CS 6320.002: Natural Language Processing
Fall 2019

Homework 3 – 92 points
Issued 23 Sept. 2019
Due 8:30am 07 Oct. 2019

Deliverables: A tarball or zip file containing your code and your PDF writeup. Note that beginning in this assignment, you **must** use the filename and function signatures in these instructions; failure to do so will result in a 50% penalty on your score.

Warning! This assignment uses a large dataset. Training and decoding can take a long time. Start early so you have time to train and run your model and decode its predictions!

0 Getting Started

Make sure you have downloaded the test data file, `test.txt`.

Make sure you have installed the following libraries:

- NLTK, `https://www.nltk.org/`
- NLTK Corpora, `https://www.nltk.org/data.html`
- Numpy, `https://numpy.org/`
- Scikit-Learn, `https://scikit-learn.org/stable/`
- Scipy, `https://www.scipy.org/`

Open a new file `[your netid]_memm.py` and import `numpy`, `scipy`, and `sklearn`. At the bottom of your file, put the following two lines:

```
if __name__== '__main__':
    main()
```

Finally, define the `main()` function.

1 Loading the Data – 6 points

We are going to train a bigram maximum entropy Markov model to do part of speech tagging. First we need to load the training data. Add another import statement: `from nltk.corpus import brown`. In `main()`, load the Brown corpus: `brown_sentences = brown.tagged_sents(tagset='universal')`. The variable `brown_sentences` is a list of lists (sentences), where each sentence is a list of `(word, tag)` tuples. For example, `brown_sentences[0][0]` is the first `(word, tag)` pair of the first sentence in the corpus.

Writeup Question 1.1: Notice that we are using the Universal Dependencies tagset instead of Penn Treebank (17 tags instead of 45). What are some advantages and disadvantages of using a smaller tagset? Give at least one advantage and one disadvantage.

It will be convenient to seperate the input (words) from the labels (tags). Split `brown_sentences` into two lists, `train_sentences` and `train_tags`; both should be lists of lists, but the lists in one contain the words and the lists in the other contain the tags.

Finally, we will identify rare words, which we will need to know in the next section. Create a set `rare_words` that contains all words that occur fewer than 5 times in the training data. (I assume we all know how to get counts by now!)

2 Generating Features – 30 points

Now we need to generate the features. For ease of implementation, we are using simple features, not the pairwise feature shown in the slides. We will break down the features into two types: word features and word n-gram features. Each type of feature will have its own generation function that returns a list of features (strings).

Write a function `word_ngram_features(i, words)`. The argument `i` (int) is the index of the current word to generate features for, and `words` is a list containing all the words in the sentence (ie. `words[i]` is the current word w_i). The function should return a list containing the following features:

- `'prevbigram-[x]'`, where `[x]` is the previous word w_{i-1}
- `'nextbigram-[x]'`, where `[x]` is the next word w_{i+1}
- `'prevskip-[x]'`, where `[x]` is the next previous word w_{i-2}
- `'nextskip-[x]'`, where `[x]` is the next next word w_{i+2}
- `'prevtrigram-[x]-[y]'`, where `[x]` is w_{i-1} and `[y]` is w_{i-2}
- `'nexttrigram-[x]-[y]'`, where `[x]` is w_{i+1} and `[y]` is w_{i+2}
- `'centertrigram-[x]-[y]'`, where `[x]` is w_{i-1} and `[y]` is w_{i+1}

You will need to check for corner cases where $i \pm 1$ and/or $i \pm 2$ are negative or larger than the length of the sentence. In these cases, use meta-words `'<s>'` and `'</s>'` to fill in those features.

Writeup Question 2.1: The two features `'prevskip-[x]'` and `'[nextskip-[x]'` are skip-bigrams, ie. a bigram where the two words, w_i and w_{i-1} are not consecutive. Why do we need skip-bigram features when w_2 is already captured by the word trigram features? What is the advantage of a skip-bigram over a trigram?

Write a function `word_features(word, rare_words)`**.** The argument `word` is the current word w_i, and `rare_words` is the set of rare words we made in the previous section. The function should return a list containing the following features:

- `'word-[word]'`, *only if* `word` is not in `rare_words`
- `'capital'`, *only if* `word` is capitalized
- `'number'`, *only if* `word` contains a digit
- `'hyphen'`, *only if* `word` contains a hyphen
- `'prefix[j]-[x]'`, where `[j]` ranges from 1 to 4 and `[x]` is the substring containing the first `[j]` letters of `word`

- `‘suffix[j]-[x]’`, where `[j]` ranges from 1 to 4 and `[x]` is the substring containing the last `[j]` letters of `word`

You will need to check for corner cases where `word` is short and `[j]` becomes larger than the length of `word`, in which case skip the prefix/suffix features for large values of `[j]`.

Writeup Question 2.2: What is the goal of this last set of features? (Hint: why we using `rare_words`?)

Now **write a wrapper function** `get_features(i, words, prevtag, rare_words)` that calls the two feature functions we just wrote. The function should create a single list containing all of the generated features from both feature functions. It should also add to this list the tag bigram feature, `‘tagbigram-[prevtag]’`, lowercase all features using `lower()`, and return the list.

3 Training – 30 points

We are ready to generate features for the training data. In `main()`, create a new list of lists `training_features`. Iterate through the training set and call `get_features()` for each word in each sentence to fill in its spot in `training_features`. For example, `training_features[0][0]` should be the list of features for the first word in the first sentence of the corpus. You will need to check for corner cases where $i = 0$ and there is no `prevtag`; in these cases, use the meta-tag `‘<S>’` .

Write a function `remove_rare_features(features, n)`. The argument `features` is a list of lists of feature lists (ie. `training_features`), and `n` is the number of times a feature must occur in order for us to keep it in the feature set. The function should do the following:

- Iterate through `features` and count the number of times each feature occurs.
- Create a two sets, one containing the rare features (fewer than n occurrences) and one containing the non-rare features (n or more occurrences).
- Create a new version of `features` with only the rare features removed (this can be done quickly and easily using the rare feature set and Python list comprehensions).
- Return the new version of `features` and the non-rare set (we can use it to build the feature dictionary next).

In `main()`, call `remove_rare_features()` on `training_features` with $n = 5$ and overwrite `training_features` with the new version. Use the non-rare set of features to create a feature dictionary `feature_dict`, as in the previous assignment.

Writeup Question 3.1: Why do we want to remove rare features? Give at least two reasons why we do this.

This is also a good time to create the tag dictionary. In the previous assignment, we didn't need a label dictionary because there were only two labels. Now we have 17 tags, and recall that the MEMM will output a probability distribution over the labels, so we need to assign indices to labels as well. Create a dictionary `tag_dict` where the keys are the 17 tags and the values are the indices assigned to the tags.

Now it's time to build `X_train` and `Y_train`. Recall that an MEMM is just a multiclass logistic regression; the sequence information is part of the feature vector, not part of the model itself. So a training example isn't an entire sequence of words and tags, it's a single word/tag pair; thus, the number of training examples isn't the number of sentences, it's the number of tokens.

`Y_train` is easy, so let's do that first. **Write a function `build_Y(tags)`.** The argument `tags` is a list of lists of tags (ie. `train_tags`, which we made back in Part 1). We need to flatten this list of sentences into a list of tokens. The function should do the following:

- Create an empty list `Y`.
- Iterate through `train_tags` by first iterating through the first sentence tags, then the second sentence tags, and so on
- For each tag, look up its index in the tag dictionary and append the index to `Y`.
- Convert `Y` to a Numpy array using `numpy.array()` and return it.

`X_train` is going to be harder because we are going to use a sparse matrix to represent it. The features that we generated in Part 2 are all binary features, ie. for a given token, the feature is either generated (1) or not (0). So for any given word, we want its feature vector to have value 1 at the positions of its features, and 0 for all other features. To represent this information, we can save the positions of all entries that are *not* 0, and all other positions are assumed to be 0. This representation is called a sparse matrix.

Writeup Question 3.2: Why do we want to use a sparse matrix for `X_train`? What is the advantage of a sparse matrix over a dense (normal) matrix?

We are going to use the compressed sparse row format. The idea is that we can construct a sparse matrix using three lists, `row`, `col`, and `value`, to indicate which positions are not 0: `X[row[i][col[i]] = value[i]`, and all other positions in `X` are assumed to be 0. Of course, all non-0 positions have value 1 for us, so `value` would just be a list of 1s.

Import the class `csr_matrix` from the `scipy.sparse` module, then **write a function `build_X(features)`.** The argument `features` is a list of lists of feature lists (ie. `training_features`). The function should do the following:

- Create two empty lists, `examples` and `features` – recall that in an input matrix, the rows correspond to training examples, and the columns correspond to individual features.
- Iterate through `features` as we did in `build_Y()`: first iterate through the feature lists for the first sentence, then the feature lists for the second sentence, and so on.
- Keep a counter `i` for how many feature lists (ie. how many tokens) we have seen so far (note that `i` is going to be used as an index into our matrix `X`, so it should start at 0).
- For the `i`th feature list, iterate through the features in the list. For each feature
  - Append `i` to `examples`.

  - Look up the feature in the feature dictionary and append its assigned index to `features`.
- Create a third list, `values`, which has length equal to `examples` and `features`, and contains all 1s.
- Convert all three lists to Numpy arrays.
- Create a `csr_matrix` using `examples`, `features`, and `values` and return it.

In `main()`, use `build_X` and `build_Y` to create `X_train` and `Y_train`. We are now finally ready to train the model.

Import the `LogisticRegression` class from the `sklearn.linear_model` module. In `main()`, instantiate a `LogisticRegression` model with the following options:

- `class_weight=balanced`
- `solver=saga`
- `multi_class=multinomial`

Train the model with `fit()` as in the previous assignment.

4 Testing – 24 Points

Write a function `load_test(filename)`. The argument `filename` is a string containing the path to the test file, where each line in the test file is a sentence. The function should return a list of lists, where each sublist is a sentence (a line in the test file), and each item in the sentence is a word. The test sentences have already been tokenized, so you can simply call `split()` to get the list of words. Make sure you strip any extra whitespace from the words/sentences and then return the list. Then, in `main()`, load the test data from `test.txt`.

We will implement Viterbi decoding to get the highest-probability sequence of tags for each test sentence. Recall that the Viterbi algorithm needs to know the value of $p(t|w_i, t')$ for all triples (t, w_i, t'), so we need to compute those values ahead of time and save them so that Viterbi can look them up.

Write a function `get_predictions(test_sentence, model)`. The argument `test_sentence` is a list containing a single list of words (we continue to use a nested list because the functions we wrote for generating features for the training data expect a list of list(s)), and `model` is a trained `LogisticRegression` model. The function should do the following:

- Use `numpy.empty()` to create a matrix `Y_pred` of size $(n-1) \times T \times T$, where n is the number of words, T is the number of tags in the tagset.
- Iterate through the words in the sentence, skipping the first word (w_0). For each word w_i, where $i > 0$, do the following:
  - Iterate through the tags in the tagset.
  - For each possible previous tag t_j, use `get_features()` and `build_X()` to create an input matrix `X`. As a sanity check, `X` should be of size $1 \times F$, where F is the number of features in the feature dictionary.

    - Use your trained `LogisticRegression` model to call `predict_log_proba()` on `X`. This will return a log-probability distribution of size $1 \times T$; set `Y_pred[i,j]` to hold this log-probability distribution.
- For the first word in the sentence, build an input matrix `X` using the only possible previous tag: the meta-tag `'<S>'`, and save the model's predicted log-probability distribution in a separate Numpy array `Y_start`.
- Return `Y_pred` and `Y_start`.

Now **write a function** `viterbi(Y_start, Y_pred)`. The function should do the following:

- Use `numpy.empty()` to create two matricies, `V` and `BP`, both of size $n \times T$.
- Initialize `V[0, j]` to be `Y_start[0,j]` and `BP[0,j]` to be -1 for all j. (The choice of base for the log doesn't matter.)
- Iterate through the rows of `Y_pred`. For each row i (which corresponds to w_{i+1}), do the following:
    - Iterate through the candidate tags. For each candidate tag t_k, do the following:
        * Compute the value of `V[i,j]` + `Y_pred[i,j,k]` for each possible previous tag t_j.
        * Set `V[i+1,k]` to be the maximum of these values.
        * Set `BP[i+1,k]` to be the argmax over these values (you can use `numpy.argmax()` to do this).
- Create an empty list `backward_indices` to hold the decoded tag sequence.
- Find `index`, the argmax over `V[n]`, and append it to `backward_indices`.
- Overwrite `index` with the value at `BP[n, index]`, append it to `backward_indices`, and decrement n.
- Repeat until $n = 1$ is reached. (Recall that we set `BP[0,j] = -1` for all `j` to indicate the meta-tag `'<S>'`, so we don't need to include $n = 0$ in our decoding here.)
- Reverse the order of `backward_indices` so that it is no longer backward, replace each index in the list with its corresponding tag using the tag dictionary, and return the list of tags.

Finally, in `main()`, iterate through each test sentence, using `get_predictions()` and `viterbi()` to decode the highest-probability sequence of tags.

Writeup Question 4.2: Put your predicted tag sequence for each of the test sentences.

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 – 8 points

Add the following additional features from the textbook to either `get_features()` or one of its two helper functions, `word_ngram_features()` and `word_features()`:

- `‘word-[word]-prevtag-[prevtag]’`
- `‘allcaps’`, for all uppercase letters
- `‘wordshape-[x]’`, where `[x]` is an abstracted version of the word where lowercase letters are replaced with ‘x’, uppercase letters are replaced with ‘X’, and digits are replaced with ‘d’. Note that word shape features should *not* be lowercased, unlike the other features.
- `‘short-wordshape-[x]’`, like the normal word shape feature, except consecutive character types are merged into a single character (eg. “hello” → `‘xxxxx’` → `‘x’` and “Hello” → `‘Xxxxx’` → `‘Xx’`). Again, short word shape features should *not* be lowercased.
- `‘allcaps-digit-hyphen’`, for words that contain at least one digit, at least one hyphen, and all uppercase letters
- `‘capital-followedby-co’`, for words that are capitalized and where the words “Co.” or “Inc.” occur at most three words after the word.

Retrain the model using the new, larger feature set, and use it to predict tags for the test sentences again.

Writeup Question 6.1: Put your new predicted tag sequence for each of the test sentences. Are they better or worse than the predictions from Part 4, and if so, in what way?