Python Exam Assignment

Computational Linguistics, Fall semester 2014 [Summer re-examination]

All questions: Mike.Kestemont@uantwerp.be (CC: Walter.Daelemans@uantwerp.be)

For this exam assignment, you will have to implement IB1, a classic algorithm which is commonly used for automatic classification in computer science (Aha, Kibler, and Albert, 1991). In classification, the idea is to train algorithms that can automatically assign items to a category. When dealing with texts, a good example would be spam filtering: based on the presence of specific words (e.g. "lottery"), the algorithms in your mailbox decide whether a new incoming email should be classified to the category "spam mail" or "regular email". These categories are often also called classes (hence "classification"). IB1 is a "learning algorithm": by having the algorithm look at examples, we have it "learn" how it can best generalize its classification behaviour to new examples. This example data is called training data; after the training phase, we can test what the algorithm has learned, by testing its performance on a test data set.

The project below is deliberately complex and it is normal if you experience difficulties in implementing the procedures described. We provide step-by-step instructions that will help you to implement IB1 from the bottom up. The exercises focus on the diminutives classification task (Daelemans, Berck, and Gillis, 1997), in which we will automatically try to predict the correct diminutive suffix for Dutch words (e.g. boom+pje vs aap+je vs. man+netje). This exam exercise takes its inspiration from TiMBL (Daelemans et al., 2010), a computer program for classification which was partly developed by the computational linguistics group at the University of Antwerp. You can always consult the extensive manual to this software for more information on the diminutives classification problem, or more details on the IB1 algorithm (esp. pp. 11-12). Three files accompany this assignment:

(1) the training data (`dimin.train`)
(2) the test data (`dimin.test`)
(3) the TiMBL manual

Hand in your Python code electronically before **Thursday 27 August 2015** by sending it attached to an email to `Mike.Kestemont@uantwerp.be`.

Ex. 1.
The file `dimin.train` contains the data which we will use for training:

```
=,=,=,=,+,k,e,=,-,r,@,l,T
=,=,=,=,-,fr,i,=,+,z,I,n,E
=,=,=,=,=,=,=,=,+,sn,},f,J
=,=,=,=,+,l,I,=,-,x,a,m,P
=,=,=,=,=,=,=,=,+,tr,A,p,J
…
```

On each line, we find the representation of a single training item. Each training item represents a Dutch noun. In the TiMBL manual, we can read more about the format and task of diminutive form prediction. Each noun is represented in terms of its syllable structure: for each of the last three syllables of the noun, four different characteristics are encoded: whether the syllable is stressed or not (values - or +), the string of consonants before the vocalic part of the syllable (i.e. its onset), its vocalic part (nucleus), and its post-vocalic part (coda). Whenever a feature value is not present (e.g. a syllable does not have an onset, or the noun has less than three

syllables), the value '=' is used. The class to be predicted is either E (-etje), T (-tje), J (-je), K (-kje), or P (-pje). The correct class label (i.e. the diminutive form) is always the last item on the line. A number of examples:

+	b	i	=	-	z	@	=	-	m	A	nt	J	*biezenmand*
=	=	=	=	=	=	=	=	+	b	I	x	E	*big*
=	=	=	=	+	b	K	=	-	b	a	n	T	*bijbaan*
=	=	=	=	+	b	K	=	-	b	@	l	T	*bijbel*

The values or fields on each line are separated by a comma in `dimin.train`. Write a function `load_file()` that takes two parameters: the name of the file (`filename`) we would like to load (in this case `dimin.train`) and the string (`sep`) used to separate items (in this case a comma). This function should read the contents from the specified file and return a tuple of tuples: a tuple representation of the items in dimin.train, in which each item is represented as a tuple of the fields on the original line. Make sure to test that your function can also handle files in which the field values in each line are separated by tabs and semicolons (e.g. `=;=;=;=;+;k;e;=;-;r;@;l;T`). Before returning the result, you should check (i) whether there were any items at all in the file and (ii) whether each line has the same number of fields. If not, the function should raise a meaningful `IOError` which explains to the user of the function what went wrong.

Ex. 2.
The last field on each line in this data format represents the class label with which each training item has been annotated, in this specific case the form of the diminutive. Write a function `print_stats()` which prints statistics about the class label distribution in a data set. As parameters, it should take the tuple of tuples, as well as the index of the class label. If this last parameter is not specified, the function should default to the last element in each tuple as the class label's index. The function should calculate and print (in a pretty fashion):

- The number of items
- The number of features
- The unique class labels (in alphabetical order)
- The relative proportion of the individual class labels.

If there is only a single class label in the data set, the function should again raise a meaningful `ValueError` which explains what went wrong.

Ex. 3.
Apply the previous two functions to load and inspect the test data (`dimin.test`) we will use for the diminutives task. To see how well our algorithm can predict the diminutive forms in the test data on the basis of what it has seen in the training data, we now turn to IB1. IB1 is based on a intuitive form of "nearest neighbour" reasoning: to determine the class label of a new item, it will first calculate which previously seen word in the training set is most similar to the new test item (i.e. its "nearest neighbour"). Next, it will extrapolate the class label from that nearest neighbour to the test item. Thus, if a new word is closest to a training item from class "B", it will classify the new item as belonging to class B too.

To calculate the distance between two items, we simply calculate how many features are not shared between them. Consider the following items:

```
(a) =,=,=,=,+,l,I,=,-,x,a,m,P
(b) =,=,=,=,+,l,I,=,-,x,a,t,P
(c) =,=,=,=,+,l,I,=,-,x,o,t,P
```

The distance between (a) and (b) should be 1 (because the final feature does not match) and the difference between (a) and (c) should be 2 (because now there are two features that don't overlap). Write a function `predict()` which takes as input the tuple of train items and the tuple of test items. For each test item, you will first have to calculate to which training item the test item is closest (i.e. the smallest distance). Make absolutely sure that you don't include the class labels of the train and test items in your calculation, because that would be cheating! Note, that in pseudo-code, this function will need multiple nested for loops:

```
for test_item in test_items:
      …
      for train_item in train_items:
            …
            for feature in features:
                  …
```

Feel free to create auxiliary functions for this larger function. In the end, the function should return a tuple of the predicted class labels for the test data. Therefore make sure that the number of predictions returned is equal to the number of test items passed (if not, raise a meaningful `ValueError`). Additionally, implement the following shortcut: if along the way, you find a nearest neighbour among the training items at distance 0, you have in fact found an exact match. In that case, you can simply extrapolate the label from the exact math, and skip the rest of loop.

Ex. 4

It is now time to evaluate the accuracy of our learning system. Write a function `evaluate()` that compares the predictions outputted by `predict()` to the correct class labels of the test items. The function should return the average number of correct predictions made by the system (i.e. the number of correct predictions divided by the total number of predictions). Additionally it should print the accuracy for each class label separately, so that which can inspect which class labels are hardest to learn.

Ex. 5:

Our system above only considers the single nearest neighbour of a new instance it to classify. In this set-up, we could there say that `n` (the number of neighbours) is set at 1. Studies have shown that it is often interesting to inspect a larger "neighbourhood" and take into account the class information of multiple nearest neighbours (e.g. n=3). Re-write the `predict()` function in ex. 3 so that it accepts a new parameter `n`, specifying how many nearest neighbours the algorithm should taker into account (the default should be 1). For each individual classification, you will now have to implement a "majority vote" among the nearest neighbours. Say that we have determined the five nearest neighbours for a particular test item. One of these belongs to class A, three to class B, and one to class C. In this case, we will choose label B for the test item, because it is the class with the highest vote in the neighboourhood. Use

the `evaluate()` function from ex. 4 to experimentally compare the effect of increasing `n` on the test accuracy.

References

Aha, D. W., D. Kibler, and M. Albert. 1991. Instance-based learning algorithms. *Machine Learning*, 6:37–66.

Daelemans, W., P. Berck, and S. Gillis. 1997. Data mining as a method for linguistic analysis: Dutch diminutives. *Folia Linguistica*, XXXI(1–2):57–75

Daelemans, W., Zavrel, J., Van der Sloot, K., and Van den Bosch, A. (2010). *TiMBL: Tilburg Memory Based Learner, version 6.3, Reference Guide*. ILK Research Group Technical Report Series no. 10-01.