does N-fold cross validation, as described in Chapter 1).
- After each classifier has been run, the various statistics (precision, recall, …) for that classifier for each dataset are printed.
- At the end, the Jaccard scores for each classifier:dataset pair are printed. In the current example there is only one classifier and one dataset, so this final table isn't really needed, but when we have multiple classifiers and datasets it becomes very useful.

```
>>> from sklclassifiers.baseclassifiers import *

>>> from sklclassifiers.allclassifiers import *

>>> x = everything(datasets=os.path.join(SEM4.PATH, "EN"),
classifiers=LEXCLASSIFIER)
```

```
params = {

N: 70000,

alpha: 1e-05,

bestthreshold: 'global',

classifier: <class 'sklclassifiers.lexclassifier.LEXCLASSIFIER'>,

clsfname: 'LEX',

dataset: 'SEM4-EN',

fold1: 0,

folds: 0,

hiddenlayers: <function everything.<locals>.<lambda> at 0x7f9185f215e0>,

justone: False,

max_iter: 20000,

maxdictsize: 10000,

model: False,

solver: 'sgd',

stemmer: 'none',

threshold: 1,

tokeniser: 'standard',
useDF: False,
useneutral: True,
usesaved: True,
wthreshold: 5,
}
MAKING DATASET
1.00 done: 0.003 secs remaining
DONE -- 0.61 seconds
FOLD 0
FOLD 1
FOLD 2
FOLD 3
FOLD 4
FOLD 5
FOLD 6
FOLD 7
FOLD 8
FOLD 9
Scores for LEX
          Precision  Recall     micro F1   macro F1   Jaccard
SEM4-EN   0.636      0.747      0.687      0.700      0.523

All scores for datasets SEM4-EN using classifiers LEX
          LEX
SEM4-EN   0.523
```

The highlighted parameters here are

N: this is the maximum number of tweets in the dataset that we want to look at
clsfname: 'LEX', dataset: 'SEM4-EN': abbreviated forms of the names of the dataset and classifier, to be used in the tables and to make it see what is actually being done.
fold1: 0, folds: 0: if folds is set to 0 then we do 10 folds if the actual size of the dataset is less than 20K and 5 folds if it's greater than 20K, but this can be overridden. If fold1 is set to anything other than 0 then only that fold will be carried out (saves time when doing development).
stemmer: 'none', tokeniser: 'standard': if tokeniser is 'none' then we simply use the built-in split function to break the text into words; 'standard' will get the regex-based tokeniser from **chapter4.tokenisers**. You can use other things by adding them to **chapter4/tokenisers.py** and then making the appropriate changes to makeTweet in **sklclassifiers/tweets.py.** Similarly, setting stemmer to 'standard' will use the stemmers from **chapter4.arabicstemmer, chapter4.spanishstemmer** and **chapter4.stem3**. Again you can

use other stemmers by adding them to chapter4 and then making the required changes to **makeTweet**.

And that's it. Apply **sklclassifiers.baseclassifiers.everything** to a set of classifiers and a set of datasets and see what happens and see what happens. **baseclassifiers** has a number of predefined collections of datasets which can be more convenient to use (at the bottom of the file), e.g. **ENGLISH0** is the SEM4-EN, SEM11-EN and WASSA-EN datasets:

```
>>> x = everything(datasets=ENGLISH0, classifiers=[LEXCLASSIFIER, NBCLAS-
SIFIER])
```

```
...
...
Scores for LEX
           Precision  Recall     micro F1   macro F1   Jaccard
SEM4-EN    0.617      0.733      0.670      0.683      0.504
SEM11-EN   0.476      0.557      0.513      0.513      0.345
WASSA-EN   0.568      0.661      0.611      0.618      0.440
...
...
Scores for NB
           Precision  Recall     micro F1   macro F1   Jaccard
SEM4-EN    0.887      0.863      0.875      0.859      0.778
SEM11-EN   0.476      0.390      0.429      0.435      0.273
WASSA-EN   0.831      0.822      0.826      0.812      0.704

All scores for datasets SEM4-EN,SEM11-EN,WASSA-EN using classifiers
LEX,NB
           LEX        NB
SEM4-EN    0.504      0.778
SEM11-EN   0.345      0.273
WASSA-EN   0.440      0.704
```

To try out different parameters, we can set overrides. If we set a list of overrides, we will get a table with an enumeration of the results for the different cases. If we specify several sets of overrides, using different tokenisers, we get the following:

```
>>> x = everything(datasets=ENGLISH0,
                   classifiers=[NBCLASSIFIER],
                   overrides=[{"tokeniser": "none"},
                              {"tokeniser": "NLTK"},
                              {"tokeniser": "standard"},
                              {"tokeniser":"standard1"}])
```

```
...
```

```
          NB-0       NB-1       NB-2       NB-3
SEM4-EN   0.826      0.900      0.887      0.881
SEM11-EN  0.452      0.497      0.475      0.481
WASSA-EN  0.760      0.855      0.841      0.834
```

The classifiers are called NB-0, NB-1, ..., showing that the first one uses the first set of overrides (i.e. the ones at position 0 in the list) and the second uses the second (at position 1). This lets us see that the built-in NLTK tokeniser outperforms the regex-based one from chapter4.tokenisers, and that the standard version of the regex-based one, which treats sequences of emojis as single tokens, outperforms one which treats them individually. The major differences between the three lies in the way that they treat emojis: the NLTK tokeniser treats sequences of words and emojis as single tokens, i.e. it treats *cake*😒😒 as a single item, while the first regex-based one treats this as two items, *cake* and 😒😒, and the second treats it as three items, *cake*,😒 and 😒. These results are slightly unexpected, since you wouldn't expect the single item *cake*😒😒 occur often enough to carry any emotional weight, and you would also expect the single token 😒 to occur much more often than the pair 😒😒 and hence to be more likely carry emotional weight. So it may well be worth thinking more deeply about this, since tokenisation is the very first thing you do. But for now we will just note that allowing sets of overrides makes it easy to carry out experiment s with minor variations in the parameters. Following such experiments up is a task for the reader: the point here is that using sets of overrides makes doing them straightforward.

Lexicon-based classifiers (Chapter 5)

NRCCLASSIFIERS

The first classifiers we looked at used hand-coded sentiment lexicons. Obtaining such a lexicon is quite hard work, but once you have one then it is extremely easy way to write a classifier. The one we use is based on the EMOLEX lexicon (Mohammad & Turney, 2013). The terms and conditions for this lexicon say “The lexicon mentioned in this page can be used freely for non-commercial research and educational purposes”, so as with the SEM4/SEM11/WASSA datasets we are not distributing the lexicon itself. Instead you should download it from https://saifmohammad.com/WebPages/NRC-Emotion-Lexicon.htm and unzip the NRC-Emotion-Lexicon.zip file that you get from in your CORPORA directory under DATA.
Then doing

```
>>> x = everything(datasets=os.path.join(SEM4.PATH, "EN"),
classifiers=NRCCLASSIFIER)
```

```
...
All scores for datasets SEM4-EN using classifiers NRC
          NRC
SEM4-EN   0.451
```

will split the dataset stored at **<CORPORA>/TWEETS/SEM4/EN/wholething.csv** (**<CORPORA>** is the directory where you have put the corpora) into a series or training/testing folds, make a classifier for the training set in each fold, apply to the test set for the fold, and print the aggregated scores.
At the end of all this, **x** is a table linking classifiers and datasets to sets of classifiers and scores:

```
>>> for c in x:
...  print(c)
...  for d in x[c]:
```

```
...   print(" %s"%(d))
...   print("  %s %s"%x[c][d][:2])
...   for k in x[c][d][2]: print("  %s"%(k))
...   print(" %s"%(x[c][d][-1]))
```

```
NRC
 SEM4-EN
  0.6218219605126429 0.9973917284654185
  <sklclassifiers.nrcclassifier.NRCCLASSIFIER object at 0x7f7e1b20f400>
  <sklclassifiers.nrcclassifier.NRCCLASSIFIER object at 0x7f7e1b20f6d0>
  <sklclassifiers.nrcclassifier.NRCCLASSIFIER object at 0x7f7e1be429a0>
  <sklclassifiers.nrcclassifier.NRCCLASSIFIER object at 0x7f7e1be42b50>
  <sklclassifiers.nrcclassifier.NRCCLASSIFIER object at 0x7f7e18f9a790>
  <sklclassifiers.nrcclassifier.NRCCLASSIFIER object at 0x7f7e1c1b7c40>
  <sklclassifiers.nrcclassifier.NRCCLASSIFIER object at 0x7f7e1c1b7cd0>
  <sklclassifiers.nrcclassifier.NRCCLASSIFIER object at 0x7f7e1be49b80>
  <sklclassifiers.nrcclassifier.NRCCLASSIFIER object at 0x7f7e197c49d0>
  <sklclassifiers.nrcclassifier.NRCCLASSIFIER object at 0x7f7e1c1b7040>
  [0.5469835466179159, 0.7203852327447833, 0.6218219605126429, 
0.5957142857142862, 0.45119131396400924]
```

In other words, **x["NRC"]["SEM4-EN"]** is a tuple containing the following:

- the average Jaccard score for the folds
- the average proportionality score for the folds
- the classifiers for each fold
- a list containing all the metrics for the folds (including a second copy of the Jaccard score as the last item: it may be a bit clunky to have the scores in two places, with two copies of the Jaccard score, but that's just how the code ended up)

Note that we ended up with 10 classifiers, one per fold. We can easily get hold of one:

```
>>> c = x["NRC"]["SEM4-EN"][2][0]
>>> print(c)
```

```
<sklclassifiers.nrcclassifier.NRCCLASSIFIER object at 0x7f7e1b20f400>
```

A classifier has a number of constituent parts (**showattr** is defined in **basics/utilities** and is useful for showing what an object is made of):

```
>>> showattrs(c)
```

```
emotionIndex:   [True, False, False, True, True, False, False, True, 
False, False]
lex:            {'aback':[0, 0, 0, 0], 'abacus':[0, 0, 0, 0], 
                  'abandon':[0, 1, 0, 1], 'abandoned':[1, 1, 0, 1],
                  ...}
name:           NRC
params:         {'threshold': 0.9500000000000003, 'N': 70000, ...}
score:          (0.6071161048689144, 457, 2119, 373, 166)
targetIndex:    [True, True, True, True, False]
test:           <sklclassifiers.tweets.DATASET object at 0x7f7e1c3dbac0>
testtime:       3.926137859901686e-05
threshold:      0.9500000000000003
train:          <sklclassifiers.tweets.DATASET object at 0x7f7e1c3db940>
traintime:      0.7854578495025635
```

These are as follows (attributes that are common to all classifiers are highlighted in pink:

emotionIndex and **targetIndex**: these are used for comparing and merging the emotions in the lexicon and the dataset – the EMOLEX lexicon uses a different set of emotions from the SEM4-EN dataset, and obviously only the ones that are common to both can be used. See *Chapter 5*.

lex: this is an encoding of the emotions assigned to each word by the lexicon. The shared emotions between the lexicon and the dataset are ['**anger**', '**fear**', '**joy**', '**sadness**'], so saying that *abandon* expresses [0, 1, 0, 1] means that it expresses **fear** and **sadness**. Note that *abandoned* expresses **anger** and **fear** and **sadness**: the fact that inflected forms of a word can express different combinations of emotions means that stemming probably is not a good thing to do with this lexicon.

name: obvious enough.

params: these are the parameters governing the behaviour of the classifier. Most of them are the ones supplied when the classifier was initialised, but threshold was set by trying a range of thresholds on the training data and choosing the one that works best.

score: this is a record of the scores for the classifier when it is applied to the test data

test and **train**: the datasets obtained for the training and testing parts of the current fold.

testtime, **traintime**: timings

threshold: the same thing as params["threshold"]

test and **train** are both **datasets**, where a **dataset** has the following attributes:

```
GS:   [array([0, 0, 0, 1, 0]), array([1, 0, 0, 0, 0]), ...]
df:   {'Incredibly': 2, 'shocked': 1, 'and': 1487, 'disappointed': 7,...}
emotions:  ['anger', 'fear', 'joy', 'sadness']
idf: {'Incredibly': 0.3333333333333333, 'shocked': 0.5, '...}
index:     {'Incredibly': 0, 'shocked': 1, 'and': 2, ...}
```

```
invindex:  {0: 'Incredibly', 1: 'shocked', 2: 'and', ...}

params:    {...}

tweets:    [<sklclassifiers.tweets.TWEET object at 0x7f7e19976370>, …]
```

These are as follows:

GS: this is the Gold Standard set of emotions for the tweet, encoded as a vector of 0s and 1s as above
df, idf: these are used for calculating TF-IDF scores if we want them
emotions: the emotions that are available for this tweet:
index, invindex: used for mapping sets of words to sparse matrices and back again
params: as above
tweets: the set of tweets that make up this dataset.

Tweets in turn are represented as follows:

```
GS:        [0 0 0 1 0]

id:        2017-En-41199

params:    {...}

predicted: [1, 0, 0, 1, 0]

src:  Incredibly shocked and disappointed with @united customer service. Really making me rethink flying with them in the future. #unhappy

tf:  {'Incredibly': 0.043478260869565216, ...}

tokens:    ['Incredibly', 'shocked', 'and', 'disappointed', 'with', …]
```

GS: the Gold Standard for this tweet
id: self-axplanatory

params: as above

predicted: the predicted values for the tweet if the classifier has been applied to it, otherwise **None**

src: the source text

tf: term frequencies for the words in the tweet

tokens: list of tokens obtained by the tokeniser (and stemmer if used) that was used for making the dataset

Everything in an **NRCCLASSIFIER** is driven by its emotion lexicon (the **lex**). The classifier works by adding together the entries in the lexicon for the words that it contains and comparing these with the threshold. Very simple, **very fast,** works OK.

LEXCLASSIFIERS

The next set of classifiers were made by extracting a sentiment lexicon from the training data (Chapter 5). The following example compares the performance of a classifier based on an extracted lexicon with an NRCCLASSIFIER.

```
>>> # setting showresults and showparams to False suppresses printing during training and testing
>>> x = everything(datasets=ENGLISH0, classifiers=[NRCCLASSIFIER, LEX-
CLASSIFIER], overrides=[{"stemmer": "none"}], showresults=False, show-
params=False)
```

```
Scores for NRC
           Precision  Recall     micro F1   macro F1   Jaccard
SEM4-EN    0.547      0.720      0.622      0.596      0.451
SEM11-EN   0.357      0.448      0.397      0.354      0.248
WASSA-EN   0.522      0.712      0.603      0.576      0.431

Scores for LEX
           Precision  Recall     micro F1   macro F1   Jaccard
SEM4-EN    0.636      0.747      0.687      0.700      0.523
SEM11-EN   0.461      0.598      0.521      0.521      0.352
WASSA-EN   0.574      0.683      0.624      0.632      0.453

All scores for SEM4-EN,SEM11-EN,WASSA-EN using classifiers LEX,NRC
           LEX        NRC
SEM4-EN    0.523      0.451
SEM11-EN   0.352      0.248
WASSA-EN   0.453      0.431
```

We see that using a lexicon extracted from the training data is more effective than using a hand-coded one. The attributes of a LEXCLASSIFIER are the same as for an NRCCLASSIFIER apart from the following:

```
matrix:      (0, 0) 1, (3, 0) 1, (5, 0), 1, (8, 0), 1 (11, 0), 2, ...
             ...
             (4232, 1792), 1, (5376, 1792) 1, ...
model:     False
name:      LEX
scoredict: {'Incredibly': array([0., 0., 0., 1.]),
            'shocked': array([0., 0., 0., 1.]),
            'and': array([0.2387, 0.2481, 0.2837, 0.2293,]), ...}
values:    [3, 0, 4, 0, 3, 1, 1, 0, 3, 3, ...]
```

- **matrix**: the matrix is a sparse matrix representing the entire dataset, where each row in the matrix represents the words that occur in a single tweet. We can see what is going on better if we convert this sparse matrix to a standard array and pick out the non-zero elements of a row:

  ```
  >>> clex = x["LEX"]["SEM4-EN"][2][0]
  ```

```
>>> a = clex.matrix.toarray()
>>> a # converted the sparse array to a normal numpy array
```

```
array([[1, 2, 1, ..., 0, 0, 0],
       [0, 0, 3, ..., 0, 0, 0],
       [0, 1, 1, ..., 0, 0, 0],
       ...,
       [1, 0, 0, ..., 0, 0, 0],
       [1, 0, 2, ..., 0, 0, 0],
       [3, 2, 0, ..., 0, 0, 0]])
```

```
>>> a[0] # have a look at the first line
```

```
array([1, 2, 1, ..., 0, 0, 0])
```

```
>>> # get the non-zero entries from this line
>>> l = [i for i, x in enumerate(a[0]) if x > 0]
>>> l
```

```
[0, 1, 2, 3, 8, 13, 23, 27, 87, 249, 367, 425, 445, 541, 1207,
1370, 1466]
```

```
>>> # look these up in the index
>>> print(", ".join(rindex[i] for i in l))
```

```
#, ., @, the, and, in, me, with, them, unhappy, making, service,
future, customer, Really, united, disappointed
```

In other words, the matrix is a sparse representation of a multi-set of the words in each tweet that occur more than **wthreshold** times in the entire dataset (words that don't occur very frequently tend not to be much help – they're unlikely to occur in the testset, and the assignment of an emotion to a very rare word is probably going to be wrong anyway: we set **wthreshold** in the params, with a default value of 1. Varying this will produce different overall scores).

- **model**: this can be set to a model for calculating word similarities, to be used for assigning emotions to unknown words, as described in Chapter 5.

- **scoredict**: this is a representation of how strongly each word is linked to each emotion, e.g. **'shocked': array([0., 0., 0., 1.])** says that every occurrence of *shocked* is associated with **sadness**, while **'and': array([0.2387, 0.2481, 0.2837, 0.2293,])** says that occurrences of *and* are roughly evenly distributed between the four emotions.

- **values**: this is a single number representing the emotions associated with a tweet, e.g. if the Gold Standard was **[0, 0, 0, 1]** then the value would **3**. This is required by some of the later classifiers, but **it will not work** if tweets can express more than one emotion.

CPCLASSIFIERS

LEXCLASSIFIERs just compare the normalised sums of scores for words that make up a sentence with a threshold and use that to decide whether a tweet expresses an emotion. Conditional probability classifiers (CPCLASSIFIERs) do a slightly more complicated sum with the same data (*Chapter 5, …*).

Comparing CPCLASSIFIERS and LEXCLASSIFIERs we get:

```
>>> x = everything(datasets=ENGLISH0, classifiers=[LEXCLASSIFIER, CPCLAS-
SIFIER], overrides=[{"stemmer": "none"}], showparams=False)
```

```
Scores for LEX
           Precision  Recall     micro F1   macro F1   Jaccard
SEM4-EN    0.636      0.747      0.687      0.700      0.523
SEM11-EN   0.461      0.598      0.521      0.521      0.352
WASSA-EN   0.574      0.683      0.624      0.632      0.453

Scores for CP
           Precision  Recall     micro F1   macro F1   Jaccard
SEM4-EN    0.734      0.780      0.756      0.762      0.608
SEM11-EN   0.457      0.619      0.526      0.523      0.357
WASSA-EN   0.615      0.770      0.684      0.699      0.520

All scores for datasets SEM4-EN,SEM11-EN,WASSA-EN using classifiers
CP,LEX
           CP         LEX
SEM4-EN    0.608      0.523
SEM11-EN   0.357      0.352
WASSA-EN   0.520      0.453
```

The attributes of a CPCLASSIFIER are identical to those of a LEXCLASSIFIER trained on the same data. The only difference lies in what the two do with the scores associated with each word – clearly the calculation carried out by the CPCLASSIFIER makes better use of the data.

Using models

One of the issues with the lexicon-based models is that they don't include all the words that are contained in the test data. Since some of the missing words might be emotion-carrying, it would be nice to somehow fill in the gaps. We can get the missing words as below (clex is LEXCLASSIFIER trained on SEM4-EN: other classifiers will give similar results):

```
>>> # make a set to contain all the distinct words in the training data
>>> traininglexicon = set()
>>> for tweet in clex.train.tweets: traininglexicon = traininglexicon.u-
nion(set(tweet.tokens))
>>> # how many are there?
>>> len(traininglexicon)
```

```
18453
```

```
>>> # get all the words in the test data

>>> testlexicon = set()

>>> for tweet in clex.test.tweets: testlexicon = 
testlexicon.union(set(tweet.tokens))

>>> len(testlexicon)
```

```
3824
```

```
>>> # get the words that are in the test data but not in the training 
data

>>> diff = testlexicon.difference(traininglexicon)

>>> len(diff)
```

```
1468
```

We see that 1468 of the words in the test data are not in the training data, i.e. 38%. Many of these are very rare words, and a good proportion are not emotion bearing, but there will be some that are significant for our task.
Can we fill in the missing words? Consider the word *obnoxious* (which is in **diff**). This is clearly a negative word, almost certainly expressing **anger**. Can we do anything with it?
There are a number of ways of computing similarity measures between words. Using one well-known one, find that the 10 most similar words to *obnoxious* are *annoying*, *egotistical*, *pompous*, *arrogant*, *overbearing*, *idiotic*, *whiny*, *impulsive*, *boorish*, *pushy*. If we look in the lexicon for our classifier, we find that some of these are indeed recorded as expressing emotions:

```
>>> for w in ['annoying', 'egotistical', 'pompous', 'arrogant', 'over-
bearing', 'idiotic', 'whiny', 'impulsive', 'boorish', 'pushy']:

...  if w in clex.scoredict: print(w, clex.scoredict[w])
```

```
annoying [0.25 0.25 0.5  0.   0.  ]

pompous [0. 1. 0. 0. 0.]

impulsive [0. 1. 0. 0. 0.]
```

On the basis of this, it looks as thought it would be reasonable to assume that *obnoxious* expressed something like [0.16666667 1.5 0.33333333 0.   0.], i.e. that it would vote strongly for emotion 1 and a bit for emotions 0 and 2. Unfortunately, the set of emotions that used in SEM-4 is ['**anger**', '**fear**', '**joy**', '**sadness**', '**neutral**'], so that means that the strongest emotion that we would associate with *obnoxious* is **fear**, with a little bit of **anger** and a little bit of **joy**.
There's an idea here that might be worth following up. It didn't work out all the well for the word that looked at, but maybe it will pay off in the end.
We will try two well-known similarity models, namely GLOVE (Pennington et al., 2014) and Word2Vec (Mikolov et al., 2013). To use GLOVE, download (and unzip!) https://nlp.stanford.edu/data/glove.6B.zip into <CORPORA>/DATA/GLOVE. To use Word2Vec, we use the **gensim** implementation (Řehůřek & Sojka, 2010), which you can install by doing

```
% pip install gensim
```

(this does require specific versions of various other things to be installed, so it may not work: given that using similarity models doesn't work out all that well for our task, it's not worth spending hours and hours getting it installed).
Once you've got **gensim** installed you have to supply it with a training set. In the experiments below we have trained it on 1M words from the BNC, as a reasonable compromise between waiting ages and getting good results:

```
>>> # W2VMODEL and GLOVEMODEL are wrappers for word2vec and Glove
>>> wmodel = W2VMODEL(N=1000000) # Train with 1M words from the BNC
```

```
reading corpus
took 2.0 mins 10 secs
making model
took 2.0 mins 5 secs
```

```
>>> gmodel = GLOVEMODEL()
>>> words = ["king", "man", "peach", "love", "happy", "sad"]
>>> for word in words: print("%s\t%s"%(models.shownearest(word, wmodel)))
```

```
king  ['charlemagne', 'emperor', 'throne', 'earl', 'prince']
man   ['woman', 'boy', 'girl', 'gentleman', 'soldier']
peach ['ginger', 'cherry', 'tiger', 'lacy', 'reed']
love  ['joy', 'sorrow', 'happiness', 'kindness', 'delight']
happy ['pleased', 'lucky', 'nice', 'quiet', 'disappointed']
sad   ['funny', 'strange', 'curious', 'pity', 'tragic']
```

```
>>> for word in words: print("%s\t%s"%(models.shownearest(word, gmodel)))
```

```
king  ['prince', 'queen', 'son', 'brother', 'monarch', 'throne', 'king-
dom', 'father', 'emperor', 'ii']
man   ['woman', 'boy', 'one', 'person', 'another', 'old', 'life', 'fa-
ther', 'turned', 'who']
peach ['apricot', 'pear', 'mango', 'raspberry', 'pecan', 'pumpkin', 'wa-
termelon', 'blueberry', 'plum', 'cherry']
love  ['me', 'passion', 'my', 'life', 'dream', 'you', 'always', 'wonder',
'i', 'dreams']
happy ["'m", 'feel', "'re", 'i', "'ll", 'really', 'glad', 'good', 'we',
'sure']
sad   ['sorry', 'awful', 'tragic', 'horrible', 'happy', 'heartbreaking',
'poignant', 'scary', 'terrible', 'unfortunate']
```

They both return some reasonable words and some odd ones (the word2vec model returns *funny* as the most similar word to sad, the Glove model returns quite a lot of irrelevant words for *happy*). You can try other models, or you can train word2vec on a larger dataset (but be warned: running it on larger subsets of the BNC doesn't make much difference to what it returns, but it does take longer).

What happens when we use these to look for similar words that do appear in the sentiment lexicon to fill in the gaps for words that don't? Using the word2vec model (which returns similar words much faster than the glove model) and printing out some of the substitutions that it makes and the emotions associated with them we get the following. We have marked places where the substituted word is similar to the target one in green and places where the substituted word is more-or-less the opposite of the target one in pink, and we have marked places where the substituted word does express the right emotion in green and ones where it strongly expresses something incorrect in pink.

```
>>> x = everything(datasets=ENGLISH0[0], classifiers=[CPCLASSIFIER],
overrides=[{"stemmer": "none", "model": wmodel}, {"stemmer": "none",
"model": False}])
```

```
embarrassed chosen for shocked: anger:0.00, fear:0.00, joy:0.72,
sadness:0.00, neutral:0.00

complains chosen for realizes: anger:0.00, fear:0.00, joy:0.00,
sadness:0.72, neutral:0.00

emulate chosen for capture: anger:0.00, fear:0.72, joy:0.00,
sadness:0.00, neutral:0.00

jealousy chosen for guilt: anger:0.72, fear:0.00, joy:0.00, sadness:0.00,
neutral:0.00

jeans chosen for hats: anger:0.00, fear:0.72, joy:0.00, sadness:0.00,
neutral:0.00

tournament chosen for marathon: anger:0.16, fear:0.00, joy:0.00,
sadness:0.16, neutral:0.00

reminiscent chosen for devoid: anger:0.00, fear:0.72, joy:0.00,
sadness:0.00, neutral:0.00

baskets chosen for bowls: anger:0.72, fear:0.00, joy:0.00, sadness:0.00,
neutral:0.00

boyfriend chosen for girlfriend: anger:0.00, fear:0.72, joy:0.00, sad-
ness:0.00, neutral:0.00

identify chosen for compare: anger:0.00, fear:0.00, joy:0.00,
sadness:0.72, neutral:0.00

adjusts chosen for lowers: anger:0.00, fear:0.00, joy:0.00, sadness:0.72,
neutral:0.00

worsening chosen for volatility: anger:0.00, fear:0.72, joy:0.00, sad-
ness:0.00, neutral:0.00

approved chosen for recommended: anger:0.00, fear:0.00, joy:0.00, sad-
ness:0.72, neutral:0.00

...

All scores for datasets SEM4-EN using classifiers CP-0,CP-1 # CP-0 used
the model, CP-1 didn't

           CP-0       CP-1

SEM4-EN    0.574      0.608
```

Two things stand out from this:

- using the model made things worse!
- For the model to be useful, the words it suggests have to be similar to the target words **and** the emotions associated with them have to be right.

It could be that things will be better if you use a better similarity model, but it doesn't seem all that promising. Get the best model you can (e.g. train word2vec on the entire BNC) and try it again, but it's probably not going to be a game-changer. In any case, as the training data gets larger the proportion of words that do appear in the test data will go up, so it may be that just having more training data will be more useful.

SKLEARN CLASSIFIERS

The classifiers above were all constructed by using a sentiment lexicon, either manually contructed or obtained from the training data, and doing some fairly simple calculations based on that. The next three all make use of standard machine learning algorithms. We use the SKLEARN implementations of these algorithms (Pedregosa et al., 2011).

Naive Bayes Classifiers (NBCLASSIFIERs, Chapter 6)

We make an NBCLASSIFIER by just reading the training and testing data and then using that to build a standard **sklearn.naive_bayes.MultinomialNB**:

```
class NBCLASSIFIER(sklearnclassifier.SKLEARNCLASSIFIER):


    def __init__(self, train, params={}):
        # Convert the training data to sklearn format
        self.params = params
        self.threshold = checkArg("threshold", params, None)
        self.readTrainingData(train, params=params)
        # Make a naive bayes classifier
        self.clsf = naive_bayes.MultinomialNB()
        # Train it on the dataset
        self.clsf.fit(self.matrix, self.values)
```

If we compare NBCLASSIFIERs and CPCLASSIFIERs (since they were the best of the lexicon-based classifiers) we get:

```
>>> x = everything(datasets=ENGLISH0, classifiers=[NBCLASSIFIER, CPCLAS-
SIFIER], overrides=[{"stemmer": "none"}], showparams=False)
```

```
All scores for datasets SEM4-EN,SEM11-EN,WASSA-EN using classifiers CP,NB
           CP         NB
SEM4-EN    0.608      0.787
SEM11-EN   0.357      0.274
WASSA-EN   0.520      0.708
```

Naive Bayes is better than CP on SEM4-EN and WASSA-EN, which have exactly one emotion per tweet, and worse on SEM11-EN, where tweets may have any number (including 0) of emotions. The problem is that the basic sklearn implementation of Naive Bayes assumes that every item has exactly one label. We discussed this issue at length in Chapter 10, and we will return to ways of dealing with it below.
An NBCLASSIFIER has an sklearn.naive_bayes.MultinomialNB() as its underlying classifier. We can use this to find the conditional probabilities of words for each emotion: these weights are how likely a text that contained just this word would be to express the given emotion – the underlying classifier does the proper calculations for complete texts.

```
>>> cnb.showAllWeights("love joy anger angry sorrow sadness hate")
```

```
           anger      fear       joy        sadness
love:      0.166      0.135      0.480      0.219
joy:       0.111      0.087      0.633      0.170
anger:     0.919      0.019      0.011      0.051
angry:     0.914      0.029      0.044      0.013
```

```
# sorrow wasn't in the training data
sorrow:    0.000      0.000      0.000      0.000
sadness:   0.027 0.   021        0.023      0.929
hate:      0.293      0.261      0.147      0.299
```

These all seem reasonable enough. Again, we have gaps in the lexicon, but as we saw just now using a word-similarity model to fill them in doesn't seem to help. We just have to get more (and more) data.

How much more data? In the book we plotted training size against Jaccard score and training time for some of the classifiers, because if you find that Jaccard score flattens out after you've looked a certain amount of data then it may that getting more (and more) data isn't actually worth the bother.

We can do a plot as follows. We start by setting up a sequence of overrides where we've chosen to increase the value of **N** (i.e. the size of the training data) from 1000 to 60000 where we multiply each successive value of **N** by 1.5, using a set of overrides initialised to {**"useneutral": True, "justone": False}**). We could vary a different parameter, e.g. the size of the dictionary (**maxdictsize**) or the number of iterations (**max_iter**), we could obviously change the start, end and step size, and we could use a different initial set of overrides.

```
>>> overrides = makeoverrides(label="N", start=1000, end=60000, step=1.5, override0={"useneutral": True, "justone": False})
>>> overrides
```

```
[{'N': 1000, 'useneutral': True, 'justone': False},
 {'N': 1500, 'useneutral': True, 'justone': False},
 {'N': 2250, 'useneutral': True, 'justone': False},
...
 {'N': 25628, 'useneutral': True, 'justone': False},
 {'N': 38443, 'useneutral': True, 'justone': False},
 {'N': 57665, 'useneutral': True, 'justone': False}]
```

We then run our classifier on our chosen dataset (we've used the CARER dataset here because it's one of the biggest so it's the most likely to show trends):

```
>>> plotnb = everything(datasets=ENGLISH[-2], classifiers=NBCLASSIFIER, overrides=overrides)
```

The raw scores do show that increasing the size of the training data leads to improved performance, but it does also look as though the improvement is flattening out:

```
           NB-0  NB-1  NB-2  NB-3  NB-4  NB-5  NB-6  NB-7  NB-8  NB-9  NB-10
CARER-EN   0.267 0.326 0.432 0.534 0.595 0.647 0.680 0.701 0.719 0.730 0.732
```

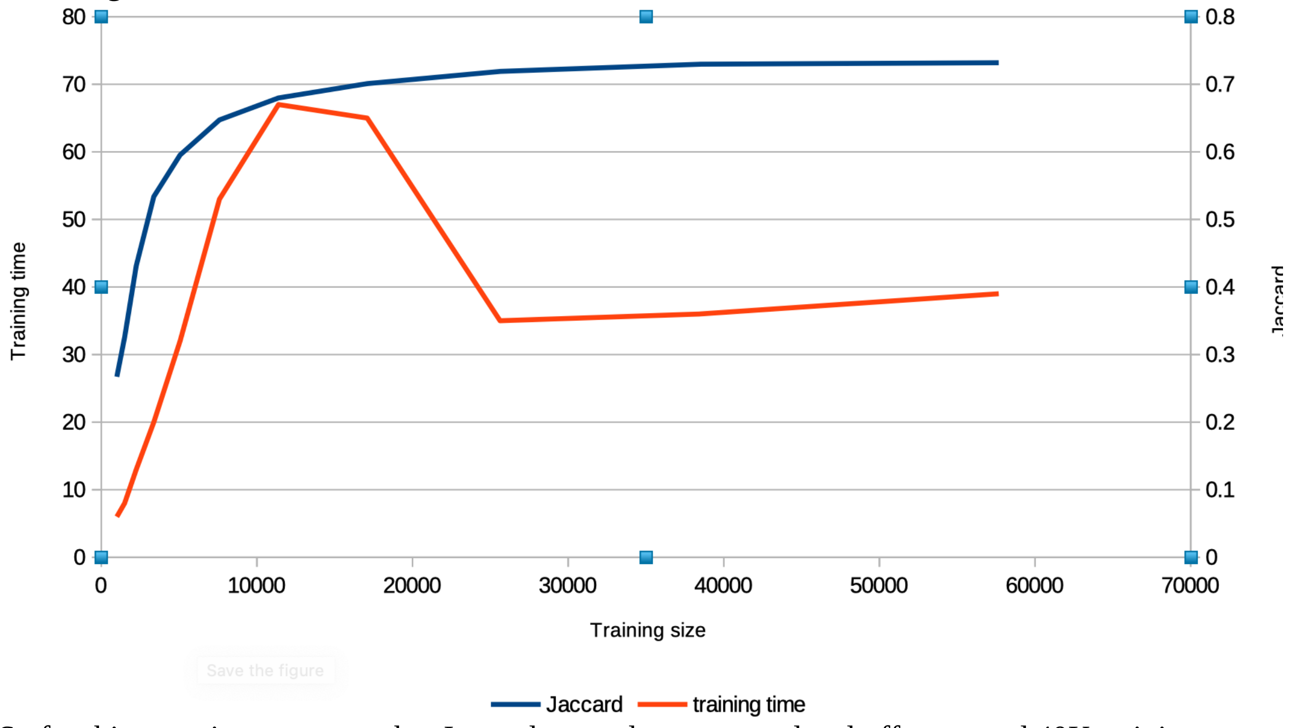
We can get a more visual representation of this by outputting the various scores as tab-separated columns, reading this into a spreadsheet and plotting the curves (we used the LibreOffice CALC for this, but Microsoft EXCEL or anything else you like):

```
>>> makeplots(plotnb, overrides)
```

```
size       Jaccard              accuracy             train      test
1000       0.2671990171990172   0.4217159476490548   6          0.003
1500       0.32562442183163737  0.49127704117236565  8          0.003
…
```

```
17085        0.7009256025148446    0.8241696185635813    65          0.003
25628        0.7190097259062777    0.836539450673799     35          0.001
38443        0.7297015961138098    0.8437311935807422    36          0.001
57665        0.7319550882195687    0.8452356451944845    39          0.001
```

Plotting the columns headed Jaccard accuracy and Training time against Training size we get the following:



So for this experiment we see that Jaccard score does seem to level off at around 40K training tweets, but the training time doesn't seem to actually be connected to size of the datasets. For other classifiers the time does go up, linearly or worse (see Chapter 7 on SVMs), but for Naive Bayes it seems to be pretty well swamped by other effects.

We can also plot Jaccard against the size of the dictionary: this time we vary the maxdictsize from 100 to 2000 (which is the number of distinct words in the CARER dataset that occur more than once):

```
>>> overrides = makeoverrides(label="maxdictsize", start=100, end=20000,
step=1.5, override0={"useneutral": True, "justone": False})

>>> printall(overrides)
```

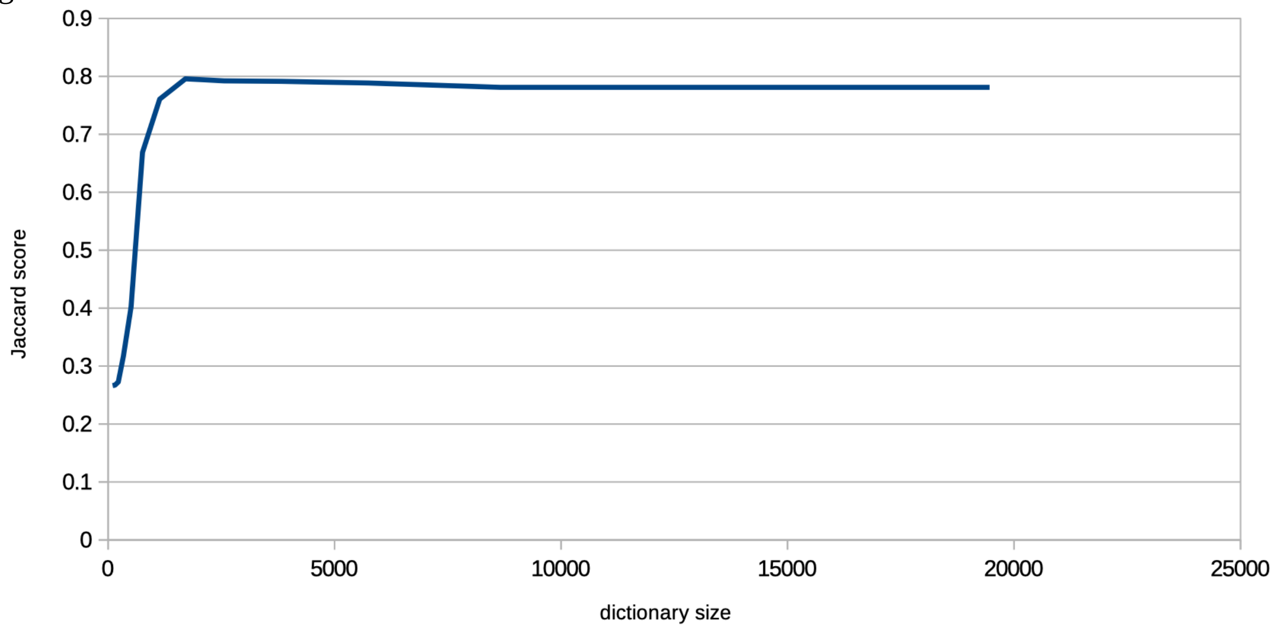
```
{'maxdictsize': 100, 'useneutral': True, 'justone': False}

{'maxdictsize': 150, 'useneutral': True, 'justone': False}

...

{'maxdictsize': 12974, 'useneutral': True, 'justone': False}

{'maxdictsize': 19461, 'useneutral': True, 'justone': False}
```

```
>>> plotdictsize = everything(datasets=ENGLISH[-2], classifiers=NBCLASSI-
FIER, overrides=overrides)

>>> makeplots(plotdictsize, overrides, label="maxdictsize")
```

```
dsize Jaccard                  accuracy    training time    testing time

100   0.2663194756218012     0.42061972629936895   37     0.001
```

```
150   0.26714833645526714   0.4216528227509503    40    0.001
...
12974 0.7809859154929577    0.8770264926848557    49    0.002
19461 0.7809859154929577    0.8770264926848557    49    0.002
```

And then when we copy that into a spreadsheet and plot dictionary size against Jaccard score we get:



This is quite revealing. When we just look at the 100 or 150 commonest words the score is pretty poor, but once we get up to about 2K words the score flattens out and even starts to decrease. Very common words don't carry much information about emotions, but if you include ones that only occur once or twice the classifier starts to overtrain.
Experiments of this kind can be very revealing. If there's a numerical parameter which you think might affect the performance in some way, use makeoverrides to generate an experiment and try it out.

Support Vector Machines (SVMCLASSIFIERS, Chapter 7)

We now look at the SVMCLASSIFIERS from Chapter 7. We start by comparing them CPCLASSIFIERs and NBCLASSIFIERS

```
>>> x = everything(datasets=ENGLISH0, classifiers=[NBCLASSIFIER, CPCLAS-
SIFIER, SVMCLASSIFIER])
```

```
All scores for datasets SEM4-EN,SEM11-EN,WASSA-EN using classifiers
CP,NB,SVM
             CP          NB          SVM
SEM4-EN      0.628       0.787       0.834
SEM11-EN     0.360       0.278       0.259
WASSA-EN     0.551       0.710       0.741
```

We can see that the SVM is the best on the single-label datasets and the worst on the multi-label one – see Chapter 7 for more about how SVMs work and why they might be poor at dealing with multi-label datasets, see below for more about how to deal with multi-label datasets.
The definition of SVMCLASSIFIERs is almost identical to that for NBCLASSIFIERs:

```
class SVMCLASSIFIER(sklearnclassifier.SKLEARNCLASSIFIER):
    def __init__(self, train, params={"useDF":True}):
        self.readTrainingData(train, params=params)
        self.params = params
        max_iter = checkArg("max_iter", self.params, 20000)
        self.clsf = sklearn.svm.LinearSVC(max_iter=max_iter)
        self.clsf.predict_proba = self.clsf.decision_function
        self.clsf.fit(self.matrix, self.values)
        self.weights = self.clsf.coef_
```

The only differences are that we have to specify the maximum number of iterations for the training phase, since the SVM learning algorithm is iterative; that we make a sklearn.svm.LinearSVC rather than a sklearn. naive_bayes.MultinomialNB (obviously enough); and that extract the coefficients from the trained classifier to use as the weights for individual words.
As with NBCLASSIFIERS we can see the weights for each emotion for individual words:

```
>>> csvm = x['SVM']['SEM4-EN'][2][0]
>>> csvm.showAllWeights("love joy anger angry sorrow sadness hate")
```

```
          anger     fear      joy       sadness
love:     -0.165    -0.177    0.239     0.024
joy:      -0.287    -0.268    0.533     -0.242
anger:    1.776     -0.912    -0.845    -0.930
angry:    1.831     -1.208    -0.582    -1.405
sorrow:   0.000     0.000     0.000     0.000
sadness:  -0.608    -1.041    -0.938    1.790
hate:     0.153     -0.068    0.074     0.015
```

The most notable difference between these weights and the NBCLASSIFIER ones is that SVMCLASSIFIER can assign negative weights – that *love* can vote against **anger** as well as for **joy**, that *angry* can vote against **fear**, **joy** and **sadness** as well as for **anger**.

Deep Neural Nets (DNNCLASSIFIERS, Chapter 8)

Finally in this section we consider deep neural net classifiers. We will just compare them with SVMCLASSIFIERs.

```
>>> x = everything(datasets=ENGLISH0, classifiers=[SVMCLASSIFIER,
DNNCLASSIFIER])
```

```
All scores for datasets SEM4-EN,SEM11-EN,WASSA-EN using classifiers
DNN,SVM

          DNN       SVM
SEM4-EN   0.839     0.834
SEM11-EN  0.253     0.259
WASSA-EN  0.752     0.741
```

The DNNCLASSIFIERs marginally outperform the SVMCLASSIFIERS on the single-label cases, where SVMCLASSIFIERs were already our best classifiers, and they do worse on the multi-label case, where SVMCLASSIFIERs did not outperform the lexicon-based classifiers. They do, however, take much longer to train and quite a bit longer to apply: if we get the lists of classifiers for all the folds for DNNCLASSIFIERs and SVMCLASSIFIERs for SEM4-EN we see that the DNNCLASSIFIERs took roughly ten times as long to train as the SVMCLASSIFIERs, and ran at about 60% of the speed of SVMCLASSIFIERs when we applied them to the test sets.

```
>>> clsfsvmlist = x['SVM']['SEM4-EN'][2]

>>> clsfdnnlist = x['DNN']['SEM4-EN'][2]

>>> timings(clsfsvmlist)
```

```
training time 46.419 sec (120 tweets/sec), testing time 0.0016 sec
(382595 tweets/sec)
```

```
>>> timings(clsfdnnlist)
```

```
training time 537.936 sec (10 tweets/sec), testing time 0.0024 sec
(262000 tweets/sec)
```

They both run reasonably fast when applied to the test set (260K words/sec is good enough for most purposes). Is the marginal increase in accuracy worth the 10-fold increase in training time? Maybe, because in practice you will only train a classifier once, but you will probably want to apply it many times.

DNNs have, by their very nature, layers between the input and output layers. You can specify the number and size of the hidden layers (we assume that every layer is fully connected to the following one), either by specifying it explicitly or by providing a function that will derive it from the properties of the dataset (e.g. the number of distinct words in the training data or the number of output layers).

sklearn.neural_network import MLPClassifier, which are what we use for our DNNs, requires the layout of the hidden layers to be supplied as a tuple of integers, where each integer specifies the number of nodes in a layer, e.g. (3,5,2) would be a DNN with three hidden layers with 3, 5 and 2 nodes. It is very easy to get drawn into trying out different configurations of hidden layers, and to think that adding more layers or more nodes per layer will make things work better. **Don't do it**: you can waste a huge amount of time messing around, and you won't get anywhere unless you think carefully about what the hidden layers are contrbuting and why having more, or having more nodes in a given layer, might improve things. Don't just assume that a network with 100 hidden layers with 1000 nodes each will be better than one with one hidden layer with 15 hidden nodes. Think about it before you do it.

It can be useful to link the size or number of hidden layers to some property of the training data. We therefore make it possible to supply an explicit tuple like (3, 5, 2) or a function that will create such a tuple based on properties of the training data, e.g. **`lambda d: (len(d.training.emotions)*1.5)`** would create a single hidden layer with 1.5 times as many nodes as the number of emotions (this one works quite well). We will compare using this function with using a simple DNN with no hidden layers (DNN-0 is the version that has a hidden layer with 1.5 times as many nodes as there are emotions in the dataset, DNN-1 is the version with no hidden nodes):

```
>>> x = everything(datasets=ENGLISH0, classifiers=[DNNCLASSIFIER], over-
rides=[{"hiddenlayers": lambda clsf: int(len(clsf.train.emotions)*1.5),
"folds":5, "N":10000}, {"hiddenlayers": (), "folds":5, "N":10000}])
```

```
All scores for datasets SEM4-EN,SEM11-EN,WASSA-EN using classifiers DNN-
0,DNN-1

          DNN-0      DNN-1

SEM4-EN    0.789      0.755

SEM11-EN   0.285      0.257
```

```
WASSA-EN   0.735      0.620
```

There are a million experiments you can do with different numbers of hidden layers and different numbers of nodes per hidden layer. But as you get more and larger hidden layers, things start to get very slow. Really very slow. So think about what you're doing and why you think some particular configuration of hidden layers will be good before you actually try it out.

As with the previous classifiers, we can look at the connections between individual words and emotions. This is, however, now more difficult to interpret than before: in particular, if our DNN has hidden layers then to understand how a word is connected to an emotion you would have to understand what the nodes in the hidden layer are doing. We can ask our two classifiers how strongly a word *by itself* would suggest a given emotion, but particularly where our DNN has hidden layers that doesn't actually tell us how to calculate what the classifiers would predict for a *combination* of words.

If we ask the two classifiers (remember, dnn0 is the one with a hidden layer, dnn1 is the one without) what are the words with the strongest links to each emotion we get the following:

```
>>> for e in dnn0.train.emotions: print("%s: %s"%(e, ", ".join(dnn0.get-
StrongLinks(e)[:5])))
```

```
anger: bitter, rage, fuming, revenge, burning

fear: nightmare, panic, shocking, awful, bully

joy: optimism, hilarity, glee, cheering, playful

sadness: unhappy, sober, blues, depressing, grim
```

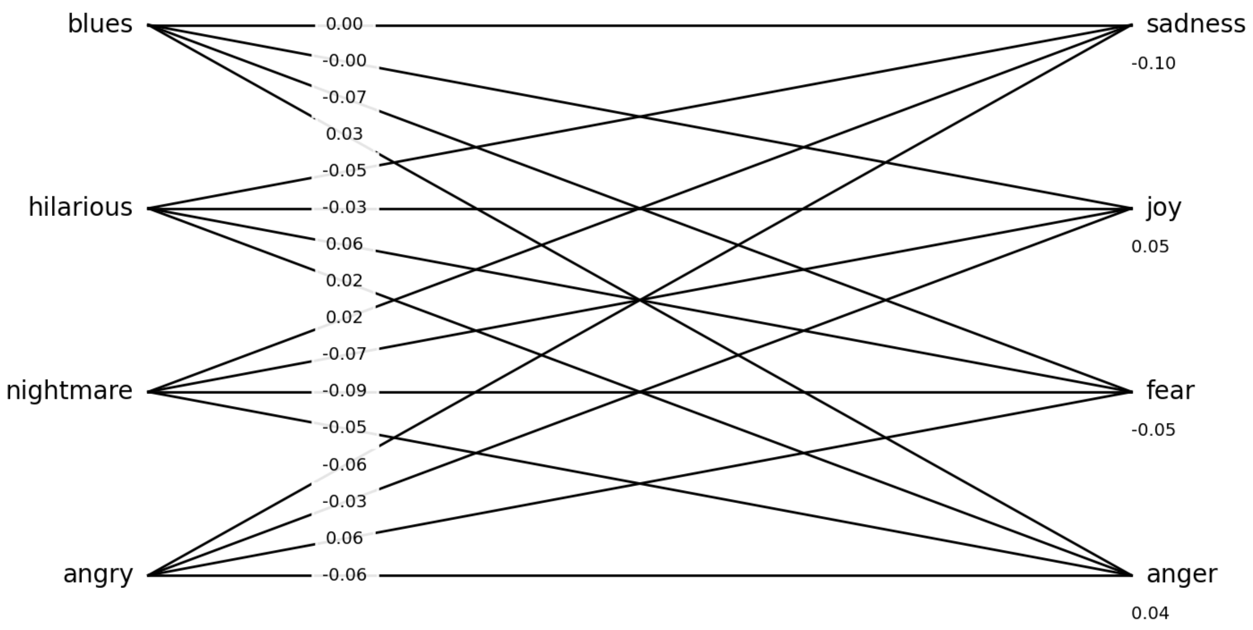
```
>>> for e in dnn1.train.emotions: print("%s: %s"%(e, ", ".join(dnn1.get-
StrongLinks(e)[:5])))
```

```
anger: angry, bitter, rage, anger, fuming

fear: nightmare, awful, nervous, terrible, terrorism

joy: hilarious, optimism, smile, happy, glee

sadness: blues, sober, lost, sadness, sad
```

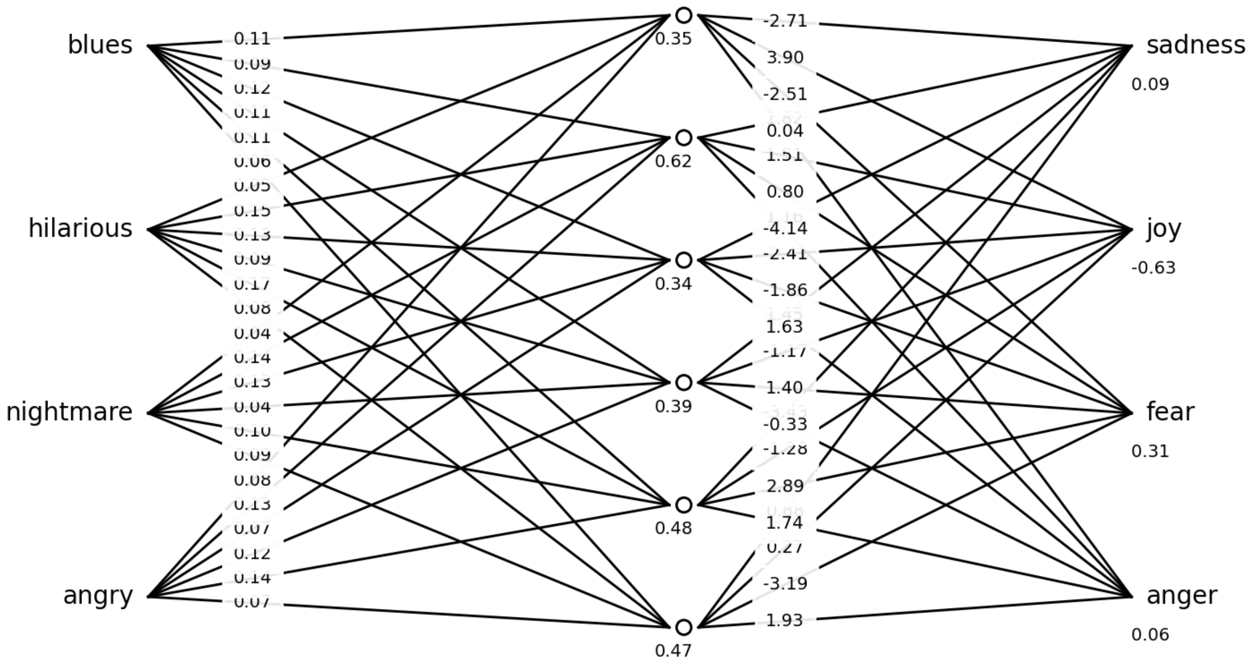
In other words, they both give us a fairly sensible set of words which by themselves would suggest the various emotions. But that doesn't really tell us what they would do with sentences containing these words.

We can display the parts of the actual networks for specified words. We will look at the one for the network with no hidden layer first, and we'll use the words that it says are most strongly linked to the emotions:

```
>>> dnn1.showdnn("angry nightmare hilarious blues".split())
```

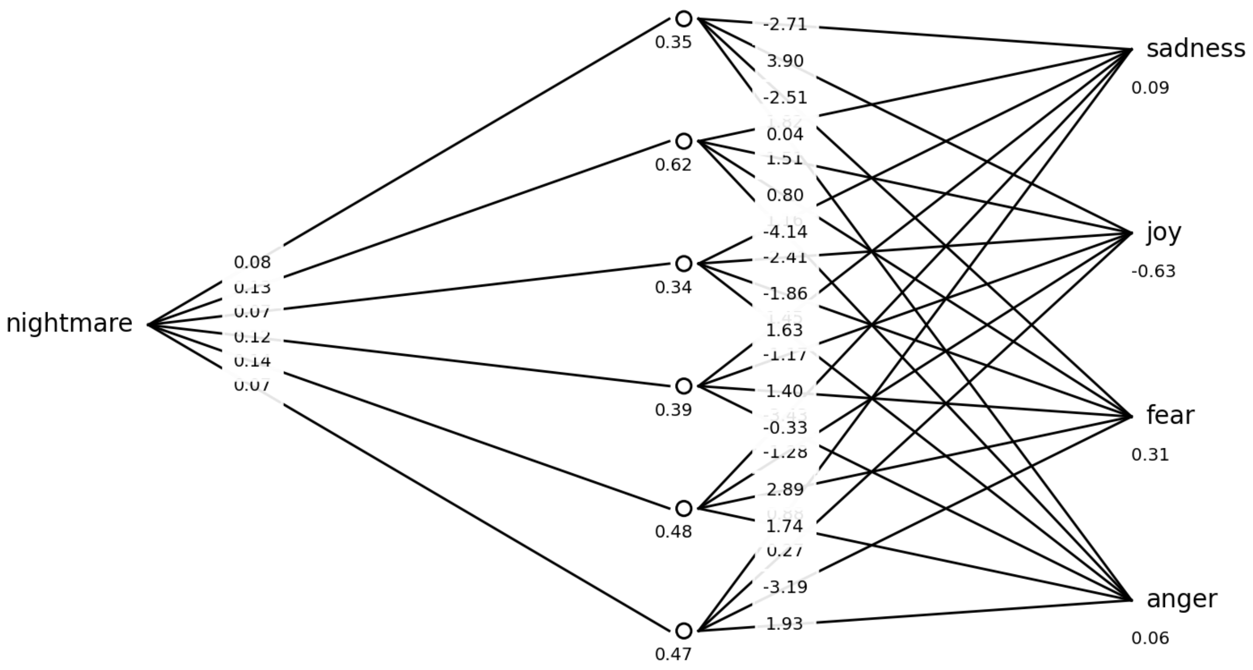


Some of these connections look reasonable, some don't. It's interesting enough having a look at the connections, but they don't really tell you as much as you would like. It's even worse when we look at the one with hidden layers: words are connected to nodes in the hidden layer, nodes in the hidden layer are connected to emotions, there's a huge amount of information here and it's all pretty well impossible to make sense of it.



Even if we just look at one word it's extremely difficult to see what it all means:

```
>>> dnnclassifier.showdnn(dnn0, "nightmare".split())
```



You can sort of interpret what the hidden nodes are doing. The top one has a positive connection to **joy** and negative connection to the others, so it is a sort of representation of **joy** and it says that **joy** is incompatible with the other emotions. The bottom one has positive connections to **anger** and **sadness**, and a strongly negative connection to **fear**, so words that vote for this one are in favour of both **anger** and **sadness** and opposed to **fear**. But it's all a bit approximate and unclear. DNNs are notoriously opaque: they quite often work quite well, but seeing exactly why they do what they do is basically impossible.

Multi-label classifiers (Chapter 10)

Chapter 10 described three ways of trying to deal with multi-label datasets:

- allow neutral as a label: to do this we just set **useneutral** to **True**
- use thresholds (preferably local thresholds) to allow a classifier: to do this we set **justone** to **False** and **bestthreshold** to "**global**" or "**local**"
- train a set of X vs not X classifiers and allow them each to make independent suggestions: use a MULTICLASSIFIER and set what to use as its subclassifiers

We can try this with Naive Bayes: we start with an experiment where we try Naive Bayes on the multi-label datasets (MULTI) with and without **useneutral**, with **justone** set to **True** or **False**, and **bestthreshold** set to **global** or **local**. We make a set of overrides and use those with all the datasets:

```
>>> overrides=[{"useneutral": False, "justone": True},

               {"useneutral": True, "justone":True},

               {"useneutral": True, "justone":False,

                "bestthreshold": "global"},

               {"useneutral": True, "justone":False,

                "bestthreshold": "local"}]

>>> nbmulti = everything(datasets=MULTI, classifiers=[NBCLASSIFIER],
overrides=overrides)
```

```
           NB-0       NB-1       NB-2       NB-3

SEM11-EN   0.230      0.228      0.283      0.257
```

```
SEM11-AR   0.191      0.191      0.263      0.234

SEM11-ES   0.223      0.218      0.249      0.200

KWT.M-AR   0.199      0.226      0.272      0.254
```

In every case, NB-2 (which is **useneutral: True, justone:False, bestthreshold: "global"**) is is the best. This is slightly surprising, since you might expect (I did) that **useneutral: True, justone:False, bestthreshold: "local"** would be better), but that's the point of doing experiments. If we try using a MULTICLASSIFIER, using NBCLASSIFIER as the sub-classifier (I.e, we train a collection of NBCLASSIFIERS, one for angry vs not-angry, one for happy vs not-happy, …) we get the following:

```
multi = everything(datasets=MULTI, classifiers=MULTICLASSIFIER, over-
rides=[{"subclassifiers": NBCLASSIFIER}])
```

```
           MULTI-NB

SEM11-EN   0.437

SEM11-AR   0.338

SEM11-ES   0.289

KWT.M-AR   0.274
```

These are all better than the ones where we use a single NBCLASSIFIER with different options. Again, we can try all sorts of variants (use a different subclassifier, try using or not using neutral, do stemming or don't do stemming, …, …, …)

And that's it. The book ended with

Always, always, always, try out different classifiers, with different settings, and make your own decision about which works best for your task on your data.

The code in the repository is there to help you do that. Have fun.

References

Mikolov, T., Chen, K., Carrado, G., & Dean, J. (2013). *Efficient Estimation of Word Representations in Vector Space* (1st ed.). http://arxiv.org/pdf/1301.3781.pdf

Mohammad, S. M., & Turney, P. D. (2013). Crowdsourcing a Word-Emotion Association Lexicon. *Computational Intelligence*, *29 (3)*, 436–465.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Pennington, J., Socher, R., & Manning, C. (2014). GloVe: Global Vectors for Word Representation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 1532–1543. https://doi.org/10.3115/v1/D14-1162

Řehůřek, R., & Sojka, P. (2010). Software Framework for Topic Modelling with Large Corpora. *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, 45–50.