

CS 4740/5740 Fall 2019 Project 2
Metaphor Detection with Sequence Labelling Models
Natalie Karlsson (nk435), Larissa Pereira (lp445), Kaveesha Shah (ks2379)
Kaggle group name – Team NLK

1. Sequence Labeling Models

(a) Implementation Details:

Language used - Python

Data structure - dictionary for transition and lexical generation probabilities

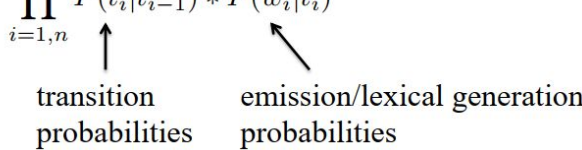
Libraries - sklearn, string, numpy, csv, nltk

1. HMMs

Our primary purpose is to map a sequence of words with an observed sequence of labels. In this project we label a word sequence with whether or not it is a metaphor. We use Hidden Markov Model to predict the sequence of labels (metaphor/not metaphor).

We used the below formula for HMM assuming a bigram model, to determine the tag sequence.

$$P(t_1, \dots, t_n) * P(w_1, \dots, w_n | t_1, \dots, t_n) \approx \prod_{i=1, n} P(t_i | t_{i-1}) * P(w_i | t_i)$$



The key components used to calculate Score for HMM are as follows:

1. **Transition Probability** is the probability of moving from one state to another. We calculate these for the below states –
 1. Metaphor (1) → Non Metaphor (0)
 2. Non Metaphor (0) → Metaphor (1)
 3. Non Metaphor (0) → Non Metaphor (0)
 4. Metaphor (1) → Metaphor (1)

Formula used to calculate the bigram transition probabilities for the above transitions:

$$P(t_i | t_{i-1}) = \frac{C(t_{i-1}, t_i)}{C(t_{i-1})}$$

$$P(0|1) = \frac{Count(10)}{Count(1)}$$

2. **Lexical/Emission generation probability** is the probability of a label being generated from a particular state. We calculate the lexical probabilities for a word given it is a metaphor or not

Formula used to calculate the lexical generation probabilities :

$$P(w_i|t_i) = \frac{C(t_i, w_i)}{C(t_i)}$$

$$P(\text{again}|0) = \frac{\text{Count}(\text{again},0)}{\text{Count}(0)}$$

For training the model -

1. We calculated the counts for the combination (word,label) from the training corpus by zipping the word and its associated label into a dictionary and then gathered counts for the combination in the dictionary (lexical_prob)

```
dict_iteration=dict(zip(s, m))
for i in dict_iteration.items():
    if i in lexical_prob:
        lexical_prob[i]+=1
    else:
        lexical_prob[i]=1
```

2. We calculated the counts for the label transitions by creating a tuple of the (current_label,next_label) and then gathered counts for these tuples from the training corpus in the dictionary (trans_prob)

```
tup=(m[i],m[i+1])
if tup in trans_prob:
    trans_prob[tup]+=1
else:
    trans_prob[tup]=1
```

Unknown word handling and Smoothing-

When we encounter a word while testing that was not observed while training we assign it a zero probability. To avoid data sparsity (i.e. the overall probability becoming 0) we employ add -k smoothing where the value of k is determined by experimenting with different values on the validation set.

```
if (sentence[word],label) not in lexical_prob:
    lexical_prob[(sentence[word],label)]=0
Pwi=(lexical_prob[(sentence[word],label)]+k)/(count[label]+k*len_p)
```

Implementing Viterbi -

1. Initialization - Here we calculate the score and backpointer with respect to the first word in the sequence

```

#Initialization
for i in range(2):
    #if a word, label pair not found, assign a 0 count to it. it will be handled by smoothing
    if (sentence[0],i) not in lexical_prob:
        lexical_prob[sentence[0],i]=0
    score[i][0]=(start_0/(start_0+start_1)) * (((lexical_prob[sentence[0],i])+k)/((count[i]+k*len_p))
    bptr[i][0] = -1

```

2. Iteration- Here we calculate the score and backpointer for the remaining words

```

#Iteration
for word in range(1,n):
    for label in range(2):
        scores_for_next_label=[]
        for j in range(2):
            Pti=trans_prob[(j,label)]/count[j]
            sc=score[j][word-1]*Pti
            scores_for_next_label.append(sc)

        if (sentence[word],label) not in lexical_prob:
            lexical_prob[(sentence[word],label)]=0
        Pwi=(lexical_prob[(sentence[word],label)]+k)/((count[label]+k*len_p))

        score[label][word]=max(scores_for_next_label)*Pwi
        bptr[label][word]=round(scores_for_next_label.index(max(scores_for_next_label)))

```

3. Backtracking - We then finally backtrack on the back pointer matrix to obtain the final predicted metaphor tag sequence

```

#Backtracking
t=[-1]*n

t[n-1] = np.where(score == (max(score[i][n-1] for i in range(2))))[0][0]
for i in range(n-2,-1,-1):
    t[i]=bptr[t[i+1]][i+1]

```

2. MEMM

We implemented a Maximum Entropy Markov Model for model 2, using the MaxEnt classifier of the nltk library. To train the model, we first called the `maxent.TypedMaxentFeatureEncoding.train()` and then `maxent.MaxentClassifier.train()` with this encoding passed in as the 'encoding' parameter. For the encoding, we passed in a list of (feature dictionary, tag) tuples. To get these tuples we iterated over the training data, getting the features of each word and passing those to a function called `get_train_tuples()`.

```

def get_train_tuple(word,pos,tag,length,word_position,prev_pos,next_pos):
    feature_dict = {"POS":pos, "word":word, "length":length, "position":word_position, "prev_pos":prev_pos, "next_pos":next_pos}
    return (feature_dict,tag)

```

```

#encoded the maxEnt classifier using the tuples created in getTrain function
#Use the encoded part to train the MaxentClassifier
train = getTrain()
def train_maxEnt():
    global train
    encoding = maxent.TypedMaxentFeatureEncoding.train(
        train, count_cutoff=3, alwayson_features=True)
    classifier = maxent.MaxentClassifier.train(
        train, bernoulli=False, encoding=encoding, trace=0,max_iter=15)
    return classifier
classifier = train_maxEnt()

```

Training is where we implemented feature engineering. At first the features we used were the word's part of speech and the word itself. We then added word length and position. Finally, we added features for the previous word's part of speech and the next word's part of speech. For the first word's previous part of speech we used the string "start" and for the last word's next part of speech we used the string "end". The intuition for using the respective features is mentioned in the Experiments section below.

Following training, we called `classify_many(test)` on the classifier returned by the `train()` function.

```
classifier.classify_many(test)
disb = classifier.prob_classify_many(test)
```

The input to this, `test`, is a feature dictionary of the test data. `classify_many()` returned a list of probability distributions which we used in the Viterbi algorithm to tag each word in the test set. Previously, our calculation for each entry in the SCORE matrix included:

$P(t_i | t_{i-1})P(w_i | t_i)$, this represents transition probability * lexical probability. With the MaxEnt classifier, we get $P(t_j | w_i)$ for each word w_i in the vocabulary and each tag t_j in the tagset. We experimented with replacing both the lexical and transition probabilities with this probability. To do this, we used the same Viterbi algorithm function as for the HMM, but we replaced the lines where we calculated the probabilities using the counts we had calculated. When we first replaced lexical probability (P_{wi} in the code), the returned array contained all 0s. This gave us pretty good accuracy (~88%), but an F-Score of 0 since both precision and recall are 0 in this case. So we instead replaced the transition probability with $P(t_j | w_i)$.

```
for j in range(2):
    Pti=disb[word].prob(label)
    sc=score[j][word-1]*Pti
    scores_for_next_label.append(sc)
```

In the HMM, the transition probability is the probability of a tag given the previous tag. In the MEMM, we replaced this with the probability of a tag given a set of features. It makes sense to switch these two probabilities because they are essentially calculating the same thing: the probability of a tag given some features. In MEMM we just used more features, and in HMM the only feature was the tag of the previous word.

When testing, we used the same method of unknown word handling as in the HMM model: add-k smoothing. We set k to .00001.

See the references section for our references for MEMM. We used the Jurafsky and Martin textbook for an explanation of MEMM, site 1 for a tutorial on MaxEnt in nltk, and site 2 for the documentation of MaxEnt on nltk.

(b) Pre-processing:

In order to eliminate redundancy and filter irrelevant details that don't add value .

- Lower case - we convert all characters to lowercase using the `lower()` method of string data type
- Split – we split the sentence into word tokens using the `string.split()` method

- Replace – we also remove the unwanted punctuations using the string.replace() method

c) **Experiments:**

We used the following features for the MaxEnt and our hypothesis for choosing these features is mentioned below

1. The word itself is an important feature that the classifier should learn from because there could be a lot of words in the training dataset that would be tagged as metaphors multiple times and can be used to easily classify if observed in the test set
2. Part of Speech is one of the critical features we considered as it could be a strong factor to help determine/identify metaphors. For example, Metaphors could generally be Nouns or Verbs but it is very rare for a metaphor to be a Determiner.
3. Length of the word - The length of the word could also help identify words that cannot be metaphors. For example - short words like- 'I' 'am' 'the' 'of' are not metaphors.
4. Position of the word - In most common cases metaphors occur in the middle of the sentence. In very rare cases will a sentence end or begin with a metaphor. Hence considering the position of the word will help gain more insights.
5. Previous and Next Part of Speech tags - Here we consider a window based classification approach considering the previous and next part of speech will also help to determine the tag of the current word. For example a metaphor tagged as a noun could be surrounded with adjectives or determiners.

(d) **Results:**

We evaluated the 2 models based on its Accuracy, F-score, Precision and Recall values obtained by running the models on the Validation set. Accuracy is not a great measure when it comes to assessing the performance of these models because of class imbalance. Hence we primarily consider the F-score, Precision and Recall values to choose the best outcome.

Model 1

Experimenting with different hyperparameters(k) that gives highest accuracy on the validation set

k values	0.3	0.5	1	0.05	0.1	0.2	0.00001	0.0000001
Accuracy	89.34%	89.67%	89.32%	0.887	89.09%	89.25%	88.63%	88.63%
F-score	40.86	37.81	26.51	48.15	47.45	44.02	49.38	49.37
Precision	57.52	62.88	66.13	51.56	53.89	55.75	51.13	51.13
Recall	31.69	27.03	16.58	45.17	42.38	36.37	47.76	47.73

Model 2

1) Features used - (Word, Part of Speech)

accuracy	87.87	87.83	87.76
f-score	0.33	0.313	0.276
k	0.1	0.2	0.5

2) Features used - (Word,Part of Speech,word length, word position)

accuracy	87.11	87.34	88.05
f-score	0.419	0.4	0.37
k	0.1	0.2	0.5

3) Features used - (Word,Part of Speech,word length, word position, prev_pos,next_pos)

Accuracy	88.15	88.37	87.81	87.69	87.69
f-score	39.54	37.28	40.68	41.69	41.66
Precision	48.57	49.9	46.82	46.34	46.35
Recall	33.34	29.75	35.9	37.88	37.84
k	0.1	0.2	0.05	0.00001	0.0000001

After this experimentation we learnt that the length and the position of the words gave good classification results. Further, adding the previous and next Part of Speech to the feature set helped us obtain a better F-score on the test data. The F-score increased from **0.31**(when using word and POS tag) **to 0.43**(after adding features like previous and next part of speech) for the test data(as per Kaggle).

e) **Comparing the models**

On the Validation set-

Our HMM model performed the best, with an F-score of .4938. Following that, the MEMM with the features (word, part of speech, word length, and word position) performed the best with an F-score of .419. With the POS of previous and next word added as features, we got an F-score of .4169. This shows that the most useful features were word length and word position in the sentence.

Training time for the MEMM is proportional to the number of features used. We observed that the time to train increases by addition of more features (~15-17 mins for the current implementation).

On the real Test Data(Kaggle)-

The observations were that our model with all the mentioned features included gave best results F-score of 0.437 as opposed to the initial model with only the word and its Part of Speech gave F-score of 0.319. Hence our Kaggle score was highest with model 2 including all features (word, part of speech tag, position of the word, length of the word, previous POS tag, next POS tag)

f) **Kaggle competition**

Kaggle group name – **Team NLK**

Your most recent submission				
Name	Submitted	Wait time	Execution time	Score
HMM2.csv	2 days ago	0 seconds	0 seconds	0.43734
Complete				
Jump to your position on the leaderboard ▾				

Individual Member Contribution

Work was split evenly between the three members of the group, with most of the work being done as pair (trio) programming, with all of us working together on the same code. Contributors for individual sections are broken down as follows:

Training model 1 and Viterbi - We implemented training for model 1 and the Viterbi algorithm (same for both models) with pair programming with the three of us all working together.

Training model 2 - Natalie researched MaxEnt and implemented the framework for the MaxEnt classifier and we worked all together to debug the code.

Feature extraction model 2 - Kaveesha and Larissa researched, implemented, and experimented with different features for model 2.

Report - Joint collaboration between all three members.

Feedback - The project gave us a much better insight into HMM. A little more detail about the nitigrities would have been appreciated.

References:

1. <http://www.nltk.org/howto/classify.html>
2. <https://www.nltk.org/api/nltk.classify.html#nltk.classify.maxent.MaxentClassifier>