CS5740 NLP Project2 Report(Kaggle Team: thisTeam)

Names: Boliang Yang(by283), Jinwei Shen(js3559), Shenghua Li(sl3293)

1. The HMM model for Metaphor Detection

In this part of report, we manage to explain the principles and implement the model for the metaphor detection task.

1.1 Implementation of the HMM model by Viterbi Algorithm

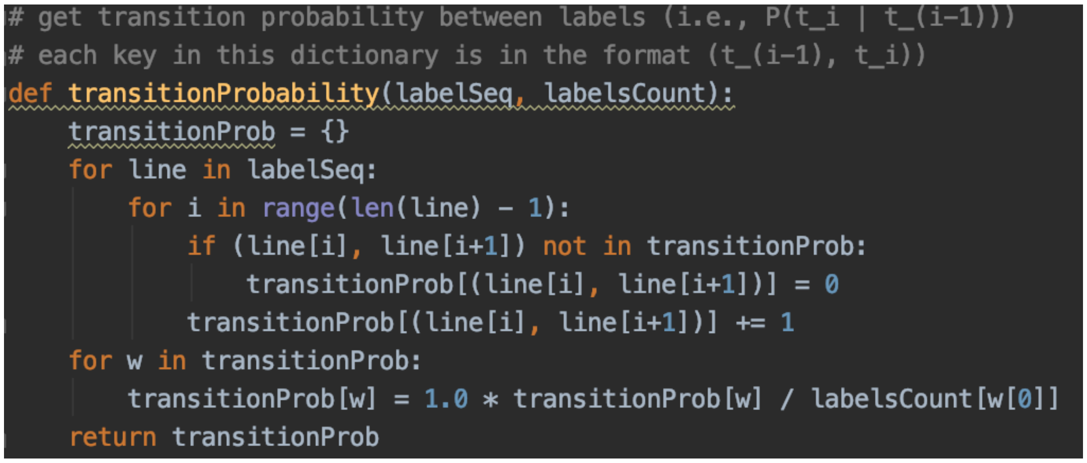
The HMM model is a generative model that combines the transition probability that how states change from one to another and the emission probability that how much probability an event could happen given a certain observation.

So, in our case, the states are rather easy, only two states: is a metaphor or not, so we can denote them by 1 and 0. And the emission probability can be obtained by calculating the frequency of a word related to a state.

For the **Pre-processing** step, since the raw datasets are in .csv file and have a characteristic: each sentence is divided by line and each token is divided by space. So, using python's 'open' function and 'split' method, we can then get all the words with their labels and give their counts in sequence. Then we can calculate the transition and emission probability. Besides, we add <s> to each line of the corpus and <ls> token to each line of labels to define the beginning state of the HMM model.

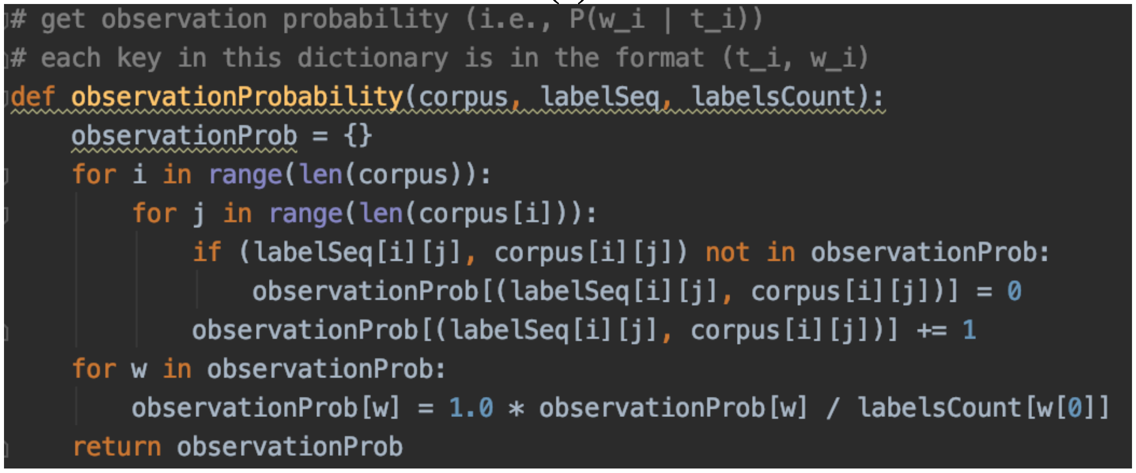
In order to calculate the transition probability, we apply the **Markov assumption**, which consider the state itself and only the previous state of it. For emission probability, we apply the **output independence assumption** and only consider each observation with its present state and observation. And to obtain the probability, we still use the maximum likelihood estimation, that is using counts of states and words to calculate the most likely probability for each case. For **unknown words （or unknown combinations of words and tags）**, we believe unknown word handling doesn't help that much since it is not a meaningful method to apply here and the model will depend more on transition probability when seen the unknown word, we first just simply provide them with a very small probability 1e-20, because 0 will cause undefine error in **log computation** (in our calculation, the minimal emission probability value for any existing combination is larger than 1e-8.). We will also test add-k-smoothing method to deal with the 0's in the emission probability later.

The code pieces for transition and observation probability's calculation are shown in Figure 1.



```python
# get transition probability between labels (i.e., P(t_i | t_(i-1)))
# each key in this dictionary is in the format (t_(i-1), t_i))
def transitionProbability(labelSeq, labelsCount):
    transitionProb = {}
    for line in labelSeq:
        for i in range(len(line) - 1):
            if (line[i], line[i+1]) not in transitionProb:
                transitionProb[(line[i], line[i+1])] = 0
            transitionProb[(line[i], line[i+1])] += 1
    for w in transitionProb:
        transitionProb[w] = 1.0 * transitionProb[w] / labelsCount[w[0]]
    return transitionProb
```

(a)



```python
# get observation probability (i.e., P(w_i | t_i))
# each key in this dictionary is in the format (t_i, w_i)
def observationProbability(corpus, labelSeq, labelsCount):
    observationProb = {}
    for i in range(len(corpus)):
        for j in range(len(corpus[i])):
            if (labelSeq[i][j], corpus[i][j]) not in observationProb:
                observationProb[(labelSeq[i][j], corpus[i][j])] = 0
            observationProb[(labelSeq[i][j], corpus[i][j])] += 1
    for w in observationProb:
        observationProb[w] = 1.0 * observationProb[w] / labelsCount[w[0]]
    return observationProb
```

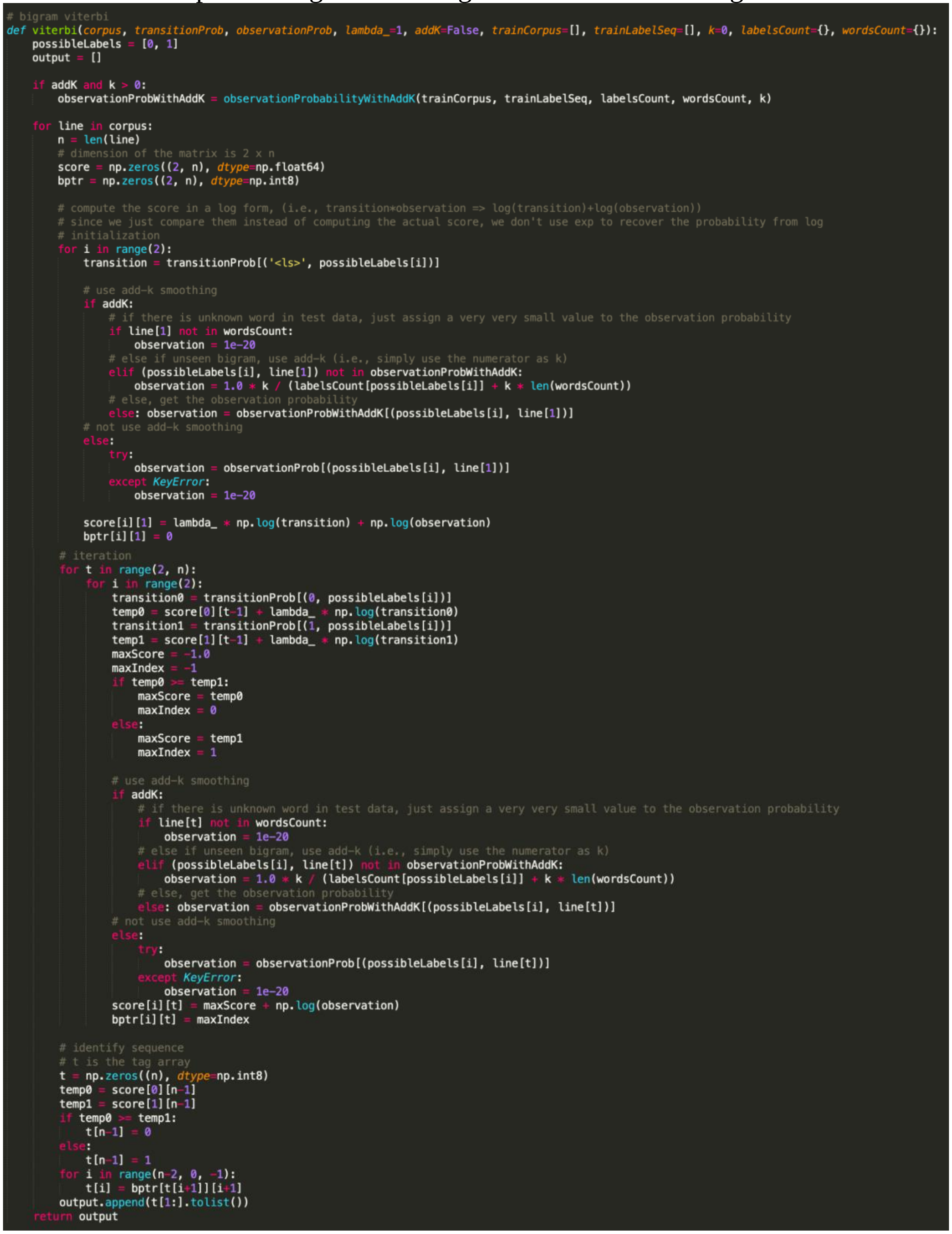
(b)

Figure 1 Codes for computing (a)transition probability (b) emission probability

Thus, we have extracted the features of the given dataset by calculating the related probabilities under our assumptions. We calculate them all in the **bigram** format, just as introduced in the J&M's book [1].

After calculating the features, we need to build the model to predict the results. Since the brute-force method will calculate all the possible sequences which will have ridiculously huge computation, we will use **Viterbi Algorithm** since we have already taken the **Markov Assumption** and **Output Independence Assumption**, which make calculating all the sequences not necessary.

Viterbi algorithm is not so hard to explain: first. We give each starting label an initial probability; then between two layers, we will calculate all the possible transition and emission probability and save the best score with its sequence for each layer. So, the overall complexity will be O(c^2*n), c is the number of labels and n is the length of a given sentence.
The codes for implementing a Viterbi algorithm are shown in Figure 2.



```python
# bigram viterbi
def viterbi(corpus, transitionProb, observationProb, lambda_=1, addK=False, trainCorpus=[], trainLabelSeq=[], k=0, labelsCount={}, wordsCount={}):
    possibleLabels = [0, 1]
    output = []

    if addK and k > 0:
        observationProbWithAddK = observationProbabilityWithAddK(trainCorpus, trainLabelSeq, labelsCount, wordsCount, k)

    for line in corpus:
        n = len(line)
        # dimension of the matrix is 2 x n
        score = np.zeros((2, n), dtype=np.float64)
        bptr = np.zeros((2, n), dtype=np.int8)

        # compute the score in a log form, (i.e., transition*observation => log(transition)+log(observation))
        # since we just compare them instead of computing the actual score, we don't use exp to recover the probability from log
        # initialization
        for i in range(2):
            transition = transitionProb[('<ls>', possibleLabels[i])]

            # use add-k smoothing
            if addK:
                # if there is unknown word in test data, just assign a very very small value to the observation probability
                if line[1] not in wordsCount:
                    observation = 1e-20
                # else if unseen bigram, use add-k (i.e., simply use the numerator as k)
                elif (possibleLabels[i], line[1]) not in observationProbWithAddK:
                    observation = 1.0 * k / (labelsCount[possibleLabels[i]] + k * len(wordsCount))
                # else, get the observation probability
                else: observation = observationProbWithAddK[(possibleLabels[i], line[1])]
            # not use add-k smoothing
            else:
                try:
                    observation = observationProb[(possibleLabels[i], line[1])]
                except KeyError:
                    observation = 1e-20

            score[i][1] = lambda_ * np.log(transition) + np.log(observation)
            bptr[i][1] = 0
        # iteration
        for t in range(2, n):
            for i in range(2):
                transition0 = transitionProb[(0, possibleLabels[i])]
                temp0 = score[0][t-1] + lambda_ * np.log(transition0)
                transition1 = transitionProb[(1, possibleLabels[i])]
                temp1 = score[1][t-1] + lambda_ * np.log(transition1)
                maxScore = -1.0
                maxIndex = -1
                if temp0 >= temp1:
                    maxScore = temp0
                    maxIndex = 0
                else:
                    maxScore = temp1
                    maxIndex = 1

                # use add-k smoothing
                if addK:
                    # if there is unknown word in test data, just assign a very very small value to the observation probability
                    if line[t] not in wordsCount:
                        observation = 1e-20
                    # else if unseen bigram, use add-k (i.e., simply use the numerator as k)
                    elif (possibleLabels[i], line[t]) not in observationProbWithAddK:
                        observation = 1.0 * k / (labelsCount[possibleLabels[i]] + k * len(wordsCount))
                    # else, get the observation probability
                    else: observation = observationProbWithAddK[(possibleLabels[i], line[t])]
                # not use add-k smoothing
                else:
                    try:
                        observation = observationProb[(possibleLabels[i], line[t])]
                    except KeyError:
                        observation = 1e-20
                score[i][t] = maxScore + np.log(observation)
                bptr[i][t] = maxIndex

        # identify sequence
        # t is the tag array
        t = np.zeros((n), dtype=np.int8)
        temp0 = score[0][n-1]
        temp1 = score[1][n-1]
        if temp0 >= temp1:
            t[n-1] = 0
        else:
            t[n-1] = 1
        for i in range(n-2, 0, -1):
            t[i] = bptr[t[i+1]][i+1]
        output.append(t[1:].tolist())
    return output
```

Figure 2 codes for Viterbi Algorithm's initialization step, iteration step and sequence identifying step

Some things to note about here in the Viterbi function:

1. During the score computation step, we compute the score in log form to avoid numerical instability, and since we just compare the results instead of getting the actual score, we don't use exponential to recover the probability from log result.

2. Since there is no 0's in transition probabilities, we just need to handle 0's in emission probabilities. And we provide two ways: first, we just assign a very small value 1e-20 to all of those 0's; second, we use add-k smoothing to deal with the 0's (for all the bigram combination, we do the add-k smoothing; for unknown words in test data, we assign a very small value 1e-20 to them).

1.2 Related Experiments on lambda selection

As suggested in the instruction, using $\lambda * \log(P(t_i|t_{i-1}))$ to replace the original transition probability may improve the model efficiency. We also agree that it is a reasonable method because the transition probabilities between any these two states (i.e., from 0 to 0, 0 to 1, 1 to 0, 1 to 1) are not quite useful, thus we should use lambda to lower the weight of the transition probability. Then, we will take an empirical method to select the best lambda.

All of the results are tested on the **validation dataset**. We test the lambda from 0 to 1 and find it performs best around 0.5.

Table 1 Experimental Results for lambda selection from 0.1 to 0.5

	$\lambda = 0.1$	$\lambda = 0.2$	$\lambda = 0.3$	$\lambda = 0.4$	$\lambda = 0.5$
Precision	0.411	0.412	0.427	0.454	0.505
Recall	0.803	0.796	0.785	0.744	0.691
F1	0.544	0.548	0.553	0.564	**0.583**
Accuracy	0.844	0.847	0.852	0.866	0.885

Table 2 Experimental Results for lambda selection from 0.6 to 1.0

	$\lambda = 0.6$	$\lambda = 0.7$	$\lambda = 0.8$	$\lambda = 0.9$	$\lambda = 1.0$
Precision	0.521	0.545	0.558	0.568	0.605
Recall	0.662	0.621	0.590	0.548	0.434
F1	0.582	0.581	0.574	0.558	0.506
Accuracy	0.890	0.896	0.898	0.899	0.901

At last, we consider the best overall result for lambda = 0.5 and its overall F1 is 0.5833647 for validation set. Thus, we will use this parameter in the HMM model for the test data. We achieve the score of 0.61057 on Kaggle.

1.3 Related Experiments on add-k-smooth for emission probability

As noted in viterbi function, we use add-k to do the smoothing on emission probability. Based on the best lambda we chose above (lambda = 0.5), we test through the k value from 0 to 1 with increasing 0.1 each time. The results are in the following Table 3 and 4.

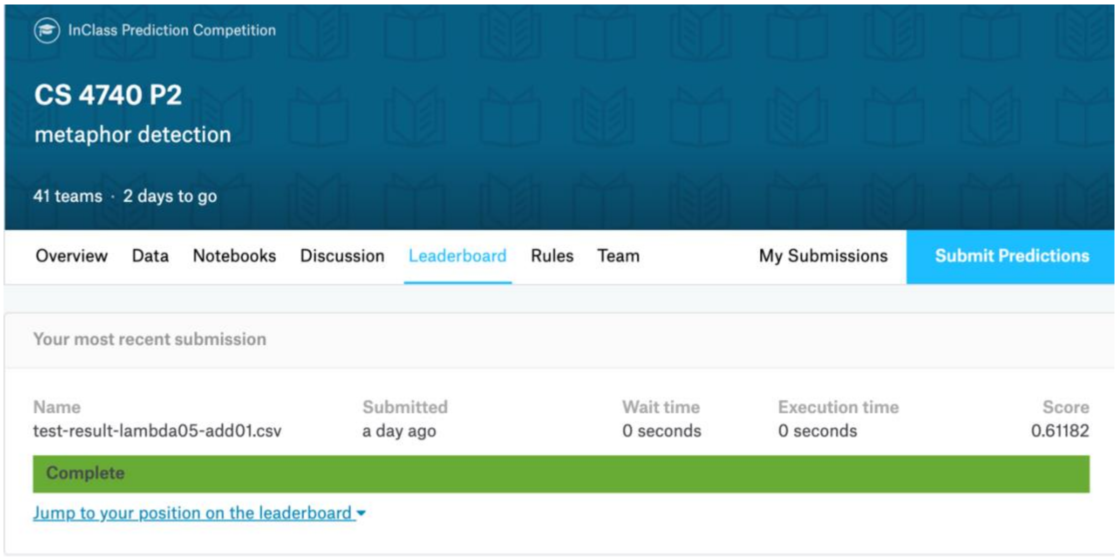
Table 3 Experimental Results for add-k-smoothing from 0 to 0.5 with lambda 0.5

	$k = 0$	$k = 0.1$	$k = 0.2$	$k = 0.3$	$k = 0.4$	$k = 0.5$
Precision	0.505	0.510	0.519	0.521	0.530	0.533
Recall	0.691	0.682	0.667	0.657	0.649	0.641
F1	0.583	**0.584**	0.583	0.581	0.582	0.582
Accuracy	0.885	0.887	0.889	0.890	0.892	0.893

Table 4 Experimental Results for add-k-smoothing from 0.6 to 1.0 with lambda 0.5

	$k = 0.6$	$k = 0.7$	$k = 0.8$	$k = 0.9$	$k = 1.0$
Precision	0.537	0.540	0.542	0.524	0.532
Recall	0.630	0.619	0.619	0.604	0.604
F1	0.580	0.578	0.578	0.566	0.566
Accuracy	0.894	0.895	0.895	0.890	0.892

From the experiment results, we choose k = 0.1 for its highest F1-score. We also submit this model to the Kaggle, the score is **0.61182**, which is a little higher than just choosing a small probability. So it explains that add-k-smoothing method can still work well in this task. Thus, our final choice is lambda = 0.5 and k = 0.1.



Your most recent submission

Name	Submitted	Wait time	Execution time	Score
test-result-lambda05-add01.csv	a day ago	0 seconds	0 seconds	0.61182

Complete

Jump to your position on the leaderboard

Figure 4 Screenshot for Kaggle Submission

2. A second model for Metaphor Detection

In this section, we go through a new model for metaphor detection. We will try to combine different features extracted from dataset and use the multinomial logistic regression classifier in the experiment.

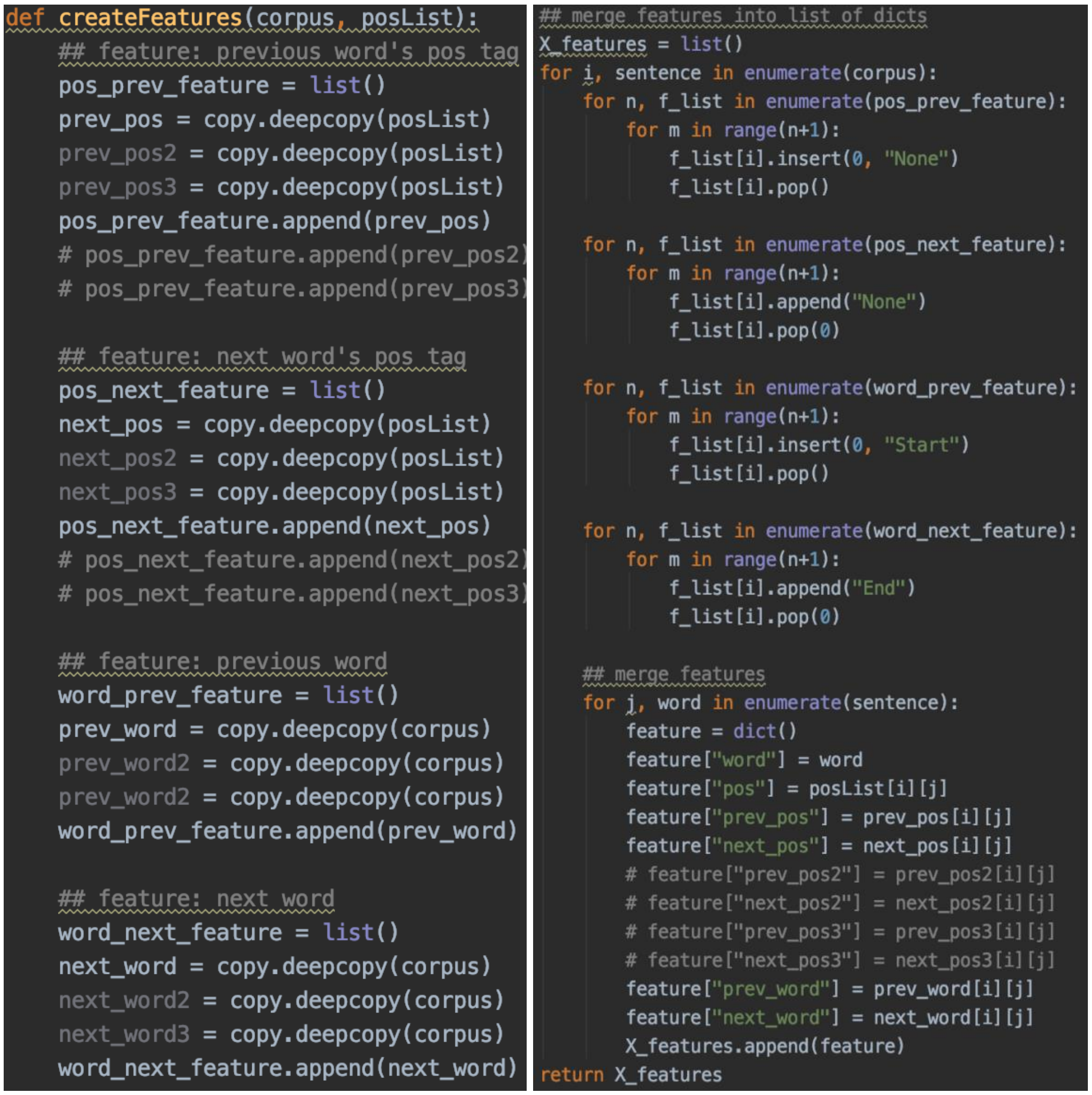
2.1 Implementation of the second model by Viterbi Algorithm

In order to combine and test as much efficient features as possible for the experiment, we test from a relatively little features to more details. We will first explain the logic of our **feature selection**. We will first explain the methodology in this section. For listing and analysis of results, they will be in the section 2.2.

Since we should still perform a Viterbi algorithm for the model as the instructions ask, we start thinking about find some features related to transition or emission probability. At first, we only calculate the POS feature probability and calculate it by MaxEnt to replace the transition and emission probability in the Viterbi algorithm. Then we find it not very efficient for the metaphor detection, so we only replace the transition probability with our feature probability, this time it works very well. In the following experiments, we think changing the window size and considering the words near the given word can also improve features. This choice is similar to J&M's textbook [1].

We build the final model in this procedure (the previous models just lack some parts or features of our final model, so we only describe the hardest one here): We count the previous POS and next POS for every word, the window size is 3 in our final choice. Then we think the features of how words transfer from each other is also very important. So, we also count the previous and next words for every word and use DictVectorizer from sklearn library to turn them into numerical. Still, logisticRegression model in sklearn library can calculate the probability for the features. Then we combine them with the emission probability as we did in model 1 to calculate the final score by Viterbi algorithm.

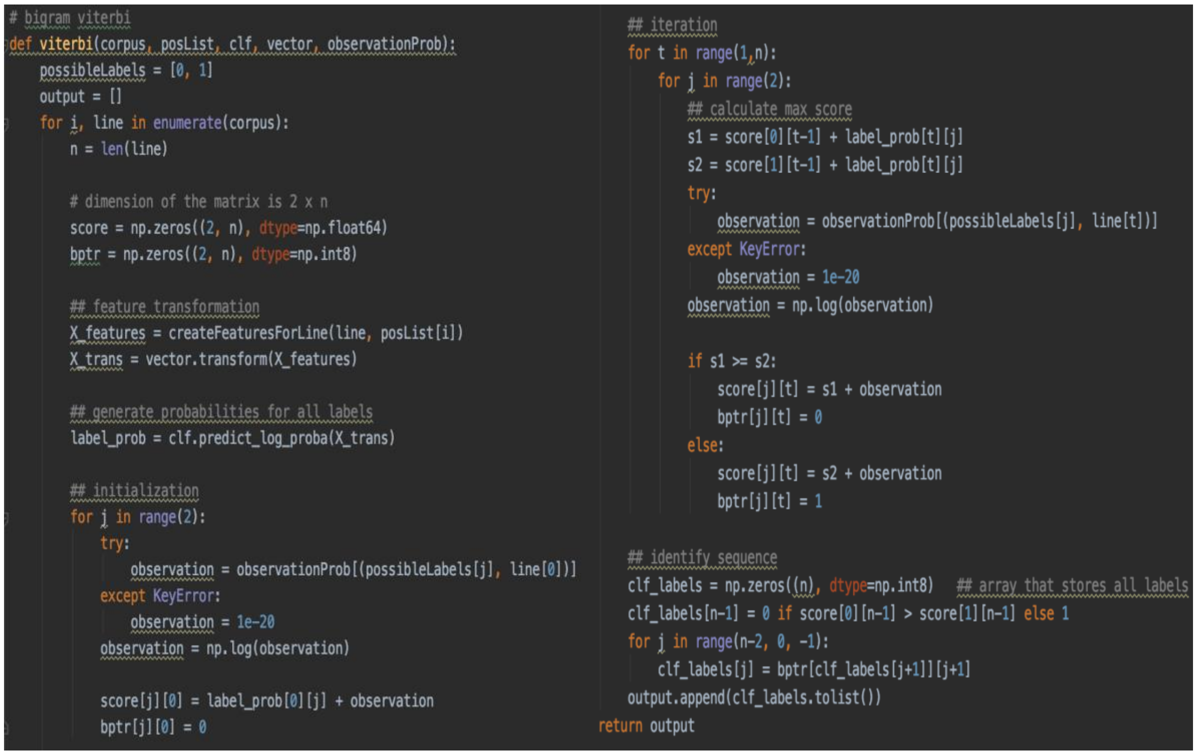
The codes for the procedure mentioned above is in Figure 5:



```python
def createFeatures(corpus, posList):
    ## feature: previous word's pos tag
    pos_prev_feature = list()
    prev_pos = copy.deepcopy(posList)
    prev_pos2 = copy.deepcopy(posList)
    prev_pos3 = copy.deepcopy(posList)
    pos_prev_feature.append(prev_pos)
    # pos_prev_feature.append(prev_pos2)
    # pos_prev_feature.append(prev_pos3)

    ## feature: next word's pos tag
    pos_next_feature = list()
    next_pos = copy.deepcopy(posList)
    next_pos2 = copy.deepcopy(posList)
    next_pos3 = copy.deepcopy(posList)
    pos_next_feature.append(next_pos)
    # pos_next_feature.append(next_pos2)
    # pos_next_feature.append(next_pos3)

    ## feature: previous word
    word_prev_feature = list()
    prev_word = copy.deepcopy(corpus)
    prev_word2 = copy.deepcopy(corpus)
    prev_word2 = copy.deepcopy(corpus)
    word_prev_feature.append(prev_word)

    ## feature: next word
    word_next_feature = list()
    next_word = copy.deepcopy(corpus)
    next_word2 = copy.deepcopy(corpus)
    next_word3 = copy.deepcopy(corpus)
    word_next_feature.append(next_word)
```

```python
## merge features into list of dicts
X_features = list()
for i, sentence in enumerate(corpus):
    for n, f_list in enumerate(pos_prev_feature):
        for m in range(n+1):
            f_list[i].insert(0, "None")
            f_list[i].pop()

    for n, f_list in enumerate(pos_next_feature):
        for m in range(n+1):
            f_list[i].append("None")
            f_list[i].pop(0)

    for n, f_list in enumerate(word_prev_feature):
        for m in range(n+1):
            f_list[i].insert(0, "Start")
            f_list[i].pop()

    for n, f_list in enumerate(word_next_feature):
        for m in range(n+1):
            f_list[i].append("End")
            f_list[i].pop(0)

    ## merge features
    for j, word in enumerate(sentence):
        feature = dict()
        feature["word"] = word
        feature["pos"] = posList[i][j]
        feature["prev_pos"] = prev_pos[i][j]
        feature["next_pos"] = next_pos[i][j]
        # feature["prev_pos2"] = prev_pos2[i][j]
        # feature["next_pos2"] = next_pos2[i][j]
        # feature["prev_pos3"] = prev_pos3[i][j]
        # feature["next_pos3"] = next_pos3[i][j]
        feature["prev_word"] = prev_word[i][j]
        feature["next_word"] = next_word[i][j]
        X_features.append(feature)
return X_features
```

(a) Feature Extraction



```python
# bigram viterbi
def viterbi(corpus, posList, clf, vector, observationProb):
    possibleLabels = [0, 1]
    output = []
    for i, line in enumerate(corpus):
        n = len(line)

        # dimension of the matrix is 2 x n
        score = np.zeros((2, n), dtype=np.float64)
        bptr = np.zeros((2, n), dtype=np.int8)

        ## feature transformation
        X_features = createFeaturesForLine(line, posList[i])
        X_trans = vector.transform(X_features)

        ## generate probabilities for all labels
        label_prob = clf.predict_log_proba(X_trans)

        ## initialization
        for j in range(2):
            try:
                observation = observationProb[(possibleLabels[j], line[0])]
            except KeyError:
                observation = 1e-20
            observation = np.log(observation)

            score[j][0] = label_prob[0][j] + observation
            bptr[j][0] = 0

        ## iteration
        for t in range(1,n):
            for j in range(2):
                ## calculate max score
                s1 = score[0][t-1] + label_prob[t][j]
                s2 = score[1][t-1] + label_prob[t][j]
                try:
                    observation = observationProb[(possibleLabels[j], line[t])]
                except KeyError:
                    observation = 1e-20
                observation = np.log(observation)

                if s1 >= s2:
                    score[j][t] = s1 + observation
                    bptr[j][t] = 0
                else:
                    score[j][t] = s2 + observation
                    bptr[j][t] = 1

        ## identify sequence
        clf_labels = np.zeros((n), dtype=np.int8)   ## array that stores all labels
        clf_labels[n-1] = 0 if score[0][n-1] > score[1][n-1] else 1
        for j in range(n-2, 0, -1):
            clf_labels[j] = bptr[clf_labels[j+1]][j+1]
        output.append(clf_labels.tolist())
    return output
```

(b) Viterbi Algorithm

Figure 5 the codes of (a) feature extraction and (b) Viterbi algorithm for model 2

2.2 Experiment results for the second model

In this section, we will go through our experiments' results and test different features and settings for different models and try to find an efficient one to be our final choice.

The six models we built have the following characteristic: **each model is built on the previous one**.

The characteristics of them can be listed as:

1. Use feature probability to replace transition and emission probability with POS window size = 1;
2. Use feature probability and observation probability, POS window size = 1;
3. Increase POS window size to 2, others same as 2;
4. Include previous word as a new feature, others same as 3;
5. Include next word as a new feature, others same as 4;
6. Increase the POS window size to 3

In fact, we also have tried to replace only transition probability with our feature probability, and the F1 score is very low, about only 9% because it has a tendency to identify a word as non-metaphor. So, we do not consider building a model like this.

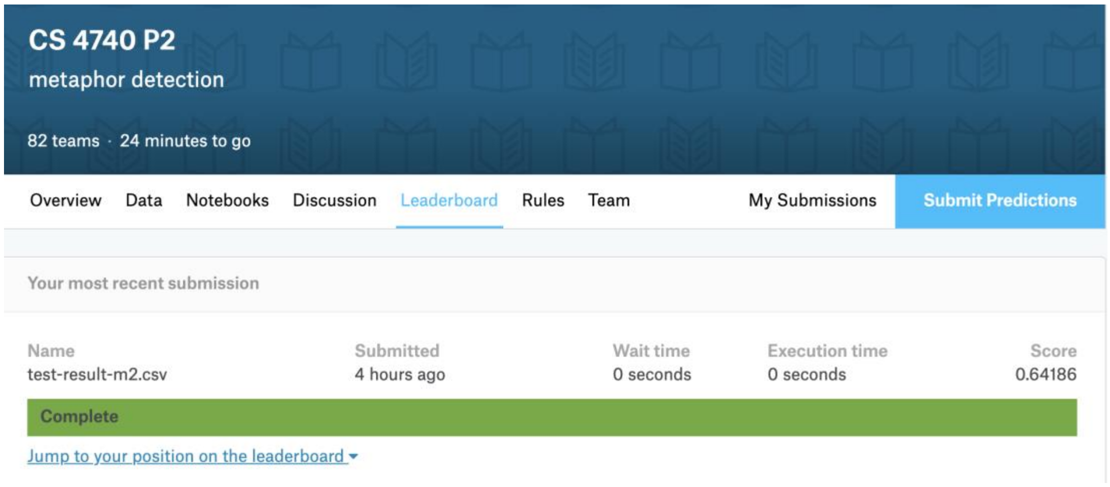
The results for this experiment are shown in Table 5: (test set can only provide F1 score, other results are all from validation set)

Table 5 Experimental Results for 6 feature-selection models

	$model1$	$model2$	$model3$	$model4$	$model5$	$model6$
Precision	0.652	0.533	0.534	0.544	0.545	0.544
Recall	0.392	0.696	0.696	0.694	0.693	0.695
F1	0.490	0.604	0.605	0.610	0.610	0.610
Accuracy	0.905	0.894	0.894	0.897	0.897	0.897
F1 for Test	0.48913	0.63699	0.63700	0.63750	0.64186	**0.64214**

As we expected, the results prove that using our chosen features can improve the overall F1 score for the test set. The most important feature is to use feature probability to only replace transition, but we have to keep the emission probability of words. So emission probability of words is a very important feature. Besides, using window of words near the given word can also improve the results to some degree. Increasing the window size of POS also works in the experiments.

A result for our Kaggle submission is as Figure 6.



Your most recent submission

Name	Submitted	Wait time	Execution time	Score
test-result-m2.csv	4 hours ago	0 seconds	0 seconds	0.64186

Complete

Jump to your position on the leaderboard

Figure 6 Kaggle Submission for model 2

3. Error analysis and comparison of HMM model with second model

Overall, the second model performs better since it provides a better F1 score than the HMM model.

For **HMM** model, since we use a lambda to lower the weight of transition probability, the observation probability plays a more important role. So, the transition matrix does not affect much as the observation matrix.

Now, we look into some examples of HMM being wrong while the second model correct.

Example 1: See look !, ['VERB', 'VERB', 'PUNCT'], [0, 0, 0]

Example 2: See him ., ['VERB', 'PRON', 'PUNCT'], [0, 0, 0]

For both of these two examples, the HMM model predicts the label sequence as [1, 0, 0]. The reason behinds this is that the observation probability plays a more important role here, so P (“See” $|t_i$) will contributes more. Since $P(\text{"See"}|\mathrm{t_i}) = \frac{\text{count("See" with tag t}_\text{i})}{\text{count(tag t}_\text{i})}$, and the count of tag 0 in training dataset is 103571, the count of tag 1 is 13051, the count of “See” with tag 0 and the count of “See” with tag 1 are extremely small (4 and 2 respectively), then $P(\text{"See"}|1) > P(\text{"See"}|0)$, in this case, HMM model will predict the label 1 for “See”, resulting in an incorrect predication. We can see that using just the count and probability as in HMM model is not a very useful way, therefore, HMM model cannot perform well sometimes.

So, in order to **improve model 1**, we can consider adding some more features, just as in model 2.

For the second model using **Logistic Regression**, we create features with POS tags and words of window size greater than 1. In this case, we designed the starting word of each sentence to get “Start” pos tag for its preceding tags and the word preceding the starting word to be “None”.

For example: That 's what, ['DET', 'VERB', 'NOUN'], [1,0,0]

With POS window size 1, the features for the word “That” becomes {word: That, pos: ‘DET, prev_pos: ‘Start’, next_pos: ‘VERB’, prev_word: ‘None’, next_word: ‘’s’}

But such combination with label 1 is very rare in training data. So, this results in many misclassifications for the label of the first word in each sentence and the model has a tendency to classify the beginning word of each sentence as label 0.

To **improve the model 2**, if costs for space are accepted, we can still increase the window size. Or we can give some special handling for the beginning words.

The major difference between the two models is the use of transition probability of labels versus feature probability. The use of feature probability yields better results as it is more related to the words and it shows the word relations with the context.

Workflow & Contribution of each group member:

At first, we discuss the whole structure of the work and think the coding work can be divided (though the implementations of Viterbi algorithm are similar). Then we keep on discussing the necessary follow-up tests that we need in this project and communicate whether or not to do some tests. We also share the codes in a private github project, so we can keep up with each other's work and pose problems if meeting confusion.
Boliang Yang (by283): implementation of model1.
Jinwei Shen(js3559): implementation of model2.
Shenghua Li(sl3293): check and test codes and structure the whole report.

Feedback:

The overall difficulty of the project is medium, we can handle it within necessary time. The instructions are not as detailed as the first project, which we think cause some confusion. And maybe having some more references directly related to the problem for us will be better.

Reference:[1] Dan Jurafsky and James H. Martin. Speech & language processing.