CS5012 Language & Computation

Practical 1 Report

University of St Andrews – School of Computer Science

Student ID: 150014151

10th March, 2020

A close up of a logo

Description automatically generated

Table of Contents

[1 Introduction 3](#_Toc34685914)

[1.1 Features implemented 3](#_Toc34685915)

[1.2 Usage 3](#_Toc34685916)

[1.3 Tools 3](#_Toc34685917)

[2 System Design 4](#_Toc34685918)

[2.1 File organisation 4](#_Toc34685919)

[2.2 Data pre-processing 4](#_Toc34685920)

[2.3 Training 4](#_Toc34685921)

[2.4 Testing 5](#_Toc34685922)

[3 Evaluation 6](#_Toc34685923)

[3.1 Baseline evaluation 6](#_Toc34685924)

[3.2 Accuracy evaluation 6](#_Toc34685925)

[3.3 Other languages 6](#_Toc34685926)

[References 7](#_Toc34685927)

[Appendix A 8](#_Toc34685928)

1. Introduction
   1. Features implemented

* Frequency Distribution smoothing
* Unknown words handling
* Other languages
  1. Usage

To run the POS tagger in Python, move to the “*src*” directory and run the following command.

1. python main.py [-r] [-d]

where:

* *-r:* is a flag that forces the program to recompute the HMM’s tag transition and word emission probabilities rather than loading previously computed versions into memory.
* *-d* is a flag that enters *debugging* mode, printing additional statements on the command line.
  1. Tools
* Programming language: Python 3.7
* Python libraries used: NLTK[[1]](#footnote-1)
* Editor: PyCharm[[2]](#footnote-2)
* Version controlling: Git and GitHub (private repository)

1. System Design
   1. File organisation

The project in organised into the following python modules:

* “*main.py*”: contains functions critical to the training and testing of the POS tagger (e.g. calculating the HMM’s tag transition/word emission probabilities, computing the Viterbi algorithm, etc.).
* “*helpers.py*”: contains support functions such as operations on data (e.g. extracting words or tags from the data sets, getting the hapax legomenon from a list, etc.) and printing statements.
* “*config.py*”: contains project-wide variables that are set by the command line arguments.
  1. Data pre-processing

The first step consists in splitting command line arguments in order to run different sections of the program. This is achieved through Python’s “*argparse*” library[[3]](#footnote-3).

Next, sentences from the Brown corpus [1] are split into a training and a testing data set. Start-of-sentence tokens <s> (with a <s> POS) are inserted at the beginning of each sentence in both data sets, as well as end-of-sentences tokens </s> (with a </s> POS), which are appended at the end of each sentence. The default splits (used for baseline evaluation) are 10,000 sentences in the training set and 500 sentences in the testing set.

1. START\_TAG\_TUPLE = ("<s>", "<s>")
2. END\_TAG\_TUPLE = ("</s>", "</s>")
3. for sentence in data:
4. sentence.insert(0, START\_TAG\_TUPLE)
5. sentence.insert(len(sentence), END\_TAG\_TUPLE)
   1. Training

Using only the training set, the HMM’s (bigram) tag transition and word emission probabilities are calculated and smoothed using NLTK’s *WittenBellProbDist[[4]](#footnote-4)* function.

These large probability matrices (which are stored in python dictionaries) are each saved in a Pickle[[5]](#footnote-5) “*.pkl*” file, allowing the whole program to execute faster in future runs by loading the matrices back into memory. A command-line flag can be set to force both matrices to be re-calculated.

* 1. Testing

The POS tagger is tested on the aforementioned testing data set by comparing the predicted tags with the actual tags. The predicted tags are determined by using the Viterbi algorithm (which makes use of the previously generated HMM’s tag transition and word emission probabilities) before back-tracing the generated Viterbi matrix to reconstruct the most likely sequence of POS tags for each sentence in the testing data set.

The POS tagger is tested by iterating through the testing data set’s sentences rather than words to avoid underflow, as Python cannot store floats that are smaller than 1e-308 [2].

1. def test\_tagger(testing\_set: list, unique\_training\_tags: list, tag\_transition\_probabilities: dict, emission\_probabilities: dict, test\_sentences: list) -> None:
2. predicted\_tags\_per\_sentence = list()
3. for sentence in test\_sentences:
4. testing\_words = [w for (w, \_) in sentence]
5. viterbi\_matrix = viterbi\_algorithm\_smoothed(testing\_words, unique\_training\_tags, tag\_transition\_probabilities, emission\_probabilities)
6. predicted\_tags\_per\_sentence.append(backtrace(viterbi\_matrix))
7. predicted\_tags = list(chain.from\_iterable(predicted\_tags\_per\_sentence))
8. tagging\_accuracy = round(calculate\_accuracy(predicted\_tags, testing\_set), 2)
9. print("POS Tagging accuracy on test dataset: {}%".format(tagging\_accuracy))

Finally, the tagger’s accuracy is calculated by counting the number of correct predictions for each predicted tag with the actual tags from the Brown corpus.

1. for index, (actual\_tag, predicted\_tag) in enumerate(zip(actual\_tags, predicted\_tags)):
2. if actual\_tag == predicted\_tag:
3. correct\_tag\_counter += 1
4. accuracy = (correct\_tag\_counter / len(actual)) \* 100

1. Evaluation
   1. Baseline evaluation

The baseline evaluation consists in the recommended data split of 10,000/500 sentences for the training/testing data sets.

The evolution of the accuracy was measured by reverting back to previous commits and measuring the accuracy at each milestone, which consist of the following:

* No smoothing, naïve infrequent world handling (assigning 1/1000 probability to each unknown word) [89.94%]
* Smoothing applied to word emission probabilities
* Infrequent word handling
  1. Accuracy evaluation

This evaluation consists in tweaking the following parameters to determine which allows for the highest accuracy:

* Witten-Bell distribution probability estimates bin sizes
* Different number of rules applied to unknown words
* Threshold of infrequent words in the training set
* Train/test data split
  1. Other languages

References

[1] W. N. Francis and H. Kucera, “Brown Corpus Manual,” *Department of Linguistics, Brown University*, 1979. [Online]. Available: http://korpus.uib.no/icame/brown/bcm.html. [Accessed: 09-Mar-2020].

[2] “sys — System-specific parameters and functions — Python 3.7.7rc1 documentation,” *Python Software Foundation*, 2020. [Online]. Available: https://docs.python.org/3.7/library/sys.html#sys.float\_info. [Accessed: 09-Mar-2020].

Appendix A

<https://www.nltk.org/book/ch05.html>

A screenshot of text

Description automatically generated

1. NLTK: <https://www.nltk.org/api/nltk.html?highlight=witten#nltk.probability.WittenBellProbDist> [↑](#footnote-ref-1)
2. PyCharm: <https://www.jetbrains.com/pycharm/> [↑](#footnote-ref-2)
3. Python argparse library: <https://docs.python.org/3.7/library/argparse.html> [↑](#footnote-ref-3)
4. WittenBellProbDist: <https://www.nltk.org/api/nltk.html?highlight=witten#nltk.probability.WittenBellProbDist> [↑](#footnote-ref-4)
5. Pickle: <https://docs.python.org/3.7/library/pickle.html> [↑](#footnote-ref-5)