Independent Project 1: Text Classification

CS5001-005 Natural Language Processing

Project Report

The central goal of the project is to predict the classification for Amazon Reviews. I tried Perceptron and Logistic Regression algorithms as follows,

Reading the Input data:

Each of the datasets (train/dev/test) were read-in using readlines() function and were stored as an array. Basic clean like striping unexpected characters, removing white spaces and other special characters from the data. The data was also converted into lower case to have consistent formatting

Perceptron Method:

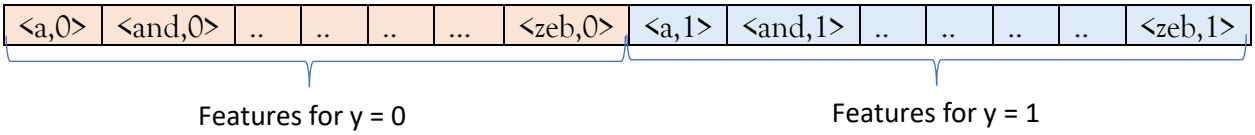
Creating Vocabulary (from scratch):

The vocabulary had to be created manually without using any in-build function. First the words were tokenized and a dictionary *'dict_word_counts'* was created with keys as the words and values as the number of occurrences of each word. Based on this dictionary I created a list *'vocab_list'* which contained only the list of words that occurred for at least 3 times in the entire vocabulary.

To create a bag words model I also created another dictionary *'f_x_dict'* that would store the index of each word in the vocabulary.

Feature Function:

I created a 'dynamic' feature function compatible to create f(x,y) feature matrix for any number of y values (not just [0,1]). For y belonging to {0,1}, my feature vector is of the form



To create this feature vector, I create a begin with a vector of zeros (np.zeros) of the size – vocab length * 2 (to have 0 and 1), then for each sentence I find the word index from *'f_x_dict'* for each words in the sentence and store it in *'word_positions'*. For these words, the values in the feature vector will be non-zero (frequency of occurrence) and stored in *'feature_vector_placeholder'*. Thus, I create f(x,0) and f(x,1) for each of the observation.

These are all implemented in *'feature_function'* function in the code. This function would return an array *'final_feature_representation'* of size (f_vec_size, n), where f_vec_size – feature vector size and n – number of unique y values (2 for our project)

mr6rx-bag-of-words.py - bag-of-words representation for each of the Amazon review.

2.1 Thus the feature size was reduced by removing standardizing words with removing white spaces, converting them to lower case and removing words with frequency less than 3. We finally end up with 31756 features (including the features for y =0 and y = 1) and 15878 unique words

Perceptron Algorithm:

I followed the JE section 2.2.1 to write the code for perceptron and update the theta value accordingly. Theta is initialized with random values for better classification. Functions 'perceptron' and 'predictions' are used for updating theta value using perceptron implementation and calculating the accuracy respectively. I ran the perceptron algorithm for 10 epochs and observed that the training and dev set accuracy improving with each epoch.

(Note: The code took about half a minute for each epoch to train and predict)

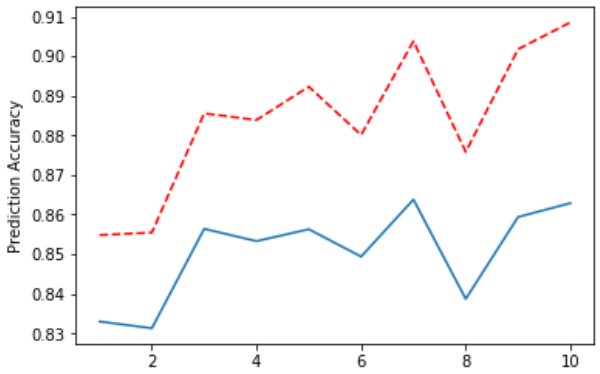
Result from each epoch,

```
Running epoch  1
Completed Training Epoch  1 ...
Performing Evaluation:
Training Accuracy:  0.8598
Dev Accuracy:  0.836

Running epoch  2
Completed Training Epoch  2 ...
Performing Evaluation:
Training Accuracy:  0.8764333333333333
Dev Accuracy:  0.8529

Running epoch  3
Completed Training Epoch  3 ...
Performing Evaluation:
Training Accuracy:  0.8895333333333333
Dev Accuracy:  0.8592
```

Plot showing accuracy across epochs (Red-train; Blue-Dev)



Is it seen that the accuracy saturates at around 85% for dev dataset for 3 epochs increases slightly with increase in epochs

Averaged Perceptron Algorithm:

I followed the JE section 2.2.2 to write the code for perceptron and update the theta value accordingly. This is very similar to the perceptron implementation, but the theta value

is added continuously to another variable m at the end of each time step. Finally, after all the epochs, the m value is divided by the number of time steps in total.

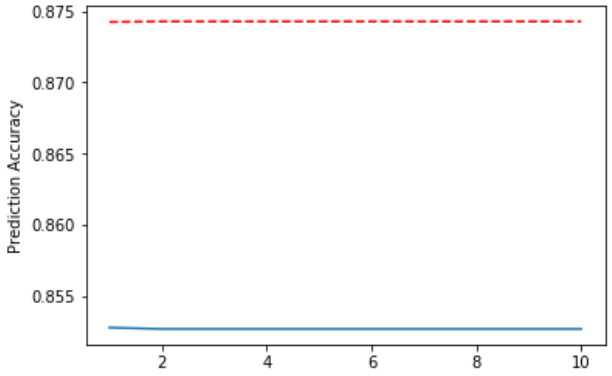
Result from each epoch,

```
Running epoch  1
Completed Training Epoch  1 ...
Performing Evaluation:
Training Accuracy:  0.8742333333333333
Dev Accuracy:  0.8528


Running epoch  2
Completed Training Epoch  2 ...
Performing Evaluation:
Training Accuracy:  0.8742666666666666
Dev Accuracy:  0.8527


Running epoch  3
Completed Training Epoch  3 ...
Performing Evaluation:
Training Accuracy:  0.8742666666666666
Dev Accuracy:  0.8527
```

Plot showing accuracy across epochs (Red-train; Blue-Dev)



Is it seen that the accuracy saturates at 85.5% for dev dataset and never increases with increase in epochs.

Logistic Regression

The input data train_data, dev_data and test_data were used for performing the following steps.

1) Default Settings:

I used the default arguments in *CountVectorizer* function to get the bag-of-words representation for each of the review. Then I fit a *LogisticRegression* model for the obtained bag-of-words data.

The feature size from *CountVectorizer* for each example is: 55577

The classification accuracy for Training set: 98%

The classification accuracy for Dev set: 88.08%

2) Changing n_gram range:

I changed the argument from (1,1) to (1,2) for n_gram_range to include bi-gram words for each of the reviews. Following are the statistics after the change,

The feature size from *CountVectorizer* for each example is: 795411

The classification accuracy for Training set: 99.993%

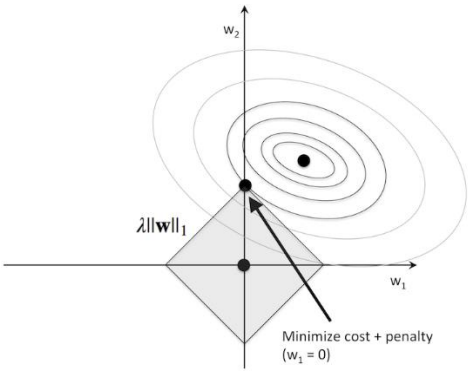
The classification accuracy for Dev set: 90.4%

3) Regularization parameter:

Now, I changed the regularization parameter c (= 1/lambda) for L2 regularization in the *LogisticRegression* function. Following are the statistics after the change,

Lambda	Training Accuracy (%)	Dev Accuracy (%)
10^ -4	100	89.38
10^ -3	100	89.8
10^ -2	100	90.27
10^ -1	100	90.31
1	99.99	90.4
10	99.37	90.5
100	94.45	89.47
5	99.79	90.61
15	98.92	90.42
20	98.43	90.29
30	97.58	90.13

4) L1 Regularization:



L1 regularization uses a slightly different loss function when compared to L2. i.e. L1 uses a lambda times modulus of theta along with the regular loss function in logistic regression. So, now the contour plot for L1 looks as shown above

It can be noted that theta will generally move towards the center point of the contour circle (without the presence of regularization), but with the presence of L1 regularization, we have contours for L1 in the shape (diamond) as shown.

So, the theta will try to move towards the centers of both the contours based on the weights. However, it will settle in between the two regions and it is seen that the nearest point for L1 contour lies on w2 axis which means w1 is zero. Therefore, while trying to settle in between the two regions, theta will find an optimal point in the w2 axis because of which w1 will turn out to be zero.

Thus, at higher dimensions similar situation happens and many other weights become zero and hence we end up with a sparse solution as compared to L2.

- I changed the regularization parameter c (= 1/lambda) for L1 regularization in the *LogisticRegression* function. Following are the statistics after the change,

Lambda	Training Accuracy (%)	Dev Accuracy (%)
0.1	100	89.19
1	99.12	89.62
5	93.3	90
8	91.84	89.39
10	91.23	89.23
15	90.1	88.57
20	89.26	88.03
30	87.95	87.16
50	86.17	85.33

5) Getting the Best Model:

To get the best model I tried the following options,

- **Rich features:** Created a rich feature set. Created uni-gram, bi-gram, tri-gram and four-gram models. Also, I removed words with frequency less than 2.
- **Regularization:** Analyzing the values of lambda from the previous steps, I narrowed down the range to the values 1 to 10. Also, based on the experimentation between L1 and L2, L2 seemed to perform better over L1 in this case.
- **Optimization methods:** Using the *LogisticRegression* from sklearn I tried various optimization options apart from lib-linear.

The best model has the following parameters,

- n_gram_range = c(1,4)

- regularization penalty– ‘l2’
- lambda = 5
- Optimization – liblinear

Training set accuracy: 99.87%
Dev set accuracy : 90.77%

The model was selected based on the highest value on dev set based on the following observations,

- Different n-gram model and even tfidf vectorizer was tried. The accuracy was highest for 4-gram CountVectorizer model and the accuracy started decreasing from 5-gram model. So, 4-gram was chosen as the optimal vocabulary set
- Various lambda values (10^-4 to 100) were tried for both L1 and L2 regularization. And the accuracy for each of the combination was checked on the dev data set before finalizing the optimal parameter
- Various optimization techniques like ‘liblinear’, ‘sag’, ‘newton-cg’ were tried before selecting the final algorithm