CS 6320.002: Natural Language Processing
Fall 2020

Homework 3 – 55 points
Issued 28 Sept. 2020
Due 11:59pm CDT 12 Oct. 2020

Deliverables: Your completed `hw3.py` file, uploaded to Gradescope.

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/`

We are going to train a bigram maximum entropy Markov model (MEMM) to do part of speech tagging.

The function `load_training_corpus()` loads the training data for you. We are using the Brown corpus with the Universal Dependencies tagset. The function returns a tuple of lists `(sents, tags)`, where `sents` is a list of sentences, ie. lists of strings (words), and `tags` is a list of tag sequences, ie. lists of strings (tags).

Note that the Brown corpus is very large (just over 57k sentences), so we will not use the whole thing, or training would take forever.

1 Generating Features – 15 points

The first thing we need to do is generate features for our input. 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 is a single word/tag pair, not an entire sentence, and we will write functions to generate features at the word level.

For ease of implementation, we are using simple features, not the pairwise features shown in the lecture slides. We break down the features into two types: n-gram features and word property features. The n-gram features capture word sequence information, while the word property features capture the word's general characteristics.

Fill in the function `get_ngram_features(words, i)`**.** The argument `words` is a list of strings (words), and the argument `i` is an int indicating the the current word to generate features for. The function should return a list containing the following strings:

- `'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}

Make to check for corner cases where $i \pm 1$ and/or $i \pm 2$ are out of range of the sentence length. In these cases, use meta-words `'<s>'` and `'</s>'` to fill in those features. Also, note that the square brackets in `[x]` and `[y]` are *not* part of the features; they are only to indicate that you are meant to replace them with the actual words.

Fill in the function `get_word_features(word)`**.** The argument `word` is a string. The function should return a list containing the following strings:

- `'word-[word]'`
- `'capital'`, if `word` is capitalized
- `'allcaps'`, if `word` is all capital 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'.
- `'short-wordshape-[x]'`, which is like the previous word shape feature, except consecutive character types are merged into a single character (eg. "Hello" $\rightarrow$ `'Xxxxx'` $\rightarrow$ `'Xx'`).
- `'number'`, if `word` contains a digit
- `'hyphen'`, 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`

Make sure to check for corner cases where `[j]` is out of range of the length of `word`, in which case skip the prefix/suffix features for those values of `[j]`. As before, the square brackets are *not* part of the features; they indicate that you are meant to replace them with actual words or values.

Finally, **fill in the function** `get_features(words, i, prevtag)`**.** The argument `words` is a list of strings, the argument `i` is an int, and the argument `prevtag` is a string (the tag of w_{i-1}). The function should do the following:

- Call the two feature functions we wrote earlier and combine them into a single list.
- Add the tag bigram feature, `'tagbigram-[prevtag]'`.
- Convert all features to lowercase *except* the word shape and short word shape features.
- Return the full list of features.

2 Training – 25 points

We are ready to work with the training data.

First, we will assume we have generated the features for the entire training corpus (we will do this in a later function).

Fill in the function `remove_rare_features(corpus_features, threshold=5)`. The argument `corpus_features` is a list of lists, where each sublist represents a sentence and contains the individual feature lists for each word in the sentence (ie. the outputs of individual calls to `get_features()`). For example, `corpus_features[0][0]` is the feature list for the first word of the first sentence in the corpus. The function should do the following:

- Iterate through `corpus_features` and count the number of times each feature occurs.
- Create a two sets, one containing the rare features (fewer than `threshold` occurrences) and one containing the common features (`threshold` or more occurrences).
- Create a new version of `corpus_features` with the rare features removed. Make sure you don't change the nested structure of `corpus_features`.
- Return a tuple `(corpus_features, common_features)` containing the new version of `corpus_features` and the common feature set (we can reuse it to build the feature dictionary).

Removing rare features is helpful because it reduces the overall size of the model by reducing the total number of features, and therefore the feature vector length. It also helps prevent overfitting on rare features that may be strongly correlated with some tag.

Now we can generate the feature dictionary. This is also a good time to create the tag dictionary. In HW2, we didn't need a label dictionary because there were only two labels, positive and negative, which we assigned to 1 and 0, respectively. Now we have 12 tags, and recall that the MEMM outputs a probability distribution over labels, so we need to assign an index/position to each tag.

Fill in the function `get_feature_and_label_dictionaries(common_features, corpus_tags)`. The argument `common_features` is the set of common features from `remove_rare_features()`, and the argument `corpus_tags` is a list of lists, where each sublist represents a sentence and contains the tags for that sentence (for example, `corpus_tags[0][0]` is the tag for the first word of the first sentence in the corpus). The function should do the following:

- Create a feature dictionary and assign an index to each feature in `common_features`, as in HW2.
- Create a tag dictionary and assign an index to each of the 12 tags.
- Return a tuple `(feature_dict, tag_dict)`.

Now we have the tools to build `X_train` and `Y_train`. Again, note that 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 in the training corpus, it's the number of tokens.

`Y_train` is easy and pretty much the same as in HW2, so let's do that first.

Fill in the function `build_Y(corpus_tags, tag_dict)`. The argument `corpus_tags` is a list of lists of tags, and the argument `tag_dict` is the dictionary from `get_feature_and_label_dictionaries()`. The function should do the following:

- Iterate through `corpus_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 a list. This "flattens" the nested list input format into a non-nested, single list for output.
- Convert the completed list 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 1 are all binary, ie. for a given word, the feature is either present (1) or not (0). So for any given word, we want its feature vector to have value 1 at the positions of its generated features, and 0 for all other features. Since most features will be 0 for a given word, we can simply save the positions of all entries that are 1, and all other positions are assumed to be 0. This representation is called a *sparse matrix*. Sparse matrices are much smaller in memory than dense ("normal") matrices, which is important for us because the training corpus is so large.

We are going to use the compressed sparse row (CSR) format for matrices. 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` will just be a list of 1s.

Fill in the function `build_X(corpus_features, feature_dict)`. The argument `corpus_features` is a list of lists of feature lists, and the argument `feature_dictionary` is the dictionary from `get_feature_and_label_dictionaries()`. The function should do the following:

- Create two empty lists, `rows` and `cols` – recall that in an input matrix, the rows correspond to training examples, and the columns correspond to features.
- Iterate through `corpus_features` as we iterated through `corpus_tags` in `build_Y()`.
  - For each feature, append the index of the training example it belongs to to `rows`.
  - Look up the feature in `feature_dictionary` and append its index to `cols`.
- Create a third list, `values`, which has length equal to `rows` and `cols`, and contains all 1s.
- Convert `rows`, `cols`, and `values` to Numpy arrays, and return a `csr_matrix` instantiated with these three lists.

Now we have all the tools to train the model.

Fill in the function `train(proprtion=1.0)`. The argument `proportion` is a float indicating how much of the Brown corpus to use in training (because the corpus is so large, training on the whole thing would take several hours). The function should do the following:

- Use `load_training_corpus()` to get the list of sentences and the list of tag sequences. Make sure to pass `proportion`. Recall that each sentence is a list of words, and each tag sequence is a list of tags.
- Build the list of lists of feature lists `corpus_features` by iterating through the words in the corpus and using `get_features()` to generate each word's feature list. Make sure to check for the corner case where $i = 0$ and there is no `prevtag`; in these cases, use the meta-tag `'<S>'` .
- Use `remove_rare_features()`, `get_feature_and_label_dictionaries()`, `build_X()`, and `build_Y()`.
- Instantiate a `LogisticRegression` model with the following options:
  - `class_weight=balanced`
  - `solver=saga`
  - `multi_class=multinomial`
- Train the model with `fit()` as in HW2.
- Return a tuple `(model, feature_dict, tag_dict)`.

You can use the first line of `main()` to check that your code runs.

3 Decoding – 15 Points

Now that the model is trained, we can use it to predict part of speech tags for test sentences.

The function `load_test_corpus(corpus_path)` loads the test data for you. The function returns a list of lists, where each sublist represents a sentence contains the words (strings) of that sentence.

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.

Fill in the function `get_predictions(test_sent, model, feature_dict, reverse_tag_dict)`. The argument `test_sent` is a list containing a single list of words (we use a nested list here because the functions we wrote for generating features earlier expect a nested list as input), `model` is a trained tagger, `feature_dict` is a feature dictionary, and `reverse_tag_dict` is a dictionary from indices to tags (simply assume we have this for now; we will generate the dictionary in a later function). 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 in the `test_sent`, and 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:

  - Initialize an empty list `features` to hold the generated features for this word. Note that we need to generate features for all possible values of t' (aka. `prevtag`), so we will generate more than one feature list per word.
  - Iterate through the tags in the tagset, and for each tag t_j, use `get_features()` to generate features using t_j as `prevtag` and append the feature list to `features`.
  - Use `build_X()` to build an input matrix. Note that we have batched all inputs corresponding to a single word in the test sentence together; as a sanity check, `X` should be of size $T \times F$, where F is the number of features in the feature dictionary.
  - Use `model`'s `predict_log_proba()` method on `X` to predict log probability distributions over the tagset. Again, because we have batched the inputs, this step should return a Numpy array of size $T \times T$.
  - Copy the predicted log probabilities into the corresponding elements of `Y_pred`.
- For the first word w_0, which we skipped, there is only one possible value for `prevtag`: the start tag `'<S>'`. Generate the features and build an input matrix `X` for w_0, and save the predicted log-probability distribution in a Numpy array `Y_start`.
- Return a tuple (`Y_start`, `Y_pred`).

Now **fill in the function** `viterbi(Y_start, Y_pred)`. The arguments `Y_start` and `Y_pred` are Numpy arrays from `get_predictions()`. The function should do the following:

- Use `numpy.empty()` to create two dynamic programming tables, `V`, the main Viterbi table, and `BP`, the backpointer table, both of size $n \times T$, where n is the number of words in the `test_sent`, and T is the number of tags in the tagset.
- Initialize `V` using the base case shown in lecture, using the values already calculated in `Y_start`.
- Fill in the rest of `V` and `BP` using the recursive case shown in lecture, based on the values already calcuated in `Y_pred`.
  - Note we used a product of raw probabilities in lecture, whereas now we have log probabilities, so you need to use a sum instead.
  - Also note that Numpy provides a function `numpy.argmax()` that you can use for filling in `BP`.
- Use the completed tables `V` and `BP` to return the highest-probability tag sequence as a list of tag indices (ints). Note that the backpointers give you the tag indices in reverse order, so make sure to flip the order so that it is no longer reversed before you return the list.

Finally, **fill in the function** `predict(corpus_path, model, feature_dict, tag_dict)`. The argument `corpus_path` is a string indicating the test corpus, `model` is a trained tagger, and `feature_dict` and `tag_dict` are the feature and tag dictionaries. The function should do the following:

- Load the test data using `load_test_corpus()`.
- Create `reverse_tag_dict`, a dictionary from indices to tags, from the input `tag_dict`.
- Iterate through the sentences in the test corpus
  - For each sentence, use `get_predictions()` to get `Y_start` and `Y_pred`.

    - Use `viterbi()` to get the list of predicted tag indices.
    - Use `reverse_tag_dict` to convert the list of indices into a list of tags.
- Return a list of lists, where each sublist represents a test sentence and holds the predicted tags for that sentence.

We are done! You can use the rest of `main()` to make sure your code runs and gives reasonable outputs.