CPSC 477/577: Natural Language Processing
Spring 2021

Assignment 1 (8 points)
Language Modeling and Part of Speech Tagging

Due Thursday, March 4, 11:59:59 pm

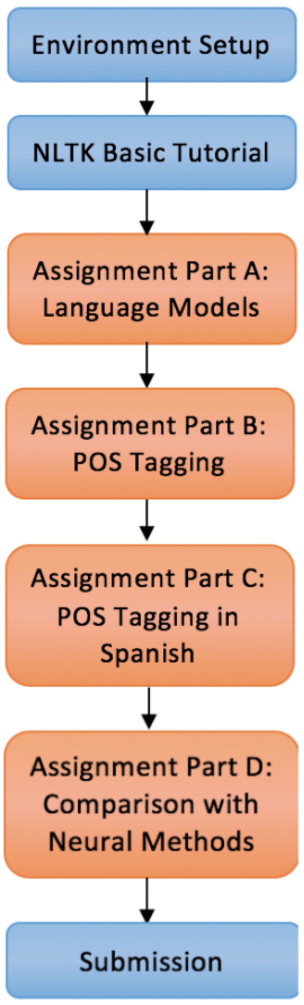
TF: Aditya Chander

Introduction

In this assignment we will:

1. Go through the basics of NLTK, the most popular NLP library for Python;
2. Develop and evaluate several language models;
3. Develop and evaluate a full part-of-speech tagger using the Viterbi algorithm;
4. Compare that part-of-speech tagger with a neural network-based model.

This document is structured in the following sequence:


Environment Setup
NLTK Basic Tutorial
Assignment Part A: Language Models
Assignment Part B: POS Tagging
Assignment Part C: POS Tagging in Spanish
Assignment Part D: Comparison with Neural Methods
Submission

Environment Setup

Once you have SSHed into the Zoo cluster, copy the homework files to your hidden directory under Homework1 folder.

```
cp -r /home/classes/cs477/assignments/2021-Homework1 ~/hidden/<YOUR_PIN>
```

For the virtualenv, all you need to do is include the following line in your environmental setup section:
Source the following environment to ensure that you are using the same packages that will be used for grading:

```
source /home/classes/cs477/venvs/hw1/bin/activate
```

If you wish to reproduce the virtualenv on your own computer, do the following:
While in the sourced virtualenv:

```
pip freeze > requirements.txt
```

Copy the requirements file to your computer

Then, create a virtualenv on your computer

```
virtualenv --python=python3 hw1
source hw1/bin/activate
pip install -r requirements.txt
```

Basic NLTK Tutorial

The Natural Language Toolkit (NLTK) is the most popular NLP library for python. NLTK has already been installed on the Zoo, so you do not have to install it yourself. If you followed the "Environment Setup" section, you should be good to start using NLTK now.

Now let's walk through some simple NLTK use cases that concern this assignment. For that, you will need to first open a python interactive shell. Just type `python` or `python3` on the command line and you should see the interactive shell start.

Import the NLTK package
To use the NLTK package, include the following line at the beginning of your code (or in this case just type directly into the interactive shell):
```
import nltk
```

Tokenization
To tokenize means to break a continuous string into tokens (usually words, but a token could also be a symbol, punctuation, or other meaningful unit). In NLTK, text can be tokenized using the `word_tokenize()` method. It returns a list of tokens that will be the input for many methods in NLTK.
```
sentence = "At eight o'clock on Thursday morning on Thursday morning on Thursday morning."
tokens = nltk.word_tokenize(sentence)
```

N-grams Generation
An n-gram (in the context of this assignment) is a contiguous sequence of n tokens in a sentence. The following code returns a list of bigrams and a list of trigrams. Each n-gram is represented as a tuple in python (if you are not familiar with python tuples read the python tuple doc page)
```
bigram_tuples = list(nltk.bigrams(tokens))
trigram_tuples = list(nltk.trigrams(tokens))
```

We can calculate the count of each n-gram using the following code:
```
count = {item: bigram_tuples.count(item) for item in set(bigram_tuples)}
```

Or we can find all the distinct n-grams that contain the word "on":
```
ngrams = [item for item in set(bigram_tuples) if "on" in item]
```

If you find it hard to understand the examples above, read about list/dict comprehensions here. List/dict comprehensions are a way of executing iterations in one line that may be very useful and convenient. Besides making coding easier, learning them will also help you understand code written by other python programmers.

Note: *if you want to calculate the counts in a faster way, you can use a Counter object from python's built-in collections library via the following code:*
```
from collections import Counter
count = Counter(bigram_tuples)
```
The dictionary comprehension code can take in the worst-case O(n^2) time, whereas the Counter object runs in O(n). If you're curious, think about why and how this is possible. For more, please check: https://docs.python.org/3.6/library/collections.html#collections.Counter.
[You can always use other methods/objects to implement instead of Counter]

Default POS Tagger (Non-statistical)
The most naïve way of tagging parts-of-speech is to assign the same tag to all the tokens. This is exactly what the NLTK default tagger does. Although inaccurate and arbitrary, it sets a baseline for taggers, and can be used

as a default tagger when more sophisticated methods fail.

In NLTK, it's easy to create a default tagger by indicating the default tag in the constructor.

```
default_tagger = nltk.DefaultTagger('NN')
tagged_sentence = default_tagger.tag(tokens)
```

Now we have our first tagger. NLTK can help if you need to understand the meaning of a tag.

```
# Show the description of the tag 'NN'
nltk.help.upenn_tagset('NN')
```

Regular Expression POS Tagger (Non-statistical)

A regular expression tagger maintains a list of regular expressions paired with a tag (see the Wikipedia article for more information about regular expressions: http://en.wikipedia.org/wiki/Regular_expression). The tagger tries to match each token to one of the regular expressions in its list such that the token receives the tag that is paired with the first matching regular expression. "None" is given to a token that does not match any regular expression.

To create a Regular Expression Tagger in NLTK, we provide a list of pattern-tag pairs to the appropriate constructor. Example:

```
patterns = [(r'.*ing$', 'VBG'),(r'.*ed$', 'VBD'),(r'.*es$', 'VBZ'),(r'.*ed$', 'VB')]
regexp_tagger = nltk.RegexpTagger(patterns)
regexp_tagger.tag(tokens)
```

See how many affix patterns you can come up with for your regexp_tagger up to a maximum of 10 rules. In what situations might this regexp_tagger report the wrong tags? Include your rules and some example situations this in your README.txt file (see below).

N-gram HMM Tagger (Statistical)

Although there are many different kinds of statistical taggers, we will only work with Hidden Markov Model (HMM) taggers in this assignment.

Like every statistical tagger, n-gram taggers use a set of tagged sentences, known as the training data, to create a model that is used to tag new sentences. In NLTK, a sentence of the training data must be formatted as a list of tuples, where each tuple is a pair in word-tag format (see example below).

```
[('The', 'AT'), ('Fulton', 'NP-TL'), ('County', 'NN-TL')]
```

NLTK already provides corpora formatted this way. In particular, we are going to use the Brown corpus.

```
# import the corpus from NLTK and build the training set from sentences in "news"
from nltk.corpus import brown
training = brown.tagged_sents(categories='news')

# Create Unigram, Bigram, Trigram taggers based on the training set.
unigram_tagger = nltk.UnigramTagger(training)
bigram_tagger = nltk.BigramTagger(training)
trigram_tagger = nltk.TrigramTagger(training)
```

Although we could also build 4-gram, 5-gram, etc. taggers, trigram taggers are the most popular model. This is because a trigram model is an excellent compromise between computational complexity and performance.

Combination of Taggers

A tagger fails when it cannot find a best tag sequence for a given sentence. For example, one situation when an n-gram tagger will fail is when it encounters an OOV (out of vocabulary) word not seen in the training data; the tagger will tag the word as "NONE". One way to handle tagger failure is to fall back to an alternative tagger if the primary one fails. This is called "using back off." One can easily set a hierarchy of taggers in NLTK as

follows:
```
default_tagger = nltk.DefaultTagger('NN')
bigram_tagger = nltk.BigramTagger(training, backoff=default_tagger)
trigram_tagger = nltk.TrigramTagger(training, backoff=bigram_tagger)
```

Tagging Low Frequency Words
Low frequency words are another common source of tagger failure, because an n-gram that contains a low frequency word and is found in the test data might not be found in the training data. One method to resolve this tagger failure is to group low frequency words. For example, we could substitute the token “_RARE_” for all words with frequency lower than 0.05% in the training data. Any words in the development data that were not found in the training data could then be treated instead as the token “_RARE_”, thereby allowing the algorithm to assign a tag. If we wanted to add another group, we could substitute the string “_NUMBER_” for those rare words that represent a numeral. When tagging the test data, we could substitute “_NUMBER_” for all tokens that were unseen in the training data and represent a numeral. We will use this technique later in this assignment.

Provided Files and Report

Now let’s go back to the assignment folder you copied to your home directory (should be ~/hidden/<Your PIN>/Homework1). In this assignment we will be using the Brown Corpus, which is a dataset of English sentences compiled in the 1960s. We have provided this dataset to you so you don’t have to load it yourself from NLTK.

Besides data, we are also providing code for evaluating your language models and POS tagger, and a skeleton code for the assignment. In this assignment you should not create any new code files, but rather just fill in the functions in the skeleton code.

We have provided the following files:

data/Brown_train.txt	Untagged Brown training data
data/Brown_tagged_train.txt	Tagged Brown training data
data/Brown_dev.txt	Untagged Brown development data
data/Brown_tagged_dev.txt	Tagged Brown development data
data/Sample1.txt	Additional sentences for part A
data/Sample2.txt	More additional sentences for part A
data/wikicorpus_tagged_train.txt	Tagged Wikicorpus training data
data/wikicorpus_dev.txt	Untagged Wikicorpus development data
data/wikicorpus_tagged_dev.txt	Tagged Wikicorpus development data
perplexity.py	A script to analyze perplexity for part A
pos.py	A script to analyze POS tagging accuracy for part B
neural_tagger_model.py	A neural POS tagger model
tagger.pt.model	Model with trained parameters for the neural tagger
solutionsA.py	Skeleton code for part A
solutionsB.py	Skeleton code for part B
solutionsC.py	A script to train and evaluate a Spanish tagger for part C
solutionsD.py	A script to evaluate the neural tagger from for part D
output/	Directory where answers to part A, B, and C are stored

The only files that you should modify throughout the whole assignment are solutionsA.py and solutionsB.py.

Data Files Format

The untagged data files have one sentence per line, and the tokens are separated by spaces. The tagged data files are in the same format, except that, instead of tokens separated by spaces, those files have TOKEN/TAG pairs separated by spaces.

Report

Before starting the assignment, create a “README.txt” file on the homework folder. At the top, include a header that contains your netID and name. Throughout the assignment, you will be asked to include specific output or comment on specific aspects of your work. We recommend filling the README file as you go through the assignment, as opposed to starting the report afterwards.

In this report it is not necessary to include introductions and/or explanations, other than the ones explicitly requested throughout the assignment.

The first thing to report in the README is the affix patterns you came up with for your regexp_tagger and some situations (with brief explanation) where the tagger didn’t work.

Assignment Part A – Language Model

In this part of the assignment, you will be filling the **solutionsA.py** file. Open the file and notice there are several functions with a #TODO comment; you will have to complete those functions. To understand the general workflow of the script, read the main() function ***but do not modify it***. You also shouldn't import any additional libraries/functions beyond what is already provided, but you also don't have to use everything that's provided if you can find a solution that works without something.

If you are having issues with saving your edits in the file, check the permissions using `ls -la` in the terminal. If the third character is `-`, you won't be able to edit: it should be `w`. To fix this, you can write the command
`chmod u+w solutionsA.py`

1) Calculate the uni-, bi-, and trigram log-probabilities of the data in "data/Brown_train.txt". This corresponds to implementing the calc_probabilities() function. In this assignment we will always use **log base 2**. (Remember that n-gram probabilities are calculated as the *probability of the nth tag conditioned on the first n-1 tags*. It's not the joint probability of the entire n-gram. This confused me at first when I took the class!)

   Don't forget to add the appropriate sentence start and end symbols; use the provided constants START_SYMBOL and STOP_SYMBOL in the skeleton code. You may or may not use NLTK to help you here.

   The code will automatically output the log probabilities in a file "output/A1.txt". Here's a few examples of log probabilities of uni-, bi-, and trigrams for you to check your results:

   UNIGRAM captain -14.2810
   UNIGRAM captain's -17.0883
   UNIGRAM captaincy -19.4102

   BIGRAM and religion -12.9316
   BIGRAM and religious -11.3466
   BIGRAM and religiously -13.9316

   TRIGRAM and not a -4.0297
   TRIGRAM and not by -4.6147
   TRIGRAM and not come -5.6147

   Make sure your results match the examples above up to four decimal places. Note that even though the code produces many decimal digits, common sense tells us to consider only the most significant ones. If they do match, include in your README the log probabilities to four decimal places of the following n-grams (**note the n-grams are case-sensitive**):

   UNIGRAM natural
   BIGRAM natural that
   TRIGRAM natural that he

2) Use your models to find the log-probability, or score, of each sentence in the Brown training data with each n-gram model. This corresponds to implementing the score() function.

Make sure to accommodate the possibility that you may encounter in the sentences an n-gram that doesn't exist in the training corpus. This will not happen now, because we are computing the log-probabilities of the training sentences, but it will be necessary for question 5. If you find any n-gram that was not in the training sentences, set the whole sentence log-probability to the constant MINUS_INFINITY_SENTENCE_LOG_PROB.

The code will output scores in three files: "output/A2.uni.txt", "output/A2.bi.txt", "output/A2.tri.txt". These files simply list the log-probabilities of each sentence for each different model. Here's what the first few lines of each file look like (truncated at four decimal places):

A2.uni.txt
-178.7268
-259.8586
-143.3304

A2.bi.txt
-92.1039
-132.0966
-90.1859

A2.tri.txt
-26.1800
-59.8531
-42.8392

Now, you need to run our perplexity script, "perplexity.py", on each of these files. This script will count the words of the corpus and use the log-probabilities computed by you to calculate the total perplexity of the corpus. To run the script, the command is:

```
python3 perplexity.py <file of scores> <file of sentences that were scored>
```

where <file of scores> is one of the A2 output files and <file of sentences that were scored> is "data/Brown_train.txt". **Include the perplexity of the corpus for the three different models in your README**. Here's what our script printed when <file> was "A2.uni.txt" (truncated at four decimal places).

```
python3 perplexity.py output/A2.uni.txt data/Brown_train.txt
```
The perplexity is 1052.4865

3) Implement linear interpolation among the three n-gram models you have created. This corresponds to implementing the linearscore() function.

Linear interpolation is a method that aims to derive a better tagger by using all three uni-, bi-, and trigram taggers at once. Each tagger is given a weight described by a parameter lambda. There are some excellent methods for approximating the best set of lambdas, but for now, set the value of all three lambdas to be equal (i.e., 1/3). You can read more about linear interpolation on page 44 of the Jurafsky & Martin textbook.

The code outputs scores to "output/A3.txt". The first few lines of this file look like (truncated at four decimal places):

-46.5892
-85.7742
-58.5442
-47.5165
-52.7387

Run the perplexity script on the output file and include the perplexity in your README.

4) In the `linearscore_newlambdas()` function, duplicate your `linearscore()` function but experiment with different values of lambda to see if you can improve the perplexity score. You don't have to use the methods from Jurafsky & Martin but if you want to try implementing them, go for it! (I'd recommend just toying around with them manually, though, as you will need to spend a decent amount of time on part B.) In the README file, include the values of your new lambdas and explain why you think they might have improved the performance.

5) Briefly answer in your README the following questions: When you compare the performance (perplexity) between the best model without interpolation and the best model with linear interpolation, is the result you got expected? Explain why. (max 60 words, but 30 is fine too!)

6) Both "data/Sample1.txt" and "data/Sample2.txt" contain sets of sentences; one of the files is an excerpt of the Brown training dataset. Use your model to score the sentences in both files. Our code outputs the scores of each into "Sample1_scored.txt" and "Sample2_scored.txt". Run the perplexity script on both output files and include the perplexity output of both samples in your README. Use these results to make an argument for which sample belongs to the Brown dataset and which does not.

Assignment Part B – Part-of-Speech Tagging for English

In this part of the assignment, you will be filling the solutionsB.py file. Open the file and notice there are several functions with a #TODO comment; you will have to complete those functions. To understand the general workflow of the script, read the main() function, ***but do not modify it***. If you have issues with the writing permissions, do the same thing as you did for part A.

1) First, you must separate the tags and words in "Brown_tagged_train.txt". This corresponds to implementing the split_wordtags() function. You'll want to store the sentences without tags in one data structure, and the tags alone in another (see instructions in the code). Make sure to add sentence start and stop symbols to **both** lists (of words and tags). Use the constants START_SYMBOL and STOP_SYMBOL already provided. You don't need to write anything on README about this question.

Hint: make sure you accommodate words that themselves contain backslashes – i.e., "1/2" is encoded as "1/2/NUM" in tagged form; make sure that the token you extract is "1/2" and not "1".

2) Now, calculate the trigram probabilities for the tags. This corresponds to implementing the calc_trigrams() function. The code outputs your results to a file "output/B2.txt". Here are a few lines (not contiguous and truncated at four decimal places) of this file for you to check your work:

TRIGRAM * * ADJ -5.2055
TRIGRAM ADJ . X -9.9961
TRIGRAM NOUN DET NOUN -1.2645
TRIGRAM X . STOP -1.9292

After you checked your algorithm is giving the correct output, add to your README the log probabilities of the following trigrams:

TRIGRAM CONJ ADV ADP
TRIGRAM DET NOUN NUM
TRIGRAM NOUN PRT PRON

Note: you might wish to reuse a function you wrote in part A to make your life easier.

3) The next step is to implement a smoothing method. To prepare to add smoothing, replace every word that occurs five or fewer times with the token specified in the constant RARE_SYMBOL. This corresponds to implementing the calc_known() and replace_rare() functions.

First, you will create a list of words that occur *more* than five times in the training data. When tagging, any word that does not appear in this list should be replaced with the token in RARE_SYMBOL. You don't need to write anything on README about this question. The code outputs the new version of the training data to "output/B3.txt". Here are the first two lines of this file:

At that time highway engineers traveled rough and dirty roads to accomplish their duties .
RARE _RARE_ vehicles was a personal _RARE_ for such employees , and the matter of providing state transportation was felt perfectly _RARE_ .

Hint: if you use a set instead of a list to store frequently occurring words, this operation will be faster. A set *is python's implementation of a hash table and has constant time membership checking, as opposed to a list, which has linear time checking.*

4) Next, we will calculate the emission probabilities on the modified dataset. This corresponds to implementing the calc_emission() function. Here are a few lines (not contiguous) from this file for you to check your work:

America NOUN -10.9992
Columbia NOUN -13.5599
New ADJ -8.1884
York NOUN -10.7119

After you check that your algorithm is giving the correct output, add to your README the log probabilities of the following emissions **(note words are case-sensitive)**:

*** ***
Night NOUN
Place VERB
prime ADJ
STOP STOP
RARE VERB

5) Now, implement the Viterbi algorithm for HMM taggers. The Viterbi algorithm is a dynamic programming algorithm that has many applications. For our purposes, the Viterbi algorithm is a comparatively efficient method for finding the highest scoring tag sequence for a given sentence. Please read about the specifics about this algorithm in section 8.4 of the book.

Note: your book uses the term "state observation likelihood" for "emission probability" and the term "transition probability" for "trigram probability."

Using your emission and trigram probabilities, calculate the most likely tag sequence for each sentence in "Brown_dev.txt". This corresponds to implementing the viterbi() function. Your tagged sentences will be outputted to "B5.txt". The first two tagged sentences should look like this:

He/PRON had/VERB obtained/VERB and/CONJ provisioned/VERB a/DET veteran/ADJ ship/NOUN called/VERB the/DET Discovery/NOUN and/CONJ had/VERB recruited/VERB a/DET crew/NOUN of/ADP twenty-one/NOUN ,/. the/DET largest/ADJ he/PRON had/VERB ever/ADV commanded/VERB ./.
The/DET purpose/NOUN of/ADP this/DET fourth/ADJ voyage/NOUN was/VERB clear/ADJ ./.

Note that, while the output doesn't have the "_RARE_" token, you still have to count unknown words as a "_RARE_" symbol to compute probabilities inside the Viterbi Algorithm.

When exploring the space of possibilities for the tags of a given word, make sure to only consider tags with emission probability greater than zero for that given word. Also, when accessing the transition probabilities of tag trigrams, use -1000 (constant LOG_PROB_OF_ZERO in the code) to represent the log-probability of an unseen transition.

Once you run your implementation, use the part of speech evaluation script pos.py to compare the output file with “Brown_tagged_dev.txt”. Include the accuracy of your tagger in the README file. To use the script, run the following command:

```
python3 pos.py output/B5.txt data/Brown_tagged_dev.txt
```

This is the result we got with our implementation of the Viterbi algorithm:

Percent correct tags: 93.3250

Do what you can to make your algorithm as efficient as possible! While we won’t give you specifics, you should think about the order in which you check the words of the trigrams, which words and/or tags you can ignore, etc. For reference, our solution runs in 11-12 seconds. This algorithm is tricky enough to implement, let alone optimize, so don’t be disheartened if you can’t get it to run at lightning speed! When I did this assignment for the class, I could only get to 35-ish seconds, and I spent the bulk of the time optimizing my solution. While we will give points for efficiency, it’ll only be a small portion of your assignment grade.

6) Finally, create an instance of NLTK’s trigram tagger set to back off to NLTK’s bigram tagger. Let the bigram tagger itself back off to NLTK’s default tagger using the tag “NOUN”. Implement this in the nltk_tagger() function. The code outputs your results to a file “B6.txt”, and this is how the first two lines of this file should look:

He/NOUN had/VERB obtained/VERB and/CONJ provisioned/NOUN a/DET veteran/NOUN ship/NOUN called/VERB the/DET Discovery/NOUN and/CONJ had/VERB recruited/NOUN a/DET crew/NOUN of/ADP twenty-one/NUM ,/. the/DET largest/ADJ he/PRON had/VERB ever/ADV commanded/VERB ./.
The/NOUN purpose/NOUN of/ADP this/DET fourth/ADJ voyage/NOUN was/VERB clear/ADJ ./.

Use pos.py to evaluate the NLTK’s tagger accuracy and put the result in your README. This is the accuracy that we got with our implementation:

Percent correct tags: 88.0399

Assignment Part C – Part-of-Speech Tagging for Spanish

In this part of the assignment, you will be using solutionsC.py. To understand the general workflow of the script, read the main() function, ***but do not modify it***. Note that solutionsC.py uses functions from solutionsB.py. It is recommended that you complete Part B before you begin Part C.

Part C uses a corpus other than the Brown corpus. The training data comes from Wikicorpus, a trilingual corpus using passages from Wikipedia. We will be using the Spanish portion to train and evaluate a part-of-speech tagger.

As you may suspect, different languages may use different tagsets. The data from Wikicorpus uses the Parole tagset, which is similar to the Brown corpus's tagset, but also has tags that are not used in English (such as DE for determiner, exclamatory; Fia for punctuation, ¿; VS* for semi-auxiliary verbs). For instance, the Brown corpus distinguishes between adverbs, comparative adverbs, and superlative adverbs (I run fast, but Miles runs faster, and Drago runs fastest.) The Parole tagset does not differentiate between such adverbs, but instead distinguishes between regular adverbs and negative adverbs (nunca - never). This particular tag may be helpful in Spanish since some sentences use double negatives for emphasis ("No conozco a nadie" word-for-word translates to "I do not know nobody" and actually means "I don't know anybody".)

1) You can use your solutionsC.py to train and evaluate a part-of-speech tagger in Spanish, with normal trigrams. To do this, run the following:

```
python3 solutionsC.py
```

This program will likely take a lot longer to run than solutionsB.py.

Run the following command to find the accuracy:

```
python3 pos.py output/C5.txt data/wikicorpus_tagged_dev.txt
```

In your README, record the accuracy of your tagger for Spanish. Our implementation of the Spanish tagger achieved the following performance (to three decimal places):

Percent correct tags: 84.472

In your README, answer the following questions:

- The Spanish dataset takes longer to evaluate. Why do you think this is the case?
- What are aspects or features of a language that may improve tagging accuracy that are not captured by the tagged training sets?

Assignment Part D – Comparison with Neural Methods

In this part of the assignment, you will compare your HMM based tagger with an existing neural network-based model for POS tagging, by Jonathan K. Kummerfeld from the University of Michigan. A version of the code that uses a library called PyTorch is available in neural_tagger_model.py. (You do not have to implement or modify it! Soon enough, you will be learning more about neural models). Their implementation uses the GloVe word embeddings, along with a neural network called a bidirectional LSTM (a method of sequence learning). For this assignment, we have already trained the model on the Brown corpus in English. You simply have to evaluate this neural model.

You will evaluate tagger.pt.model, the trained model by running:

```
python3 solutionsD.py
```

You should get something like this:
Test accuracy: 96.691

In your README, answer the following question:

- How does this result compare with your HMM based tagger?

Submission

Before submission, run your code one more time and record in your README file the time it took to execute Part A, Part B, and Part C, *separately*. Add the times of execution in the end of your README file.

You should submit your assignment through your hidden directory on the Zoo server. Once you are done with the homework and have finalized the README.txt file, check once again that this is the path for your homework:

```
~/hidden/<YOUR_PIN>/Homework1/
```

Please make sure that these files are in your folder with the directory specified exactly as shown below. If you don't, you may lose points for otherwise correct submissions! There will be other stuff in there too, but these are the crucial files:

```
~/hidden/<YOUR_PIN>/Homework1/solutionsA.py
```

```
~/hidden/<YOUR_PIN>/Homework1/solutionsB.py
```

```
~/hidden/<YOUR_PIN>/Homework1/README.txt
```

As a final step, run a script that will set up permissions for your homework files, so we can access and run your code to grade it:

```
/home/classes/cs477/bash_files/hw1_set_permissions.sh <YOUR_PIN>
```

Make sure the command above runs without errors, and **do not make any changes or run the code again**. If you do run the code again or make any changes, you need to run the permissions script again. Submissions without the correct permissions may incur some grading penalty.