CIS 530 Fall 2015 Assignment 3

Instructor: Ani Nenkova
Head TA: Anne Cocos

Released: October 14, 2015
Due: 11:59pm, October 31, 2015

Overview

In this assignment we will explore various models for part of speech tagging. You will experiment with several feature sets and methods for handling low-frequency words.

There are in total 100 points for the 6 problems in this homework.

NOTE: If you choose to develop on Penn's servers, please use the Biglab machines. The Biglab servers can be accessed directly using the command `>> ssh your_penn_key@biglab.seas.upenn.edu` or from Eniac using the command `>> ssh biglab`.

Submitting your work

You must work **independently** on this assignment and make your own submission.

Your submission consists of 6 files, including your code and 5 output files:

- Your code, which you should place in a **single file** named `hw3_code_yourpennkey.py`.
- Output from Section 3.1, `hw_3_3_1.txt`.
- Output from Section 4, `hw_3_4.txt`.
- Output from Section 5, `hw_3_5.txt`.
- Output from Section 6, `hw_3_6.txt`.
- The accuracy of your four models, written to `modelscores.txt`

To submit your code and output files, place them together in their own directory (for example `cis530hw3`), compress them into a .zip file called `hw2_yourpennkey.zip`, then submit via `turnin` using the following commands from within `cis530hw3`:

```
yourpennkey:~/cis530hw3> ls
hw2_code_yourpennkey.py
hw_3_3_1.txt
hw_3_4.txt
hw_3_5.txt
hw_3_6.txt
modelscores.txt
```

```
yourpennkey:~/cis530hw3> zip hw2_yourpennkey *
yourpennkey:~/cis530hw3> ls
hw2_code_yourpennkey.py
hw_3_3_1.txt
hw_3_4.txt
hw_3_5.txt
hw_3_6.txt
modelscores.txt
hw2_yourpennkey.zip
yourpennkey:~/cis530hw3> turnin -c cis530 -p hw2 hw2_yourpennkey.zip
```

If you choose to work locally, you can upload your code to Eniac for submission using an SFTP client such as FileZilla, WinSCP or Cyberduck, or secure copy:

```
> scp local_file yourpennkey@eniac.seas.upenn.edu:PATH
```

where `local_file` is the path to your homework on the local machine and PATH is the path to the location on Eniac where you wish to copy it.

You will get a confirmation message upon submission. You can run `turnin` multiple times before the deadline; old submissions are overwritten each time. To check that the homework was submitted successfully, you can run `> turnin -c cis530 -v` to see the list of files you have submitted.

Code Guidelines

- Your code will be tested on `biglab` using Python 2.7. Ensure that your code can be imported on `biglab` and returns the correct results when run using Python 2.7.
- In each graded section we will specify one or more required functions that will be run for grading. Along the way we will also provide tips on helper functions you may want to use. Whether you follow this implementation exactly or devise your own is up to you.
- Your code will be graded automatically. It is your responsibility to make sure your required functions can be called with the specified input and produce the specified output. To ensure your code will run properly during grading, please include **only** function declarations above the line `if __name__ == ¨__main__¨` and prefix **all** global variables with `yourpennkey`.
- You can use any NLTK modules you find useful, unless you are specifically asked not to do so.
- Do not do any preprocessing (i.e. tokenizing, lowercasing) of the input text unless the assignment specifically says to do so. Please follow the directions of the assignment for the code you submit and feel free to experiment with alternatives on your own.

Data

All of the data you will need for this homework is in: `'/home1/c/cis530/hw3/data'`.

- **Part-of-Speech tagged data:** In this assignment we will use a portion of the Wall Street Journal (WSJ) section of the Penn Treebank (PTB) POS-tagged corpus. The data is tagged using the PTB tagset.

  In the `cis530/hw3/data/train` and `cis530/hw3/data/test` directories you will find files containing POS-tagged data. You can assume that sentences are separated by one or more blank lines or `======` lines , and `token/POS` pairs are separated by spaces or newline characters. Individual sentences may span multiple lines. An example showing 2 sentences is below.

```
./.But/CC
[ analysts/NNS ]
,/, while/IN applauding/VBG
[ the/DT acquisition/NN ]
,/, say/VBP Applied/NNP
[ 's/POS chief/JJ executive/NN ]
faces/VBZ
[ a/DT tough/JJ challenge/NN ]
in/IN integrating/VBG
[ the/DT two/CD companies/NNS ]
./.

======================================

[ Barry/NNP Wright/NNP ]
,/, acquired/VBN by/IN Applied/NNP for/IN
[ $/$ 147/CD million/CD ]
,/, makes/VBZ
[ computer-room/JJ equipment/NN ]
and/CC
[ vibration-control/JJ systems/NNS ]
./.
```

- **Mapping from PTB tagset to Universal tagset** Instead of using the 45 tags in the PTB tagset, we will train our model and make predictions using the 12 tags in the Google Universal tagset instead. We have provided a mapping from the PTB tagset to the Google tagset in `cis530/hw3/data/en-ptb.map`. The format of each line in this file is `<PTBTag> <GoogleTag>`, separated by a tab.
- **word2vec Word Embeddings**: In Section 4 we will use lexical substitution to replace low-frequency words with similar, more frequent, words. To do this we will use `word2vec` word embeddings. There are pre-computed 300-dimensional `word2vec` vectors for the vocabulary in our training and testing data available in the file `cis530/hw3/data/w2vec_hw3`. More details on how these vectors were trained is available at this link: `https://code.google.com/p/word2vec/`.

1 Preparing our Data (10 Points)

Each raw PTB-tagged file contains multiple sentenced, tagged with the wrong tagset for our use. So first we will need to do some preprocessing of our data.

Required: Define the following method:

- `parse_taggedfile(wsjfile, tagmap)`: Parse the raw PTB-tagged `wsjfile`, which has token/pos pairs separated by spaces or newlines, and sentences separated by one or more blank lines. Ignore any brackets or `=====` lines in the raw file. **You should convert all tokens (words) to lowercase.**

  Param: `wsjfile` is a `str` that gives the relative path to a raw POS-tagged file, and `tagmap` is a `dict` that provides a mapping from PTB tags to Google Unviersal tags.

  Return: A `list` of `lists` of `(token, pos)` tuples, where each inner list comprises (lowercase) token/pos pairs from a single sentence. The output tuples should contain Google Universal part of speech tags, as mapped from the PTB tags in the input file using `tagmap`.

Grading: We will check the output of your `parse_taggedfile` function against test input. We will verify that the tags are properly mapped and that the sentences are properly split.

2 Baseline Model: Most Frequent Tag (15 Points)

Often before building a prediction model it is helpful to establish a baseline by seeing how well a very simple model would perform on the dataset. For this assignment our baseline will be a prediction model that simply tags every token in the test set with its most frequent POS tag from the training set.

Required: Define the following methods:

1. `create_mft_dict(filelist)`: Create a dictionary of the most frequent POS tag for each unique lowercase token that appears in the unparsed input files.

   Param: `filelist` is a list of `str` file names

   Return: a `dict` containing `str` keys corresponding to tokens, and `str` values giving the most frequent part of speech tag assigned to that token in the input files

2. `run_mft_baseline(testfilelist, mftdict, poslookup)`: Given a dictionary mapping tokens to their most-frequent tag in the training data, and a dictionary mapping PTB tags to Google Universal tags, predict the Google Universal POS tag of each token in the (unparsed) input files. Your function should parse the raw input files using your function from Section 1. After making the predictions, your function should calculate the accuracy of your predicted tags.

   Param: `list` of `str`; `dict` of `str -> str` keys and values; `dict` of `str -> str` keys and values

   Return: A `float` value giving the accuracy of predicted tags, i.e. the number of correctly predicted tags (based on the Google Universal tagset) in the test files divided by the total number of predicted tags in the test files.

3. Write the accuracy of your baseline model to the first line of a text file called `modelscores.txt`.

Grading: We will verify the correctness of your functions on a test input.

3 Model 1: Large vocabulary, small context window (30 Points)

In this homework we will be using the `libsvm` library to implement a multi-class prediction support vector machine with a linear kernel. Given a training set of sentences with POS-tagged tokens, we would like to train a model to predict the POS tags of words in new sentences.

Our feature space: Throughout the next three sections we will experiment with variations on the following feature set for our data. We represent each token t in our dataset as a numeric feature vector of length $V \times W$, where V is the size of our vocabulary, and W is the size of our *context window*, an odd positive integer. We use the context window to incorporate information about the words surrounding token t when we make our prediction about the POS tag for t.

Each word in our feature vocabulary is mapped to a unique integer index $1 < i \leq V$. If we give each token in our context window a position n with $0 < n < W$, the value of element $nV + i$ in our vector is equal to 1 if the n^{th} word in our context window corresponds to index i, and 0 otherwise. Our target token (for which we are making the tag prediction) is always in the center of the context window.
Our first model will use a window size $W = 3$. Our vocabulary will consist of all tokens in files within the `data/train` directory that occur at least eight times.

1. **Varying the context window and tag vocabulary.** Before feeding our data to `libsvm`, we want the ability to vary our models based on the size of the context window and the vocabulary. We will write a function to pre-process the data from our raw POS-tagged files into a format that is straightforward to convert to `libsvm` input.

Required:

- Define the following method: `prep_data(dirname, outfile, windowsize, tagmap, vocab)`: This function reads data from all raw POS-tagged files in directory `dirname` and converts to an intermediate format in `outfile`. The format of `outfile` consists of one `<tag> <context window>` pair per line, tab separated, with the words in the context window separated by a space. See an example of the first few lines of the output file obtained by passing a directory containing only the tagged sentence below as input with `windowsize=3`:

  Input: `The/DT cat/NN is/VB black/JJ ./.`

  Output:

  ```
  DET	<s> The cat
  NOUN	The cat is
  VERB	cat is black
  ADJ	is black .
  .	black . <s>
  ```

  Your function should replace words in the input files that are not within your `vocab` with the token `<UNK>`. Also, if the context window extends beyond a sentence boundary, it should pad with `<s>` tokens. The context window should never span more than one sentence. The part-of-speech tags as written in `outfile` correspond to the values in your `tagmap`. The order in which your function prints lines to `outfile` should be deterministic; given the same `dirname` containing the same files, the order of the samples output to `outfile` should be the same each time the function is called.

  Param:

  - `dirname` gives the relative path to a directory containing one or more raw POS-tagged files.
  - `outfile` is the relative path to a file where you will output the prepared data.
  - `windowsize` is an odd positive integer.
  - `tagmap` is a directory mapping PTB tags to Google Universal tags.
  - `vocab` is a set containing `str` tokens.

- Run your function `prep_data` twice, using each of the `data/test` and `data/train` directories as input, with the following settings:

  - `outfile` is `mod1_train_prepped` or `mod1_test_prepped` as appropriate
  - `windowsize=3`
  - `tagmap` is the dictionary returned by your function `create_mapping` on the file `data/en-ptb.map`
  - `vocab` is the set of all tokens that occur at least eight times in files in the `data/train` directory

  Please turn in the first 100 lines of your output for the `data/train` directory in a file called `hw3_3_1.txt`

Grading: We will check the correctness of your output file `hw3_3_1`.

2. **Converting to `libsvm` format.** Next we will take the file as output by the function `prep_data` above and convert it to `libsvm` sparse vector format. You can read more about the input format that `libsvm` requires in its documentation saved in the `cis530` directory at `/home1/c/cis530/Software/libsvm-3.20/README`.

   To vectorize the input, you should use the method described above (see **Our feature space**). In the function descriptions below, V corresponds to the size of our feature set (i.e. our original vocabulary, plus the tokens `<s>` and `<UNK>`) and W corresponds to the size of the context window. Note that the number of non-zero features for each sample vector is equal to the context window size.

Required:

- Define the following method: `convert_to_svm(preppedfile, outfile, posset, vocab)` that takes the `preppedfile` as output by `prep_data` and writes its sparse vector format to `outfile` in libsvm format:

  `<label> <index1>:<value1> <index2>:<value2> <index3>:<value3>`

  - Take the feature set to be the set of all tokens in `vocab`, plus `<s>` and `<UNK>`. The size of the feature set is V.
  - Map each feature in the *alphabetically sorted* feature set to increasing integer indices i from 1 to V. **NOTE** We initialize the indices at 1 because this is required for `libsvm` format.
  - For each line in `preppedfile`, output a numeric label and sparse feature vector where the $nV + i^{th}$ feature is 1 if the n^{th} word (counted from 0) in the context window corresponds to feature i, and 0 otherwise.
  - In `libsvm` format, each `<label>`, `<index>`, and `<value>` is numeric. So we must also convert the possible labels listed in `posset` to numeric indices. Use the same method as for `vocab`, giving increasing integer indices to each item in the alphabetically sorted `posset` starting from 1.

  Param:

  - `preppedfile` is the relative path to a file as output by `prep_data` above.
  - `outfile` is the relative path to your output file
  - `posset` is a set of strings, corresponding to the POS tags in the Universal tag set
  - `vocab` is a set of strings, corresponding to the feature set

- Run your function on the prepped files you created in the previous section with the following parameters:
  - `outfile` is `mod1_train.svm` or `mod1_test.svm` as appropriate
  - `posset` is the set of part of speech tags in the Google Universal tag set
  - `vocab` is the set of all tokens that occur at least eight times in files in the `data/train` directory

Grading: We will check the correctness of your `convert_to_svm` function on test input.

3. **Training and testing a model** Finally, we will write a function that uses `libsvm` to train and test a POS tagging model using our formatted data as output in the previous section.

   If you choose to develop locally, you can download `libsvm` from `https://www.csie.ntu.edu.tw/~cjlin/libsvm/`. It comes packaged with a Python module called `svmutil` that we will use to train and test our models. In order to import `svmutil` into your script, you will need to add the `libsvm` directory to your `PYTHONPATH`. On `biglab` this can be accomplished by running the following command from the command line:

   ```
   > PYTHONPATH=$PYTHONPATH:/home1/c/cis530/Software/libsvm-3.20/python
   ```

   or modifying your `PYTHONPATH` directly in your `.bashrc` file. Once you've done this, you should be able to import `svmutil` in your homework code using the line

   ```
   from svmutil import *
   ```

   Required: Write a function `train_test_model(train_datafile, test_datafile)` that trains a `libsvm` model using the training data in `train_datafile` and tests it on the data in `test_datafile`.

Use the Python API to `libsvm`, `lsvmutil`, to train and test your model. **Do not** use the command line interface to `libsvm`.

When training your model, you should specify a linear kernel and use shrinking (`'-t 0 -e .01 -m 1000 -h 0'`). All other settings should be the `libsvm` default.

Write the accuracy of Model 1 to the second line of your text file called `modelscores.txt`.

Param: both parameters are `str` relative paths to `libsvm`-formatted data as output by `convert_to_svm`.

Return: Return the list of predicted labels, accuracy tuple, and list of decision values as returned by a call to `svmutil.svm_predict`. See the `libsvm` Python README file for more details on what is included in these return values.

Grading: We will grade this section based upon our ability to correctly run your `train_test_model` implementation on test input.

4 Model 2: Small vocabulary, larger context window (15 Points)

We will repeat the steps above, but this time our model will use a wider context window and very small vocabulary comprising only the most frequent words in our corpus. Since the most frequent words in English tend to be function words, our objective is to see how much these function words alone can tell us about parts of speech within an entire sentence.

Required:

- Train and test a model that uses only the 100 most-frequently-occurring words in the `data/train` directory as its vocabulary, and has a context window of length 7.

  Write the accuracy of Model 2 to the second line of `modelscores.txt`. Also turn in the first 100 lines of your `outfile` from `convert_to_svm(prepped_train_file, outfile, posset, vocab)` as a file called `hw_3_4.txt`

Grading: We will grade this section based upon the accuracy of your Model 2 implementation as reported in `modelscores.txt`, and the correctness of your output file `hw_3_4.txt`.

5 Model 3: Lexical Substition (20 Points)

Our final model will be similar to Model 1, but instead of replacing out-of-vocabulary words with an `<UNK>` token, we will replace them with the most similar word in the vocabulary based on cosine similarity of `word2vec` word embedding vectors. Here we wish to see if replacing infrequent words with similar, but more frequent, words, will lead to improved POS tagging performance.

Pre-trained `word2vec` vectors for most of the tokens in our train and test sets are stored in a file `cis530/hw3/data/w2vec_hw3`. Words are stored one per line, as tab-separated string and comma-separated vector pairs.

Required:

- Write a function `lex_sub_dict(w2vecdict, freqwordvocab, infreqwordvocab)` that creates a lexical substitution dictionary with words from `infreqwordvocab` as keys and words from `freqwordvocab` as values. For each key from `infreqwordvocab`, the value is the word from `freqwordvocab` that has the closest vector representation in `w2vecdict` in terms of cosine similarity. You may write your own cosine similarity function or use one from an existing library, as long as that library can be imported on `biglab`.

  Param:

  - `w2vecdict` is a `dict` with `str` keys and `numpy array` values, as read from the `word2vec_hw3` file
  - `freqwordvocab` and `infreqwordvocab` are sets of `str`

  Return: a `dict` with `str` keys and `str` values

- There is a list of words, written one per line, in the file `cis530/hw3/data/wordlist.txt`. For each word in this list, find the word in your `vocab` from Section 2 (words occurring more than 5 times in the training data) that has the greatest cosine similarity.

  Write these word pairs, one per line, to the file `hw3_5.txt`. Pairs should be tab separated and appear in the same order as in `wordlist.txt`.

- Write a function `prep_data_lexsub(dirname, outfile, windowsize, tagmap, vocab, w2vecdict)` that works exacty as the function `prep_data` from Section 2, except that out-of-vocabulary words are replaced with their nearest neighbor from the vocabulary based on cosine similarity of vectors stored in `w2vecdict`. Only if an out-of-vocabulary word is not present as a key in `w2vecdict` should it be replaced with the token `<UNK>`.

  Param:

  - `dirname` gives the relative path to a directory containing one or more raw POS-tagged files.
  - `outfile` is the relative path to a file where you will output the prepared data.
  - `windowsize` is an odd positive integer.
  - `tagmap` is a directory mapping PTB tags to Google Universal tags.
  - `vocab` is a set containing `str` tokens.
  - `w2vecdict` is a `dict` of `str` keys and `numpy array` values

- Train and test your lexical substitution model using the data in our `train` and `test` directories as input. Include in the vocabulary any word that occurs eight or more times in the files in `data/train` and use a window size of 3. Write the accuracy of the model to the third line of `modelscores.txt`.

Grading: We will grade the correctness of your file `hw3_5.txt` and the accuracy of your model as reported in `modelscores.txt`.

6 Model Comparison (10 Points)

In the last section we will compare the performance of the two most accurate models from the previous sections.
Required:

- Write a function `compare_results(actual, modAlabels, modBlabels)` that compares the accuracy of predicted labels from two models against a list of actual labels.

  Param: `actual`, `modAlabels`, and `modBlabels` are lists containing actual test data labels, labels as predicted by Model A, and labels as predicted by Model B respectively.

  Return: A 2 x 2 list giving the count of samples correctly and incorrectly classified by Model A and Model B as follows:

```
[[# Model A correct and Model B correct, # Model A correct and Model B incorrect],
 [# Model A incorrect and Model B correct, # Model A incorrect and Model B incorrect]]
```

- Run your function `compare_results` with the most accurate of the three models as Model A and the second-most accurate of the three models as Model B. Print the output to a file called `hw3_6.txt` using the command:

  ```
  compmat = compare_results(actual, modAlabels, modBlabels)
  with open(outfile, 'w') as fout:
      print >> fout, compmat
  ```

Grading: We will grade the correctness of your function and your output in `hw3_6.txt`.