A2 CSE 156 Report

David Dai
A13665770
Apr. 28/2019

Section 1. Supervised Learning

1. Baseline Performance

The baseline model is the logistic regression classifier from the python sklearn. Linear_model library. The classifier selects the seed of the pseudo-random number generator to zero, and it sets the maximum number of iterations to converge to 10000. The baseline logistic regression classifier uses the 'lbfgs' solver which is suitable for multiclass problems. The following table is the result of the baseline model.

Baseline	Train	Dev	Test
Performance	0.982104	0.777293	0.77706

2. Improvements / Changes

Before I go into the details of the improvements, I actually tried to manipulate with different kinds of tokenizer and stemmer provided the nltk. Improvement b's choices are the best I found throughout my approaches. I also tried to manipulate different stop_word or language models, but stop_word is harder to implement because of linguistic edge cases. And any language model beyond bigram tends overfits. More descriptions below.

- **Improvement a. Use of TF-IDF Vectorizer**

The baseline model uses the CountVectorizer which introduces a strong bias towards words which appeared multiple times in the training dataset. CountVectorizer can be interpreted as counting the number of appearances of each word; thus words with higher appearances will be more important than the words with lower appearances. On the other hand, the TF-IDF Vectorizer first uses term frequency, which counts the number of appearances over a certain length of the text, then TF-IDF Vectorizer uses inverse document frequency to make the weight of the words more proportional and appropriate. The formula for the term frequency and the inverse document frequency formula are here below

$$TF(word,\ sequence) = \frac{number\ of\ word\ appearances\ in\ this\ sequnce}{number\ of\ words\ in\ this\ sequence}$$

$$IDF(word) = log\{\frac{N = number\ of\ sequences}{|d \in D : t \in d| = number\ of\ sequence\ of\ words\ where\ the\ word\ appeared}\}$$

- **Improvement b. Change of Tokenizer**

The CountVectorizer and the TF-IDF Vectorizer all have pre-built-in tokenizers. To improve the performance of the classifier, I imported a new tokenizer and a new stemmer from the natural language toolkit (nltk). RegexpTokenizer is used to split the string into regular expressions and I chose to use the default parameter. Then after having all the words in tokens, I need to reduce the number of words with a stemmer. A stemmer is used to remove all the prefixes and affixes of the word and leaving the stem of the word for the machine to process. The SnowballStemmer parameter is set to "English" to allow the pre-built in stemmer to process English. The implementation of the tokenize method will pass in a sentence and the tokenizer along with the stemmer will produce a word; then this word will be checked to see if it is in the middle_word array. The goal is to remove all the words that are ambiguous among classifying to positive or negative reviews.

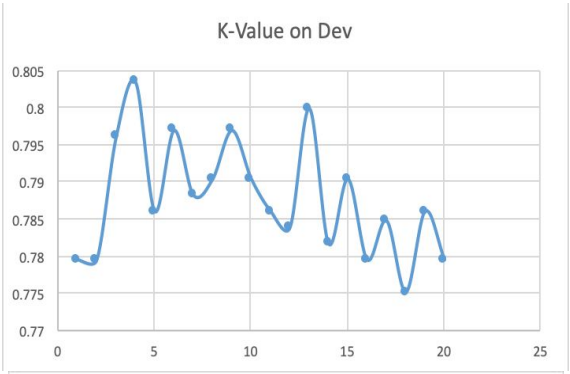
- **Improvement c. Removing Ambiguous Words**

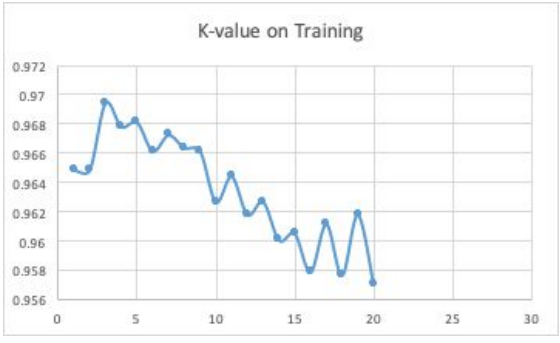
In my implementation, there are several occasions of a variable called middel_words. I used them to record the words that are rather ambiguous towards the meaning of positive or negative reviews. To train the machine to classify the words whether is ambiguous or not, I decided to run the logistic regression twice. The first time the middle_words will contain the ones like pronouns and prepositions which do not have any sentimental meaning. Then the second time, I will pick a number k = 4 to pick the first quadrant of words to be positive reviews, the last quadrant of words to be negative reviews and the middle 50-percent of the words are counted to be middle_words.

3. Hyperparameters Tuning

- **Tuning C-value**

One of the fundamental idea to train the logistic regression is to alternate other parameters on the logistic regression function. One of the common practices is tuning the C-value which is the inverse of the regularization strength. I picked C value from [0.01, 0.05, 0.1, 0.15, 0.5, 1, 5, 10, 100, 200, 300, 400, 500] and used the GridSearch to pick the best estimator and set that to be the classifier. The rest of the parameters, I stick with the same as the basic model of the logistic regression.





- **Tuning k-value**

As I have described above, the k-value is a variable that I set to select the range of the middle ambiguous words after the first logistic regression. The k-value should not be too big that it includes too many numbers of words. If the k-value is selected to be too big, many words that have already been trained will be ignored

by the classifier. Thus, I pick k range from 1 to 25 and the result is pretty clear that k = 4 will generate the best result. The graph above is the demonstration of the tuning of k-value.

- **Tuning middle_words**
  The middle words also have to be selected partially manually, partially by the algorithm based on my implementation. In the end, middle_words is set to be ['and', 'a', 'the', 'am', 'it', 'me', 'with','in','on', 'by', 'near', 'this', 'that', 'an', 'there', 'here', 'those'].

- **Tuning n-gram**
  Different language model will generate a different result. Tuning the n-gram parameter in the TF-IDF Vectorizer will generate different results. I set the parameter to be 1, 2 and 3. This means that three different kinds of language models, unigram model, bigram model, and trigram model will be tested. After tuning, I found out that the unigram model performs poorly because it is taken each word singularly and did not improve the accuracy at all. The trigram model overfits the dataset. Thus, the bigram model performs the best.

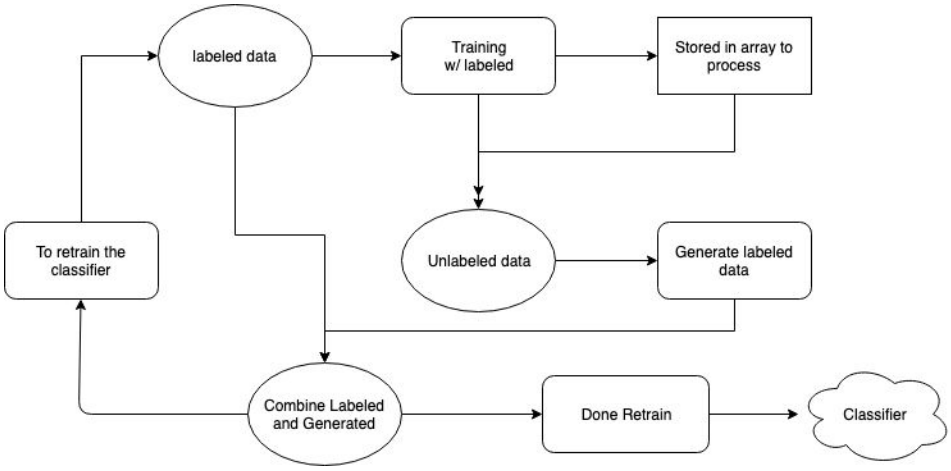
4. Result

With my implementation of the supervised classifier, the result of the accuracy improves both the test dataset, dev dataset, and the training dataset. The table below is my result.

David's implementation	Train	Dev	Test
Performance	1.0	0.803493	0.80037

Section 2. Semi-supervised Learning

1. Methodology



The idea of semi-supervised learning is described by the flowchart above. First, I used the labeled data to train the classifier and stored the results into a separate array. Then, I used this separate

array along with the unlabeled data to create a new generated labeled dataset. This step is the so-called expanding of the data. Then, I am going to throw the newly expanded data back to the classifier to let it be retrained. The algorithm will end once the newly expanded data length is smaller than the preset value.

2. Hyperparameters Tuning

With the tuning confident-value and tuning of the percent of unlabeled data, I was expecting that the tuning of the data length restriction will change the result.

- Tuning confident-value

In my implementation, the confident-value is a threshold. After the unlabeled data is being predicted with the classifier, it is used to decide whether the predicted value will be added to the extended dataset. My tested confident-values are 40%, 60%, 70%, 80% , and 90%. The following table is the result of different confident-value regarding training and dev data. I hypothesize that the accuracy of the training dataset increases but the dev dataset decreases as the confident-value increase is caused by overfitting. As the table has shown, the best paramenter to set is 40%.

Tuning	40%	60%	70%	80%
train	0.997128	0.998412	0.99888	0.99889
dev	0.790393	0.786026	0.783843	0.781052

- Tuning percent of unlabeled data

The percentage of unlabeled data being used in the dataset is really important for semi-supervised learning. My tested percentage includes 30%, 60%, 90% and 100%. All testing and tuning are under the same scenario where the confident value is set at 40%. The graph below showed that 100% provides the best result as the accuracy of both the training data and the dev data accuracies increase.

Tuning	30%	60%	90%	100%
train	0.983567	0.986052	0.99033	0.996809
dev	0.788543	0.78987	0.79103	0.791612

3. Result

First of all, the semi-supervised learning algorithm contains the same exact method from the supervised method. Ideally, the semi-supervised learning should perform better than the supervised one. In this case, however, there isn't much improvement compared to the supervised learning algorithm. As I print out the top 10 most useful features on the good sentiments, the words are:

"awesome, the best, excellent, and, delicious, best, love, amazing, !, great"

And the top 10 most useful features on the bad sentiments, the words are:

"not, horrible, worst, rude, terrible, the worst, bad, 't, no"

As you can see, "!" and "and" are not necessarily good sentiments and " 't" and "not" should not be bad sentiments hundred percent. One potential reason is that it is training the data multiple times and double or triple the errors that it might make in the supervised classifier.

David's implementation	Train	Dev	Test
Supervised	1.0	0.803493	0.80037
Semi-Supervised	0.99712817	0.790393	0.79520

Section 3. Kaggle Information

Username: Dwei
Display name: DweiD
Email: d4dai@ucsd.edu