CS 6320.002: Natural Language Processing
Fall 2021

Homework 2 Programming Component – 55 points
Issued 15 Sept. 2021
Due 11:59pm CDT 29 Sept. 2021

Deliverables: Your completed `hw2.py` file, uploaded to Gradescope.

0 Getting Started

Make sure you have downloaded the data for this assignment:

- `train.txt`, a training set of movie reviews
- `test.txt`, a testing set of movie reviews

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 Data and Preprocessing – 15 points

In this assignment, we will train a unigram logistic regression classifier for predicting the sentiment (positive or negative) of a movie review. Open the provided skeleton code `hw2.py` in your favorite text editor.

First we need to load the training data. The provided 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. Tokenization and some preprocessing have already been done: most punctuation has been split off as separate tokens, and the words have all been converted to lower case.

Fill in the function `load_corpus(corpus_path).` The argument `corpus_path` is a string. The function should do the following:

- Open the file at `corpus_path` and load the data.
- Return a list of tuples `(snippet, label)`, where `snippet` is a list of strings (words), and `label` is an int, eg. `[(["I", "love", "computer", "science", "."], 1), (["Homework", "sucks", "!"], 0)]`.

There is one preprocessing step we need to do: negation tagging. We want our model to treat the "good" in "not good" differently from the "good" in "very good." To do this, we will 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", "nor," and "cannot" are all negation words; for convenience, these words are collected in the global variable `negation_words`. Words that end with the "-n't" clitic are also negation words.

Fill in the function `is_negation(word).` The argument `word` is a string. The function should do the following:

- Return `True` if `word` is one of the words in `negation_words`.
- Otherwise, check if `word` ends in "-n't"; return `True` if so and `False` if not.

When should we stop doing negation tagging? If we encounter the word "but", or a similar word, like "however" or "nevertheless". For example, we don't want to do negation tagging on the word "good" in the sentence "I didn't like the salad, but the soup was good." If we reach the end of a sentence, ie. encounter ".", "?", or "!", we also want to stop doing negation tagging. For convenience, negation-ending words are collected in the global variable `negation_enders`, and sentence-ending punctuation is collected in the global variable `sentence_enders`.

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; for now, we will just use the NLTK function `nltk.pos_tag()`.

Now let's write the negation tagger.

Fill in the function `tag_negation(snippet).` The argument `snippet` is a list of strings (words). The function should do the following:

- Perform part-of-speech tagging on `snippet` by first converting `snippet`, which is a list of words, into a single string, and then calling `nltk.pos_tag()` on it.
- Iterate through the list of words, calling `is_negation()` on each word until you encounter a negation word.
- Do a quick corner case check. If the negation word is "not" and the next word is "only", ie. "not only", we don't want to do negation tagging.
- Otherwise, replace the words following the negation word with their negation-tagged versions (ie. `'good'` becomes `'NOT_good'`). Always use the same meta-tag `'NOT_'`, regardless of the negation word that triggered the tagging.
- Stop tagging when you find either sentence-ending punctuation, a negation-ending word, or a comparative. You can check for sentence-ending punctuation and negation-ending words using the global variables `sentence_enders` and `negation_enders`, respectively; you can check for for comparatives by looking at the POS tag corresponding to each word: the tags for comparatives are `'JJR'` for adjectives and `'RBR'` for adverbs.
- Return the negation-tagged list of words, eg. `["I", "do", "n't", "NOT_like",`

`"NOT_this", "NOT_movie", "."]`.

2 A Basic Unigram Classifier – 25 points

Now let's put together a simple sentiment classifier with unigram features. The first thing we need to do is set up the feature dictionary. Since the features are unigrams, ie. words, we want to get the vocabulary, ie. the set of unique words, in the training set. Each unique word is then assigned a position (ie. an index) in the feature vector. The goal of the feature dictionary is that, for a given word, we want to be able to look up its associated position/index in the feature vector.

Fill in the function `get_feature_dictionary(corpus).` The argument `corpus` is a list of tuples `(snippet, label)`. The function should do the following:

- Initialize a position/index counter to 0.
- Iterate through each `snippet` in `corpus` and each `word` in each `snippet`. If the word has not previously been seen, assign it the current position/index and then increment the position/index counter.
- Return a dictionary where the keys are the unique words in `corpus` (strings), and the values are the positions/indices assigned to those words (ints), eg. { `"apple":0, "banana":1, "carrot":2` }.

Using this feature dictionary, we can convert snippets into feature vectors.

Fill in the function `vectorize_snippet(snippet, feature_dict).` The argument `snippet` is a list of strings, and the argument `feature_dict` is a dictionary {`word:index`}. The function should do the following:

- Use the Numpy function `numpy.zeros()` to initialize a Numpy array of length equal to the size of `feature_dictionary`, which contains all zeros.
- Iterate through the words in `snippet`, looking up the index of each `word` using `feature_dictionary`, and incrementing the value of the array at that index. This creates a vector of word occurrence counts.
- Return the completed array, eg. `numpy.array([0, 0, 0, 0, 2, 1, 3, 1, 0])`.

Fill in the function `vectorize_corpus(corpus, feature_dict).` The argument `corpus` is a list of tuples `(snippet, label)`, and the argument `feature_dict` is a dictionary {`word:index`}. The function should do the following:

- Create two Numpy arrays `X` and `Y` to hold the training feature vectors and the training labels, respectively. Use `numpy.empty()` to initialize `X` of size $n \times d$, where n is the number of snippets in `corpus` and d is the number of features in `feature_dict`, and `Y` of length n.
- Iterate through `corpus`, using `vectorize_snippet()` to get each snippet's feature vector and copying it into the appropriate row of `X` (you can use Python's array slicing syntax to do this); similarly, copy each snippet's label into the appropriate position in `Y`.

- Return a tuple `(X, Y)`, eg. `(numpy.array([[4, 0, 0, 0, 2, 1, 3, 1, 0], [0, 4, 5, 3, 0, 0, 3, 1, 2], [0, 3, 4, 5, 0, 0, 0, 3, 1]]), numpy.array([0, 1, 1]))`.

There's actually one last thing we need to do with the features: normalization. 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. What we want is to normalize the features so that every feature has the same maximum and minimum values, and we don't get some features with a much larger range of values than other features.

We will normalize the feature values in `X` to be in the range $[0, 1]$ using min-max normalization. Recall that each row in `X` corresponds to a training snippet, and each column corresponds to a feature.

Fill in the function `normalize(X)`. The argument `X` is a Numpy array. The function should do the following:

- Iterate through the columns (features) of `X`.
- Find the minimum and maximum values in a column.
- For each value f in the column, replace it with $\frac{f - \min}{\max - \min}$.
- Make sure the function doesn't fail for any min/max values!

Now everything is ready to train a sentiment classifier!

Fill in the function `train(corpus_path)`. The argument `corpus_path` is a string. The function should do the following:

- Load the training corpus at `corpus_path` and perform negation tagging on each snippet.
- Construct the feature dictionary.
- Vectorize the corpus and normalize the feature values.
- Instantiate a Scikit-Learn `LogisticRegression` model and use its `fit()` method to train it on the vectorized corpus.
- Return a tuple `(model, feature_dict)`, where `model` is the trained `LogisticRegression` and `feature_dict` is the feature dictionary constructed earlier (we will need it for testing).

You can check your work using the first line of the `main()` function, which will use the functions you have filled in so far to train a sentiment classifier.

3 Evaluating a Classifier – 15 points

How do we evaluate our sentiment classifier? The standard metrics for any classification problem are precision, recall, and f-measure.

Fill in the function `evaluate_predictions(Y_pred, Y_test).` The arguments `Y_pred` and `Y_test` are Numpy arrays, where `Y_pred` holds the labels predicted by our model, and `Y_test` holds the true labels from the test dataset. 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) $= \dfrac{tp}{tp + fp}$
  - Recall (r) $= \dfrac{tp}{tp + fn}$
  - F-measure $= 2\dfrac{p \cdot r}{p + r}$
- Return a tuple of floats `(precision, recall, fmeasure)`.

Now let's test the trained model.

Fill in the function `test(model, feature_dict, corpus_path).` The argument `model` is a trained `LogisticRegression`, the argument `feature_dict` is a dictionary, and the argument `corpus_path` is a string. The function should do the following:

- Load the test corpus at `corpus_path` and perform negation tagging on each snippet.
- Vectorize the test corpus and normalize the feature values. (Note that it is possible for words to appear in the test corpus that are not in the training corpus! Make sure `vectorize_corpus()` and `vectorize_snippet()` do not fail on such inputs.)
- Use the classifier's `predict()` method to obtain its predictions on the test inputs.
- Evaluate the predictions and return a tuple of floats `(precision, recall, fmeasure)`.

You can use the second line of `main()` to check your work so far.

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.

Fill in the function `get_top_features(logreg_model, feature_dict, k=1).` The argument `logreg_model` is a trained `LogisticRegression` model, the argument `feature_dict` is a dictionary, and the argument `k` is an int indicating how many features to return. The function should do the following:

- Access `logreg_model.coef_`, which is a Numpy array of size $1 \times d$.

- Convert this array into a list of tuples `(index, weight)` and sort in descending order by the absolute value of `weight`.
- Use `feature_dict` to replace each `index` with the corresponding unigram that it is associated with.
- Return a list of the top k words and weights, eg. `[("apple", 6.36), ("banana", -5.46), ("carrot", 2.69)]`.

All done! You can use the last line of `main()` to check your work.