

CS 5740 Project 2 report

Team name: kaggle goose

Team members: Pu Zhao (pz253), Suet Cho (sc2299), Shunqi Mao (sm2784)

Overall Goal

In this project, we will be implementing a model that identifies relevant information in a text and tags it with the appropriate label in order to achieve Metaphor Detection. The goals for this project are to create prediction and classification models that can sufficiently assort the texts that contain metaphors and the ones without. A reasonable approach is to utilize Hidden Markov Model as our baseline and to implement a classification model with self-defined features space. Therefore, our tasks would be to preprocess our data in a proper way, to create different feature spaces for classification using our domain knowledge, to implement an optimal Hidden Markov Model, and to refine and improve our results using validation approach.

Dataset

We are given a training set with 6324 sentences, a validation set with 1551 sentences and a test set. The dataset consists of three pieces of information -- the original sentences, the corresponding Parts of Speech tags and a list of binary numbers indicating whether or not each token is a metaphor where 1 indicating the token is a metaphor and 0 otherwise. By inspection, we can see that most of the words are non-metaphorical. Using the datasets, we are hoping to compare the performance of a Hidden Markov Model and a classification model. We used Python for our analysis.

Model 1 Preprocessing and Implementation Details

A Markov chain is needed when we want to compute a probability for a sequence of observable events. In this case, the observable events are the sentences. However, we are interested in the hidden events in our case, or the metaphor tag of the corresponding words, which we cannot observe directly. We cannot observe the metaphor tags but instead we can infer the tags from the word sequences. In our case, we formulate the model this way: the set of states N is the metaphor tag for each sentence with length N , a transition probability matrix consists of the probabilities of moving from a hidden state to the other (e.g. metaphor to non-metaphor, non-metaphor to metaphor, metaphor to metaphor, non-metaphor to non-metaphor), a sequence of T observations with each observation represents each word in the sentence, an observation likelihood of the probability of a word being generated by a tag, and an initial probability distribution over the states which indicates the chances of a sentence starting with a metaphor or

not. To utilize this model, we must make two assumptions: the Markov Assumption, meaning that we will implement the bigram probability of the tags, and the Output Independence assumption meaning that we will assume that each emission probability is independent.

In terms of the data preprocessing and data structure, we preprocessed the data into two dictionaries. The first dictionary records the unique values as keys and the list of labels appeared in the training corpus as the value. The purpose of this dictionary is to calculate the emission probability of each tag to a corresponding word. The second dictionary records the label's bigrams as the key and their occurrence as the value. The purpose of this dictionary is to calculate the transitional probability from one state to another. Since we only have two states, the metaphor state and the non-metaphor state, there will be four transitional probabilities. In terms of the probability calculation, we compute the probabilities in the following ways. The emission probabilities $P(w_i|t_i)$ is the sum of the word with a corresponding tag divide by the total number of the tag in the training corpus. The transitional probabilities $P(t_i|t_{i-1})$ is the sum of the tag bigrams divided by the sum of the unigram of the prior tag. The preprocessing code piece is provided in **Appendix A**.

After we preprocessed the data, we implemented the Viterbi Algorithm to solve the Hidden Markov Model for this task. The Viterbi Algorithm is an instance of dynamic programming with the number of states N as the state transition and the length of the sentence T as the number of recursive calls. In terms of the data structure, we used a N by T 2D-array to record the transitional probabilities of each tag with each emission. After computing the maximum probability that this 2D-array can generate, we can use back pointer to find the sequence of the tag in the hidden states. In order to compute the maximum likelihood properly, we have to calculate the metaphor tag sequence probability in a log form. The reason to use a log space for the computing is that we want to avoid the underflow value when we compute the product of the probabilities since the products of a bunch of probabilities are very small. On the other hand, since we realize that there is an overwhelming number of non-metaphorical tag and it is very likely that we never seen the word previously in the training corpus, it is very likely for us to produce a probability of 0 in most of the recursive calls due to unknown words. To avoid generating maximum likelihood of 0, we can utilize the additive nature of the log space. Computing in log space avoids number underflow. In terms of unknown word handling, we used add-k smothering when we calculate the emission probabilities in order to avoid getting 0 probabilities when we encounter unknown word in the test corpus. The Viterbi Algorithm and prediction code piece is provided in **Appendix B**.

In terms of the experimental aspect, we add different weight lambda onto the transitional probabilities and see if we can optimize the results by comparing the results with the validation set. A lambda value between 0 and 1 indicates that the importance of the transitional probabilities is less than the emission probabilities. On the other hand, a lambda value greater than 1 and beyond indicates that the transitional probabilities are more significant than the emission probabilities. On the other hand, we also incorporated additive smoothing where we smooth the probability distribution of the emission probabilities by changing the value of k. Additive smoothing is essential to perform this task because it is very likely that we will encounter a lot of 0 probabilities. After defining the experimental parameters, we could optimize the prediction accuracies with various lambda values and k value.

Model 1 Experiments, Results, and Error Analysis

We experimented the different lambda values and k values for the Hidden Markov Model. Since the lambda value is attached to the transitional probabilities and the k value is infused in the emission probabilities, by changing their value, we can alter the performance of the Hidden Markov Model by a significant amount and can enable us to observe the behavior of the model when we change the two probabilities distributions. By changing the lambda values, we can add and subtract the weight of the transitional probabilities by assigning a positive number to it. A lambda value between 0 and 1 indicates transitional probabilities is less important than emission probabilities whereas a lambda value greater than 1 indicates transitional probabilities is more important than emission probabilities. By changing the k value, we change the probability distribution of the emission probabilities. A greater k value indicates a lower kurtosis on the distribution and vice versa. Therefore, we experimented with various lambda and k values and the results are listed below:

Lambda	K	Precision	Recall	F1	Accuracy
0.5	0.1	40.151%	79.407%	53.334%	83.858%
0.1	0.1	35.562%	89.971%	50.975%	79.898%
1	0.1	50.602%	50.546%	50.574%	88.524%
0.5	0.5	35.096%	90.193%	50.530%	79.486%

As we experimented with different lambda and k values, we observed a pattern that if we decrease or increase the lambda value from 0.5, there is a high chance of getting a lower F1 score. As shown in the chart above, when we keep the value of K as 0.1 and change the value of lambda away from 0.5, in our case 0.1 and 1, the F1 score went from 53% to around 50~51%. Similarly, as we tested with different K values, we also recognized that a lower or higher K value of 0.1 also produces a lower F1 score. As shown above, when we change the value of K from 0.1

to 0.5, the F1 score also goes down. As such, we decided that the best parameters we can use is 0.5 for lambda and 0.1 for K, producing a F1 score of 51.95% with the test set on Kaggle.

Model 2 Preprocessing and Implementation Details

Similar to the approach for model 1, we are using Viterbi algorithm as a framework for metaphor tag sequences. In contrast, we are using classification probabilities on the tag $P(t_i | \text{features})$ to replace the original emission probabilities $P(w_i | t_i)$ this time. With this approach, we are able to retain the Markovian nature of the sequence meaning that we will use the same transitional probabilities in Model 1, while modifying the emission probabilities into classification probabilities. To obtain the classification probabilities from the feature space, we decided to use Multinomial Naïve Bayes Classifier in the *Scikit Learn* library. As such, we brainstormed multiple features that we can incorporate into the feature space when we compute the classification probabilities. Since we were also given the parts of speech tagging of the sentence but couldn't incorporate them in the Hidden Markov Model, we decided to use the tags occurrence as a feature. On the other hand, we also incorporated the parts of speech tag occurrence of the previous and next word as two features. The index of the word can also be a feature to predict the metaphor tag of the word. And lastly, we also used the emission probabilities from the Hidden Markov Model as one of the features as it clearly captures important information in terms of metaphor tagging as we can see from the Hidden Markov Model prediction.

In terms of data preprocessing, since we want to calculate the occurrence of the POS tag in comparison with the metaphor tag, we compute the value in the following way: POS tag occurrence of the current word = number of the POS tag given a metaphor tag/ total number of the metaphor tag in the training data. This is very similar to the bigram probabilities. In verbal description, the POS tag occurrence feature is the same as the emission probability of the POS tag given a metaphor tag. We calculate the POS tag occurrence for the previous word and the next word in a similar manner. In term of the index of the word, we use the word position in the sentence as the value. Lastly, for the emission probabilities, we can reuse the same calculation as Model 1 when we perform the Hidden Markov Model computation.

Model 2 Experiments, Results, and Error Analysis

We used with different subsets of the features that we defined to perform the Multinomial Naïve Bayes probabilities calculation. As we discussed above, we defined 5 different features and we used their combinations as the feature space for the models. We defined the feature space of the four models to be the following:

	Feature(s)
Model 1	POS tag occurrence of the current word
Model 2	POS tag occurrence of the current word, POS tag occurrence of the previous word, POS tag occurrence of the next word
Model 3	POS tag occurrence of the current word, POS tag occurrence of the previous word, POS tag occurrence of the next word, index of the current word
Model 4	POS tag occurrence of the current word, POS tag occurrence of the previous word, POS tag occurrence of the next word, index of the current word, emission probability of the current word given the tag is a metaphor, emission probability of the current word given the tag is not a metaphor

We defined the predicted label to be whether the word is a metaphor or not, 1 indicates that the word is a metaphor and 0 otherwise. Similar to Model 1 implementation, we first preprocess the data into unigram probabilities, bigram probabilities, a dictionary for the word and a dictionary for the POS. Using the numbers from the dictionary above, we can compute the POS tag occurrence probabilities of the words as a feature. Then, we created a 2D-array to record the list of features for every word that appears in the training set. We performed the computation of the feature matrix with a nested for loop as indicated in the code piece. The preprocessing and feature generation code piece is provided in **Appendix C**.

After that, we feed the classification probabilities into the Viterbi model. We replicated our Viterbi algorithm code from Model 1 and replace the emission probabilities with the Naïve Bayes classification probabilities. This function also returns a metaphor tag sequence. The new Viterbi algorithm code piece is provided in **Appendix D**.

We experimented with various models and the results are listed below:

Model	Precision	Recall	F1	Accuracy
1	37.669%	84.934%	50.715%	81.925%
2	41.726%	72.647%	51.482%	84.956%
3	41.678%	69.489%	51.105%	85.161%
4	40.698%	76.599%	53.155%	84.317%

As we experimented with different subset of features in Model 2, our results are very similar to those we gathered from Model 1. However, there is a trend of increasing F1 as we incorporate more features into the feature spaces of the models. This corresponds to our assumption that a more refined and sophisticated features is more likely to capture more information and has a better prediction on the data. The models produces 51~52% F1 score on the test data.

Models Comparison

As shown in the list of results in model 1 and model 2 are very similar to each other as our model 1 performed slightly better than our model 2. The four different models are different subsets of the features that we predefined. As we discussed above, the model with all the features seem to perform the best out of the different Viterbi Naïve Bayes classification models. This is linked to our assumption that the more features we incorporated, the more information that the classifier can capture and thus has a better prediction result. The transition matrix plays a big role in the two model as it contributes as half of the entire probabilistic calculation. The lambda values that we put on the transitional probability matrix can help us to scale the importance of the transitional probabilities in comparison to the emission probabilities and Naïve Bayes probabilities. In our Hidden Markov Model, we only incorporated the metaphor tag and completely ignore the parts of speech tagging of the corpus while in model 2 we incorporated both. It shows that the parts of speech tags of the sentences play a role in the sequence prediction but not as important as computing the metaphor tag sequence directly with Hidden Markov Model. Overall, we agree that Hidden Markov Model is sufficient in performing this metaphor tagging task.

Individual Member Contribution

We implemented the program jointly and sought for help from each other mostly on debugging. Pu put an emphasis on implementing the Viterbi algorithm implementation and preprocessing the data for Model 1. Shunqi worked on preprocessing the data into usable features for Model 2 and implementing the Naive Bayes classifier. Suet emphasized on analyzing the results from each part and recording the works in text and report.

Appendix A. Model 1 preprocessing

```
def read_data(file):
    with open(file) as f:
        lines = csv.reader(f)
        next(lines)
        label_seq = []
        sentences = []
        for line in lines:
            label = ast.literal_eval(line[2])
            words = line[0].split()
            label_seq.append(label)
            sentences.append(words)
        return sentences, label_seq

def preprocess(sentences, label_seq):
    tag_unigram = {}
    for label in label_seq:
        for i in range(len(label)):
            if label[i] not in tag_unigram:
                tag_unigram[label[i]] = 1
            else:
                tag_unigram[label[i]] += 1

    tag_bigram = {}
    for label in label_seq:
        for i in range(len(label) - 1):
            tag1 = label[i]
            tag2 = label[i + 1]
            if (tag1, tag2) not in tag_bigram:
                tag_bigram[(tag1, tag2)] = 1
            else:
                tag_bigram[(tag1, tag2)] += 1

    word_dict = {}
    for i in range(len(sentences)):
        words = sentences[i]
        labels = label_seq[i]
        for j in range(len(words)):
            if words[j] not in word_dict:
                word_dict[words[j]] = [labels[j]]
            else:
                word_dict[words[j]].append(labels[j])

    tag0 = 0
    tag1 = 0
    for i in range(len(label_seq)):
        for label in label_seq:
            if label == 0:
                tag0 += 1
            elif label == 1:
                tag1 += 1
```

Appendix B. Model 1 Viterbi Algorithm and Prediction

```
def viterbi_algo(tag_unigram, tag_bigram, word_dict, lambda, k, test):
    score = np.zeros((2, len(test)))
    bptr = np.zeros((2, len(test)))

    if (test[0], 0) not in word_dict:
        word_dict[(test[0], 0)] = k
    if (test[0], 1) not in word_dict:
        word_dict[(test[0], 1)] = k

    score[0][0] = math.exp(lambda * math.log((tag_unigram[0] / (tag_unigram[0] + tag_unigram[1])))) + math.log((word_dict[(test[0], 0)] / tag_unigram[0])))
    score[1][0] = math.exp(lambda * math.log((tag_unigram[1] / (tag_unigram[0] + tag_unigram[1])))) + math.log((word_dict[(test[0], 1)] / tag_unigram[1])))
    bptr[0][0] = 0
    bptr[1][0] = 0

    for t in range(1, len(test)):
        if (test[t], 0) not in word_dict:
            word_dict[(test[t], 0)] = k
        if (test[t], 1) not in word_dict:
            word_dict[(test[t], 1)] = k

        score[0][t] = max(score[0][t - 1] * math.exp(lambda * math.log((tag_bigram[(0, 0)] / tag_unigram[0]))) + math.log((word_dict[(test[t], 0)] / tag_unigram[0])),
            score[1][t - 1] * math.exp(lambda * math.log((tag_bigram[(1, 0)] / tag_unigram[1]))) + math.log((word_dict[(test[t], 0)] / tag_unigram[0]))))
        bptr[0][t] = np.argmax((score[0][t - 1] * math.exp(lambda * math.log((tag_bigram[(0, 0)] / tag_unigram[0]))) + math.log((word_dict[(test[t], 0)] / tag_unigram[0]))),
            score[1][t - 1] * math.exp(lambda * math.log((tag_bigram[(1, 0)] / tag_unigram[1]))) + math.log((word_dict[(test[t], 0)] / tag_unigram[0]))))

        score[1][t] = max(score[0][t - 1] * math.exp(lambda * math.log((tag_bigram[(0, 1)] / tag_unigram[0]))) + math.log((word_dict[(test[t], 1)] / tag_unigram[1])),
            score[1][t - 1] * math.exp(lambda * math.log((tag_bigram[(1, 1)] / tag_unigram[1]))) + math.log((word_dict[(test[t], 1)] / tag_unigram[1]))))
        bptr[1][t] = np.argmax((score[0][t - 1] * math.exp(lambda * math.log((tag_bigram[(0, 1)] / tag_unigram[0]))) + math.log((word_dict[(test[t], 1)] / tag_unigram[1]))),
            score[1][t - 1] * math.exp(lambda * math.log((tag_bigram[(1, 1)] / tag_unigram[1]))) + math.log((word_dict[(test[t], 1)] / tag_unigram[1]))))

    T = np.zeros(len(test))
    T[len(test) - 1] = np.argmax((score[0][len(test) - 1], score[1][len(test) - 1]))
    for i in range(len(test) - 2, -1, -1):
        T[i] = bptr[int(T[i + 1])][i + 1]

    return T

def predict(tag_unigram, tag_bigram, word_dict, lambda, k, test):
    with open("pred.csv", "w", newline='') as f:
        f_writer = csv.writer(f, delimiter=",")
        f_writer.writerow(["idx", "label"])

        count = 1
        for sentence in test:
            labels = viterbi_algo(tag_unigram, tag_bigram, word_dict, lambda, k, sentence)
            for i in range(len(labels)):
                f_writer.writerow([count, int(labels[i])])
            count += 1

train_sen, train_label = read_data("train.csv")
tag_unigram, tag_bigram, word_dict = preprocess(train_sen, train_label)
test = read_test("val.csv")
predict(tag_unigram, tag_bigram, word_dict, 0.5, 0.1, test)
```


Appendix C.

```
def preprocess(sentences, label_seq, POS_tags):
    tag_unigram = {}
    for label in label_seq:
        for i in range(len(label)):
            if label[i] not in tag_unigram:
                tag_unigram[label[i]] = 1
            else:
                tag_unigram[label[i]] += 1

    tag_bigram = {}
    for label in label_seq:
        for i in range(len(label) - 1):
            tag1 = label[i]
            tag2 = label[i + 1]
            if (tag1, tag2) not in tag_bigram:
                tag_bigram[(tag1, tag2)] = 1
            else:
                tag_bigram[(tag1, tag2)] += 1

    word_dict = {}
    pos_dict = {}
    for i in range(len(sentences)):
        words = sentences[i]
        labels = label_seq[i]
        pos = POS_tags[i]
        for j in range(len(words)):
            if (words[j], labels[j]) not in word_dict:
                word_dict[(words[j], labels[j])] = 1
            else:
                word_dict[(words[j], labels[j])] += 1
            if (pos[j], 1) not in pos_dict:
                pos_dict[(pos[j], 1)] = 1
            else:
                pos_dict[(pos[j], 1)] += 1

    feature = []
    output = []
    for i in range(len(sentences)):
        for j in range(len(sentences[i])):
            if (POS_tags[i][j], 1) not in pos_dict:
                pos_dict[(POS_tags[i][j], 1)] = 0
            if j != (len(sentences[i]) - 1) and (POS_tags[i][j + 1], 1) not in pos_dict:
                pos_dict[(POS_tags[i][j + 1], 1)] = 0
            vector = [j] # index of current word
            vector.append(pos_dict[(POS_tags[i][j], 1)] / tag_unigram[1]) # emission probability for POS tags
            if j != 0 and j != (len(sentences[i]) - 1):
                vector.append(pos_dict[(POS_tags[i][j - 1], 1)] / tag_unigram[1]) # emission probability for previous word's POS tag
                vector.append(pos_dict[(POS_tags[i][j + 1], 1)] / tag_unigram[1]) # emission probability for next word's POS tag
            else:
                vector.append(0)
                vector.append(0)
            if (sentences[i][j], 1) not in word_dict:
                word_dict[(sentences[i][j], 1)] = 0
            vector.append(word_dict[(sentences[i][j], 1)] / tag_unigram[1]) # emission probability for current word
            feature.append(vector)
            output.append(label_seq[i][j])

    return tag_unigram, tag_bigram, word_dict, pos_dict, feature, output
```

Appendix D.

```
def viterbi_algo(tag_unigram, tag_bigram, word_dict, pos_dict, feature, output, lmbda, k, test_words, test_pos):
    score = np.zeros((2, len(test_words)))
    bptr = np.zeros((2, len(test_words)))

    clf = MultinomialNB()
    clf.fit(feature, output)

    X = []
    for i in range(len(test_words)):
        if (test_pos[i], 1) not in pos_dict:
            pos_dict[(test_pos[i], 1)] = k
        if i != (len(test_words) - 1) and (test_pos[i + 1], 1) not in pos_dict:
            pos_dict[(test_pos[i + 1], 1)] = k
        vector = [i]
        vector.append(pos_dict[(test_pos[i], 1)] / tag_unigram[1])
        if i != 0 and i != (len(test_words) - 1):
            vector.append(pos_dict[(test_pos[i - 1], 1)] / tag_unigram[1])
            vector.append(pos_dict[(test_pos[i + 1], 1)] / tag_unigram[1])
        else:
            vector.append(0)
            vector.append(0)
        if (test_words[i], 1) not in word_dict:
            word_dict[(test_words[i], 1)] = k
        vector.append(word_dict[(test_words[i], 1)] / tag_unigram[1])
        X.append(vector)

    probs = clf.predict_proba(X)

    if (test_words[0], 0) not in word_dict:
        word_dict[(test_words[0], 0)] = k
    if (test_words[0], 1) not in word_dict:
        word_dict[(test_words[0], 1)] = k

    score[0][0] = math.exp(lmbda * math.log((tag_unigram[0] / (tag_unigram[0] + tag_unigram[1])))) + math.log(probs[0][0]))
    score[1][0] = math.exp(lmbda * math.log((tag_unigram[1] / (tag_unigram[0] + tag_unigram[1])))) + math.log(probs[0][1]))
    bptr[0][0] = 0
    bptr[1][0] = 0

    for t in range(1, len(test_words)):
        if (test_words[t], 0) not in word_dict:
            word_dict[(test_words[t], 0)] = k
        if (test_words[t], 1) not in word_dict:
            word_dict[(test_words[t], 1)] = k
































        score[0][t] = max(score[0][t - 1] * math.exp(lmbda * math.log((tag_bigram[(0, 0)] / tag_unigram[0])) + math.log(probs[t][0])),
                          score[1][t - 1] * math.exp(lmbda * math.log((tag_bigram[(1, 0)] / tag_unigram[1])) + math.log(probs[t][0]))))
        bptr[0][t] = np.argmax((score[0][t - 1] * math.exp(lmbda * math.log((tag_bigram[(0, 0)] / tag_unigram[0])) + math.log(probs[t][0])),
                              score[1][t - 1] * math.exp(lmbda * math.log((tag_bigram[(1, 0)] / tag_unigram[1])) + math.log(probs[t][0]))))

        score[1][t] = max(score[0][t - 1] * math.exp(lmbda * math.log((tag_bigram[(0, 1)] / tag_unigram[0])) + math.log(probs[t][1])),
                          score[1][t - 1] * math.exp(lmbda * math.log((tag_bigram[(1, 1)] / tag_unigram[1])) + math.log(probs[t][1]))))
        bptr[1][t] = np.argmax((score[0][t - 1] * math.exp(lmbda * math.log((tag_bigram[(0, 1)] / tag_unigram[0])) + math.log(probs[t][1])),
                              score[1][t - 1] * math.exp(lmbda * math.log((tag_bigram[(1, 1)] / tag_unigram[1])) + math.log(probs[t][1]))))

    T = np.zeros(len(test_words))
    T[len(test_words) - 1] = np.argmax((score[0][len(test_words) - 1], score[1][len(test_words) - 1]))
    for i in range(len(test_words) - 2, -1, -1):
        T[i] = bptr[int(T[i + 1])][i + 1]

    return T
```

Kaggle Submission Competition.

9	MusicNerds	  	0.57678	6	2d
10	Fantastic Three	  	0.56688	5	2d
11	F1 Champions	  	0.55912	6	3d
12	Naive Language Processor	  	0.55529	6	7h
13	TODO	 	0.52606	2	5h
14	kaggle goose	  	0.52002	3	5m
Your Best Entry 					
Your submission scored 0.51523, which is not an improvement of your best score. Keep trying!					
15	BLACKPINK Jisoo	  	0.52002	5	3d
16	Random Is King	 	0.51413	7	3h
17	NIT_grams	  	0.51295	1	3d
18	Team kaBao	 	0.51102	8	4h
19	Team LLZ	  	0.51040	6	31m