Natural Language Processing – Assignment II

Reykjavik University – School of Computer Science

October 2015

1 NLTK: The Icelandic Frequency Dictionary – 25%

In this part, you use Python and NLTK to process a PoS-tagged Icelandic corpus, the *Icelandic Frequency Dictionary* (*IFD*; "Íslensk Orðtíðnibók").[1] A preprocessed version of the IFD containing one sentence per line, is available as the file *otb.slash.sent*, from `http://www.ru.is/~hrafn/courses/nlp/ifdpenn.zip`.[2]

This part is divided into the subparts described below, but you should return a single Python program, **ifd.py**, which includes all the code needed for carrying out these tasks.

Write code to:

1. Read in the IFD using the class *TaggedCorpusReader*, display the number of sentences, and display the individual tokens of sentence nr. 50.
2. Display the number of tokens and the number of types in the IFD.
3. Display the 10 most frequent tokens in the IFD using the class *FreqDist*.
4. Display the 20 most frequent tags in the IFD using the class *FreqDist*.
5. Generate tag bigrams and use the class *ConditionalFreqDist* to print out the 10 most frequent tags that can follow the tag 'ao' (this tag denotes a preposition which govern the accusative case).

Example output follows:

```
Number of sentences: 36922
Sentence no. 50:
landið varð ekki lengur umflúið .

Number of tokens: 590300
Number of types: 59359

The 10 most frequent tokens
. => 33182
og => 22213
, => 22083
að => 21012
í => 15319
á => 12450
hann => 8040
var => 7905
sem => 7676
er => 6362

The 20 most frequent PoS tags:
```

[1] See `http://www.malfong.is/index.php?pg=ordtidnibok&lang=en`

[2] Please make sure not to distribute the IFD corpus.

```
AA => 50423
C => 42507
Aþ => 33268
. => 33182
SFG3Eþ => 27730
, => 22083
AO => 21728
SNG => 15258
SFG3EN => 13447
CN => 11558
SSG => 6987
FPKEN => 6977
AE => 6328
NKEN-S => 6213
CT => 6204
FP1EN => 6047
NKEN => 5651
NHEþ => 5628
SFG3Fþ => 5626
NVEO => 5478

The 10 most frequent PoS tags following the tag 'ao':
NVEO => 1753
NKEO => 1681
NHEO => 1579
NKEOG => 1538
NHEOG => 1399
NVEOG => 1055
CN => 608
NVFO => 596
FPKEO => 553
C => 535
```

2 NLTK: PoS tagging – 25%

In this part, you experiment with various different types of PoS taggers in the NLTK using the Penn Treebank corpus (accessible in Python with `from nltk.corpus import treebank`).

By studying chapter 5 in the NLTK book (`http://nltk.org/book/`), you should have all the necessary material to solve this part. The Python module NLTK documentation, `http://nltk.org/api/nltk.tag.html#module-nltk.tag` is also an important source of information.

This part is divided into the subparts described below, but you should return a single Python program, **tagging.py**, which includes all the code needed for carrying out these tasks.

Write code to:

1. Split the tagged sentences of the Penn Treebank into a training set (first 3500 sentences) and a test set (the remaining sentences), print out the total count of each set, and print the first sentence in the test set.

2. Construct four taggers trained on the training set: an instance of an *AffixTagger*, *UnigramTagger*, *BigramTagger* and a *TrigramTagger* (without any "backoff" model). Evaluate them on the test set, and print out the evaluation results.

3. Construct the latter three taggers again, but now with a backoff model, i.e. such that the trigram tagger uses a bigram tagger as backoff, which in turn uses a unigram tagger as backoff, which in turn uses the affix tagger as backoff. Print the evaluation results again.

4. Tag the test set with the main ("off-the-shelf") tagger in the NLTK[3], evaluate its accuracy and print out the result. Note that for this tagger you cannot simply call a built-in evaluation function, instead you have to write your own, which compares the results of the tagger to the gold standard. You will notice that the tagging accuracy for this tagger is surprisingly high. **What do you think is the reason**?

Example output follows:

```
 Number of training sentences: 3500
 Number of test sentences: xxx

 First sentence in test corpus:
[('About', 'IN'), ('30', 'CD'), ('%', 'NN'), ('of', 'IN'), ('Ratners', 'NNP'), ("'s", 'POS'),
('profit', 'NN'), ('already', 'RB'), ('is', 'VBZ'), ('derived', 'VBN'), ('*-1', '-NONE-'),
('from', 'IN'), ('the', 'DT'), ('U.S.', 'NNP'), ('.', '.')]

 Tagging accuracies:
 -------------------
 Affix tagger: xx.yy%
 Unigram tagger: xx.yy%
 Bigram tagger: xx.yy%
 Trigram tagger: xx.yy%

 Tagging accuracies with backoff:
 --------------------------------
 Affix tagger: xx.yy%
 Unigram tagger: xx.yy%
 Bigram tagger: xx.yy%
 Trigram tagger: xx.yy%

 Accuracy of the main tagger in NLTK: xx.yy%
```

3 IceNLP: Hidden Markov Model (HMM) tagging – 25%

In this part, you experiment with a HMM tagger, *TriTagger*, the trigram tagger which is part of the *IceNLP* toolkit. Before you start experimenting with the tagger, you need to do the following:

1. Read the paper "TnT – A Statistical Part-of-Speech Tagger"[4] – `TriTagger` is a re-implementation of the `TnT` tagger.

2. Download the latest version of *IceNLP* from `http://sourceforge.net/projects/icenlp/files/`, and extract to a directory of your choice.

3. Read the section on *TriTagger* in the user manual *IceNLP.pdf* (available in the /doc directory of the IceNLP distribution) to become familiar with how to train and run the tagger.

4. Extract the training corpus *penn.train.txt* and the test corpus *penn.test.txt* from `http://http://www.ru.is/~hrafn/courses/nlp/ifdpenn.zip`. These corpora are the same as used for training and testing in Section 2. The test corpus (as well as the training corpus), contains both the tokens and tags. When running *TriTagger* on the test data, you need to supply it with a file containing only the tokens. You can easily generate this test file, *penn.test.tokens.txt*, from *penn.test.txt* by using *awk*.

Now you should be ready to train and test *TriTagger* (make sure that all files used by the tagger are UTF-8 encoded). Make a shell script, **triTagger.sh**, which performs the following:

[3]You need to install the `numpy` module for running this tagger.
[4]`http://aclweb.org/anthology/A/A00/A00-1031.pdf`

1. Builds a training model using *penn.train.txt* for training.
2. Uses *TriTagger* to tag the file *penn.test.tokens.txt* and writes the output to the file *penn.tritagger.out*.
3. Computes the accuracy of *TriTagger* for the given test set. Use Unix/Linux tools to help you derive the accuracy figure (unless you want to do it by hand!). You could do the following:
   (a) Use *sed* to remove empty lines from both the gold standard (*penn.test.txt*) and *penn.tritagger.out*.
   (b) Use *awk* to extract only the first two columns from the output of the tagger.
   (c) Use *wc* to count the number of tokens in the test.
   (d) Use *diff* (using parameters -y and –suppress-common-lines) and *wc* to find the differences between the gold standard and the output of the tagger.
   (e) Use *bc* to perform the final calculation for the accuracy.

   Hint: To assign the output of a command into a variable in a Linux bash shell: variableName=$(command)

   If everything works correctly, the accuracy figure for *TriTagger* on this test set is much higher than the accuracy of the trigram tagger in Section 2. **Explain the reason for this.**

4 CKY Parsing – 25%

Consider the following PCFG grammar:

```
S -> NP VP [1.0]
NP -> DET N [0.8]
NP -> NP PP [0.2]
VP -> V NP [0.4]
VP -> VP PP [0.6]
PP -> P NP [1.0]

DET -> the [0.8]
DET -> a [0.2]
N -> student [0.55}
N -> book [0.25]
N -> library [0.2]
V -> reads [1.0]
P -> in [1.0]
```

1. Parse, by hand, the sentence "the student reads a book in the library", using the given grammar and the CKY parsing algorithm. This means that you need to show the resulting parsing table as in Figure 13.12.

the	student	reads	a	book	in	the	library
DET 0.2							
	N 0.55						
...							

   You will notice that there are two possible parse trees. Calculate the probabilities of each constitutent in the table and thereby show which parse tree will be selected when using a probabilistic CKY parser.

2. Use a ChartParser in the NLTK to parse the sentence above given the grammar. Verify that the parser produces two parse trees.

   Return a Python program, **parsing.py**, which sets up the grammar, parses the sentence, and prints out the two parsing trees.

5 What to return

1. Three Python programs: **ifd.py**, **tagging.py**, and **parsing.py**. Make sure you use functions (where appropriate) for specific tasks in your Python code, thus minimizing the duplication of code.

2. The shell script **triTagger.sh** and the output generated by this script.

3. A file (e.g. .pdf or .txt file), which contains: *i)* answers to the two questions posed in the assignment (one question in Section 2 and another in Section 3), and *ii)* the CKY parsing table, and the two parse trees from Section 4.