CS 6320.002: Natural Language Processing
Fall 2019

Homework 2 – 90 points
Issued 09 Sept. 2019
Due 8:30am 23 Sept. 2019

Deliverables: A tarball or zip file containing your code and your PDF writeup.

0 Getting Started

Make sure you have downloaded the data for this assignment:

- `train.txt`, a training set of movie reviews
- `test1.txt`, a testing set of movie reviews
- `dict_of_affect.txt`, a sentiment lexicon

Make sure you have installed the following libraries:

- NLTK, `https://www.nltk.org/`
- Numpy, `https://numpy.org/`
- Scikit-Learn, `https://scikit-learn.org/stable/`

1 Tokenization and Preprocessing – 32 points

First we need to load the training data. Open a new file `sentiment.py` and **write a function `load_corpus(corpus_path)` that opens the file at `corpus_path` and reads in the data**. The corpus files `train.txt` and `test.txt` have the following format: each line consists of a "snippet" (generally a single sentence, but sometimes more) and a label (0 for negative, 1 for positive), separated by a tab. `load_corpus(corpus_path)` should return a list of tuples `(snippet, label)`, where `snippet` is a string and `label` is an int.

This time we are not going to ignore tokenization and preprocessing! If you look at the training and testing files, you will see that some tokenization and preprocessing has already been done. Most punctuation has been split off as separate tokens, and the words have all been converted to lower case. But there are still a few things we can do.

First, tokenization is mostly done, except that single quotes are not separated out. This is likely due to the single quote looking exactly the same as the apostrophe (notice that contractions are not split up). For example,

> shot perhaps 'artistically' with handheld cameras
> this film is your typical 'fish out of water' story

Notice that there are three possible ways a word can still have single quotes attached: at the front of the word, at the end of the word, and at both the front and end. Import the `re` library and **write three regular expressions that match these three types**

of words with single quotes attached. One thing to keep in mind is that sometimes what looks like a single quote at the front of a word is actually an apostrophe that is part of the word (eg. 'em, 'tis, '70s); your regular expression for front single quotes should avoid splitting up such words. You can test your regular expressions using `re.search()` and the two example snippets. If you match 'artistically', 'fish, and water', you know your regular expressions are working correctly. Save the regular expressions as global variables.

Now that you can match single quote words, **write a function `tokenize(snippet)` that takes a string `snippet` as input and returns a list of tokens**. The function should first call `re.sub()` three times, once for each of your three saved regular expressions, to insert a space between the single quotes and the words they are attached to. For example,

'artistically' → ' artistically '
'fish → ' fish
water' → water '

If you used groups in your regular expressions, this step should be pretty easy; if you didn't, you might want to go back and modify your regular expressions to use groups. After you have inserted the spaces, you can simply call `split()` and return the resulting tokenized list.

Next up is some preprocessing. There are two things we want to take care of. First, some snippets contain words inside square brackets. For example,

[but it's] worth recommending
the good and different idea [of middle-aged romance]

By convention, this means that the words inside the square brackets weren't written by the original author, but were added by an editor to clarify the meaning of the snippet. We want to remove the square brackets and tag the words inside with a meta tag EDIT_. For example,

EDIT_but EDIT_it's worth recommending
the good and different idea EDIT_of EDIT_middle-aged EDIT_romance

Write a function `tag_edits(tokenized_snippet)` that takes as input a list of tokens and returns the same list, but with any square brackets removed and the words inside tagged with EDIT_. Iterate through `tokenized_snippet` until you find a word with an open square bracket in front. Remove the bracket and tag the word, and continue tagging words until you find a word with a close square bracket at the end, after which you stop tagging words. Keep in mind that it is possible for a snippet to have more than one set of square brackets; make sure `tag_edits()` finds them all.

Writeup Question 1.1: There are two other ways we could have handled the square brackets. We could have deleted those words completely, or we could have removed the square brackets but left the words untagged. Why did we do it the way we did (with tags)? What are some pros and cons of the three different ways of handling the square brackets? Give at least one pro or one con for each way.

The last preprocessing step we'll do is tagging negation. We want to tag words following a negation word with a meta tag NOT_. There are several things to think about.

Besides "not", what are some other words that should trigger negation tagging? "No", "never", "cannot", words that end with "n't". **Write a regular expression that matches all negation words and save it as a global variable.** Keep in mind that contractions are not split up, so the "n't" clitic is still attached to another word (this is usually the case with tokenization for sentiment analysis).

When should we stop tagging? If we encounter the word "but", or a similar word, like "however" or "nevertheless". Or if we reach the end of a sentence, ie. encounter ".", "?", or "!". Save these negation-ending tokens in a global variable.

Writeup Question 1.2: What data type did you use to save the negation-ending tokens? Explain your choice.

There is another class of words that should end negation tagging: comparative adjectives and adverbs. For example, "It could not be clearer that blah blah blah." We don't want to tag "that blah blah blah" as negated, so we want comparatives like "clearer", "better", etc., to also be negation-ending words. How do we identify such words? The only thing their strings have in common is that they end in "er". But of course, there are other types of words that end in "er", like "cinematographer" or "scriptwriter", so how do we know which it is? Part-of-speech tagging to the rescue! We will cover POS tagging later, but for now we can just use `nltk.pos_tag()`.

We are ready to write the negation tagger. First import the `nltk` library. **Write a function `tag_negation(tokenized_snippet)` that does the following**:

- Make a copy of `tokenized_snippet` and remove any meta tags already added (ie. EDIT_tags).
- Call `nltk.pos_tag()` on this tagless copy of `tokenized_snippet`. You will get back a list of tuples `(word, pos)`.
- Put any removed meta tags back onto the words in this list of tuples.
- Iterate through the list of tuples until you find a negation word.
- Do a quick corner case check. If the negation word is "not" and the next word is "only", ie. "not only", don't tag anything.
- Otherwise tag the words following the negation word with NOT_until you find either a negation-ending token or a comparative. We can check for comparatives by looking at the POS instead of the word. The two POS tags for comparatives are JJR for adjectives and RBR for adverbs.
- Return the fully-tagged list of `(word, pos)` (we will use the POS tags again later, so we might as well keep them).

As with square brackets, it is possible for a snippet to have more than one group of negated words, so make sure `tag_negation()` finds them all.

2 A Basic Unigram Classifier – 16 points

Now let's put together a simple sentiment classifier with unigram features. Unless otherwise stated, the code in this section can go in the main function of your script, if you use one. Use the functions from the previous section, `load_corpus()`, `tokenize()`, `tag_edits()`, and `tag_negation()`, to process all the snippets in the training corpus. Recall that `load_corpus()` returns a list of `(snippet, label)` tuples. Make sure you keep each snippet associated with its label as you're doing preprocessing.

Next we need to set up the feature dictionary. For a unigram feature dictionary, we want to get the vocabulary (ie. unique tokens) for the training set. Each feature is assigned a position (ie. an index) in the feature vector, so for a given word in the vocabulary, we want to be able to look up its associated position/index. **Create a dictionary where the keys are the unique tokens in the preprocessed training set, and the values are the positions/indices associated with each token.** Make sure you skip words tagged with EDIT_, since they were not written by the original author of the snippet. Note that the assignment of features to indices is completely arbitrary.

Using the feature dictionary, we can convert the preprocessed snippets into feature vectors. Import the `numpy` library and **write a function `get_features(preprocessed_snippet)` that takes the list of tuples from the last preprocessing step, `tag_negation()`, and returns a feature vector in a Numpy array.** The function should do the following:

- Use the `numpy.zeros()` function to initialize a Numpy array of length $|V|$, where V is the vocabulary, that contains all zeros.
- Iterate through the words in `preprocessed_snippet`, looking up the index of each word, using the feature dictionary, and incrementing the value of the array at that index. Again, skip words tagged with EDIT_.
- Return the completed array.

Now back to the main function. **Create two variables `X_train` and `Y_train`, which are Numpy arrays that hold the training feature vectors and the training labels, respectively.** Use `numpy.empty()` to initialize `X_train` of size $m \times |V|$, where m is the number of training examples, and `Y_train` of length m. Then iterate through the preprocessed training set, generating each snippet's feature vector using `get_features` and copying it into the appropriate row of `X_train`; similarly, copy each snippet's label into the appropriate position in `Y_train`.

The last thing we need to do with the features is to normalize them. Normalizing features is important because, depending on your feature design, some features may have much larger or smaller values than others. This isn't so much the case with unigram features, but imagine if we were using a mix between counts and binary (0-1) features – the counts would be much larger than the binary features. But the classifier doesn't know that the two types of features are different; it just thinks that the binary features are less expressive for some reason.

Write a function `normalize(X)` that takes a feature matrix and normalizes the feature values to be in the range [0, 1]. Recall that each row in `X` corresponds to a training example, and each column corresponds to a feature. Iterate through the columns of X and do the following for each column:

- Find the minimum and maximum value in the column.
- For each value f in the column, replace it with $\frac{f - \min}{\max - \min}$.

This is called min-max normalization. Return the normalized matrix.

Now we are finally ready to train a sentiment classifier. Import the class `GaussianNB` from the `sklearn.naive_bayes` module. Instantiate a `GaussianNB` and call `fit()` on it with the finished `X_train` and `Y_train`.

3 Evaluating a Classifier – 24 points

How do we evaluate our sentiment classifier? The standard metrics for any classification problem are precision, recall, and f-measure. **Write a function `evaluate_predictions(Y_pred, Y_true)` that takes two Numpy arrays of labels and returns a tuple of floats `(precision, recall, fmeasure)`.** The function should do the following:

- Use three counter variables to count the number of
  - True positives (tp), true label is 1 and predicted label is 1
  - False positives (fp), true label is 0 and predicted label is 1
  - False negatives (fn), true label is 1 and predicted label is 0
- Calculate precision, recall, and f-measure
  - Precision (p) $= \frac{tp}{tp + fp}$
  - Recall (r) $= \frac{tp}{tp + fn}$
  - F-measure $= 2\frac{p \cdot r}{p + r}$

Writeup Question 3.1: Looking at the formulae for precision, recall, and f-measure, what does each of them measure? Why do we need all three of them?

Now let's test your trained `GaussianNB`. Load and preprocess `test.txt` and create `X_test` and `Y_true`. Generate predictions `Y_pred` by calling `predict()` on the trained model with `X_test`, then use `evaluate_predictions()` on them.

Writeup Question 3.2: What is the performance of the `GaussianNB` model?

Let's compare the Naive Bayes model to a logistic regression model. Import the class `LogisticRegression` from the `sklearn.linear_model` module. Instantiate a `LogisticRegression` model and train it and test it as we did with the Naive Bayes model.

Writeup Question 3.3: What is the performance of the `LogisticRegression` model? Discuss which model performed better on this data and why you think that might be.

Finally, let's look at which unigrams are the most important for this sentiment classification task. Recall that a logistic regression model has a weight vector w that is used to scale up or scale down different features. This weight vector is stored as an internal variable `coef_` in the `LogisticRegression` class. **Write a function `top_features(logreg_model, k)` that takes a trained `LogisticRegression` model and an int k and returns a list of length k containing tuples `(word, weight)`.** The function should do the following:

- Access `logreg_model.coef_`, which is a Numpy array of size $1 \times |V|$.
- Convert this array into a list of tuples `(index, weight)` and sort in descending order by the absolute value of `weight`.
- Use the feature dictionary to replace each `index` with the corresponding unigram that it is associated with.
- Return the sorted list of words and weights.

Writeup Question 3.4: Report the top 10 features of your `LogisticRegression` model. Why do we sort by the absolute value of `weight`, rather than the actual value? Explain why it is important to do so and what it means in terms of the features.

4 Using a Lexicon – 16 points

The last part of this assignment uses `dict_of_affect.txt`, a sentiment lexicon called the Dictionary of Affect in Language. If you look at this file, you will see that each line consists of a word and three metrics, separated by tabs. The metrics are

- Activeness, the level of activation or arousal of a word (eg. "sleep" vs. "run")
- Evaluation, the pleasantness of a word (eg. "happy" vs. "sad")
- Imagery, the concreteness of a word (eg. "flower" vs. "freedom")

Each word in the DAL was scored by a team of linguists. Positive scores mean the word is active, pleasant, or concrete; negative scores mean the word is passive, unpleasant, or abstract.

Write a function `load_dal(dal_path)` that reads in the lexicon in the file `dal_path` and returns a dictionary where the keys are words, and the values are tuples of floats `(activeness, evaluation, imagery)`. Note that the first line of the file is a header and should not be included in the dictionary.

We would like to use the DAL scores as additional features for our classifiers. But the DAL gives word-level scores, and the input to our classifiers is snippets, so we need to aggregate the scores of the words in a snippet into a single score to use as a feature. **Write a function `score_snippet(preprocessed_snippet, dal)` that takes a fully preprocessed snippet (ie. list of tuples `(word, pos)`) and the DAL dictionary and returns a tuple of `(activeness, pleasantness, imagery)` scores for the whole snippet.** For a given metric, the score of the whole snippet should be the average score among words in the snippet that are found in the DAL dictionary. As before, skip words tagged with EDIT_. For words that are tagged with NOT_, invert their scores (ie. multiply by -1).

Modify the `get_features()` function to add three additional features to the generated feature vector. The specific changes you need to make are

- The length of the feature vector is now $|V| + 3$, instead of just $|V|$.
- `get_features()` needs to call `score_snippet` and put the three scores at the end of the feature vector.

You also need to update the size of `X_train` and `X_test` to be $m \times (|V| + 3)$.

Now train, test, and examine the feature weights of a new `LogisticRegression` model using the newly expanded feature set.

Writeup Question 4.1: What is the performance of the new model with extra features? What are its top 10 features? How does it compare with the old model without extra features? Why do you think that might be?

5 Meta Questions – 2 points

Writeup Question 5.1: How long did this homework take you to complete (not counting extra credit)?

Writeup Question 5.2: Did you discuss this homework with anyone?

6 Extra Credit – 10 points

You may have noticed that the DAL is not very long, and there are a lot of words that aren't in it. For extra credit, augment the DAL's coverage using WordNet. WordNet is an ontology that encodes semantic relations between concepts called synsets (ie. groups of synonyms). The idea is, if a word is not found in the DAL, but its synonym is, then we can use the score of the synonym instead; failing that, if its antonym is in the DAL, we can invert its score and use that.

WordNet is available through NLTK: install the NLTK corpora, `https://www.nltk.org/data.html`, and import the `wordnet` class from `nltk.corpus`. You can find its documentation at `http://www.nltk.org/howto/wordnet.html`.

Modify `score_snippet()` as follows. It first attempts to look up each word in the DAL as before. If it doesn't find the word, it looks up the word's synonyms using WordNet and tries to look up each of them. If it doesn't find any of the synonyms, it looks up the word's antonyms using WordNet and tries to look up each of them. If it still doesn't find anything, then it skips the word as before.

Note that WordNet allows you to specify the part of speech of the word you are looking up. This is helpful for words that can be more than one part of speech, like "content" and "run". Conveniently, we already have POS tags for our snippets; inconveniently, WordNet uses a different set of POS tags, which is given in the documentation, so you will have to **convert the NLTK POS tags into WordNet POS tags**.

Writeup Question 6.1: Train, text, and examine the feature weights of a new `LogisticRegression` model using the new version of `score_snippet()`. What is the performance of the new model with extra features? What are its top 10 features? How does it compare with the model that used the old version of `score_snippet()`?