ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

Text Analytics
Exercise 2 Report

Papastamos Konstantinos

Kanellis Georgios

Tsolas Leonidas

Exercise

The exercise given is described below:

Develop a text classifier for a kind of texts of your choice (e.g., e-mail messages, tweets, customer reviews) and at least two classes (e.g., spam/ham, positive/negative/neutral).

You may use Boolean, TF, or TF-IDF features corresponding to words or n-grams, to which you can also add other features (e.g., length of the text). You may apply any feature selection (or dimensionality reduction) method you consider appropriate. You may also want to try using centroids of pre-trained word embeddings.

You can write your own code to produce feature vectors, perform feature selection (or dimensionality reduction) and train the classifier (e.g., 2 For e-mail spam filtering, you may want to use the Ling-Spam or Enron-Spam datasets (available from http://nlp.cs.aueb.gr/software.html). For tweets, you may want to use datasets from http://alt.qcri.org/semeval2016/task4/. For customer reviews, you may want to use datasets from

http://alt.qcri.org/semeval2016/task5/.

Consult the instructor for further details. Pre-trained word embeddings are available, for example, from

http://nlp.stanford.edu/projects/glove/.

See also word2vec (https://code.google.com/archive/p/word2vec/). (-1,1): -1 (1,1): -1 (-1,-1): 1 (1,-1): 1 1 -1 0 x1 x1x2 (+1) (+1) 1 0 (-1) -1 (-1) x1 x2 1 -1 using SGD, in the case of logistic regression), or you can use existing implementations and software libraries.

You should experiment with at least logistic regression, and optionally other learning algorithms (e.g., Naive Bayes, k-NN, SVM). Draw learning curves (slides 58, 61) with appropriate measures (e.g., accuracy, F1) and precision-recall curves (slide 26). Include experimental results of appropriate baselines (e.g., classifiers that always assign the most frequent class). Make sure that you use separate training and test data. Tune the feature set and hyper-parameters (e.g., regularization weight λ) on a held-out part of the training data or using a cross-validation (slide 28) on the training data. Document clearly in a short report (max. 5 pages) how your system works (e.g., what algorithms it uses, examples of input/output) and its experimental results (e.g., learning curves, precision-recall curves).

For our implementation we used python and the nltk package. Below all steps taken will be described in detail along with the necessary result screenshots.

The full code used for the exercise can be found here.

1. Data Loading and Preprocessing

First the file of kaggle's imdb reviews dataset was downloaded as found here. Only the "pos"and "neg" folders were kept. After the files were ready, the script iterated over the files, loaded the raw text and used the Clean_Text() function on each individual sentence. The mentioned function does the following:

1. Turns everything to lower case.
2. Removes punctuation
3. Reduces all whitespace down to one.

The result of the loading and preprocessing steps is two lists, "data" which is a list of strings containing the sentences of all files and "labels" which is a list containing 1s and 0s depending on whether the respective sentence in "data" was read from the pos (positive reviews) or neg (negative reviews) folders.

2. Feature Extraction

On this step, the CountVectorizer function was used from the sklearn package to create unigram and bigram datasets. Two different instances were created for that purpose:

Unigram_vectorizer

The unigram vectorizer created a sparse matrix of word counts. Just for unigrams we decided to remove stopwords in order to reduce dimensionality and also because we made the assumption that stopwords do not add information on the sentiment of a sentence when working with unigrams. Finally unigrams with document frequency less than 15 were ignored.

Bigram_vectorizer

The Bigram vectorizer created a sparse matrix of Bigram counts. Bigrams with document frequency less than 15 were ignored.

3. Data Shaping and Preprocessing

3.1. Set Formation

The next step was to randomly split the corpus into training, validation and test sets. The splitting was done two times, one for unigrams and one for bigrams resulting in 6 different sets. The splitting was done using sklearn's train_test_split function.

A 80 - 10 - 10 split strategy was used to form the training, validation and test sets.

The resulting shapes are the following:

```
Unigram Training Dataset: (20000, 14379)
Unigram Validation Dataset: (2500, 14379)
Unigram Test Dataset: (2500, 14379)
Bigram Training Dataset: (20000, 40859)
Bigram Validation Dataset: (2500, 40859)
Bigram Test Dataset: (2500, 40859)
```

3.2. Dimensionality Reduction

However after inspecting the resulting datasets, the number of created features is obviously too large. So we decided to use truncated SVD in order to reduce the dimensions as much as possible.

We used the TruncatedSVD module from the sklearn.decomposition package and performed SVD on both the unigram and bigram training sets. For the unigram training set we kept 4000 components while for the bigram set we kept 8000, since the bigram set had initially many more features. The choice was made after several tries and by consulting the explained variance ratio for different number of components. The resulting sets have an explained variance ratio as seen below:

```
In [47]: print('Explained Variance of Unigram model: ', pretty_percentage(uni_ex_var))
    ...: print('Explained Variance of Bigram model: ', pretty_percentage(bi_ex_var))
Explained Variance of Unigram model:  94.06%
Explained Variance of Bigram model:  90.73%
```

So the resulting dimensions after using Truncated SVD are:

```
In [26]: print('Reduced Shape of Unigram Data:',reduced_uni_train.shape)
    ...: print('Reduced Shape of Bigram Data:',reduced_bi_train.shape)
Reduced Shape of Unigram Data: (20000, 4000)
Reduced Shape of Bigram Data: (20000, 8000)
```

4. Modelling

On the next step we used the sklearn package to train different classifiers and evaluate the results.

4.1. KNN

After forming the necessary sets, we started experimenting with different models. First we used a simple KNN model in order to establish a baseline, setting the number of neighbors to 5. We trained the classifier on the training set for both unigrams and bigrams and tested it on the validation set. The KNN model achieved a low accuracy of 61% for Unigram data and an even worse 50% on the bigram set. Since the KNN classifier performed so poorly we decided to not analyze the learning curves and scores further and try a different classifier.

4.2. Logistic Regression

4.2.1. Baseline

Next we trained a logistic regression classifier. On this case we decided to use l1 regularization, since it seemed to yield better accuracy than l2, and trained the model on both bigram and unigram datasets. First we trained a baseline model, setting the penalty parameter to l1 and the inverse regularization term to 0.7, in order to decide which set we will exclude and on which we will experiment further. The resulting accuracies are seen below:

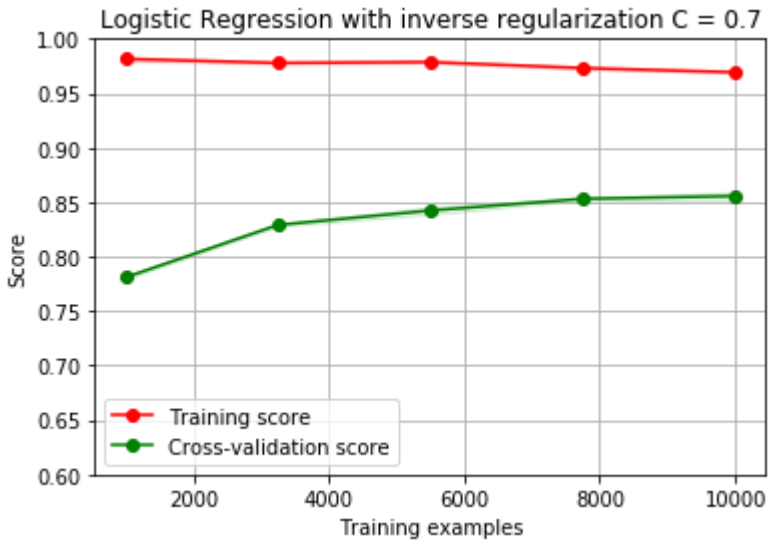
```
In [50]: print('Unigram baseline model accuracy:',
    ...:       pretty_percentage(accuracy_score(y_true = Y_Uni_Validation, y_pred = Logistic_Uni_Pred)))
    ...:
    ...: print('Bigram baseline model accuracy:',
    ...:       pretty_percentage(accuracy_score(y_true = Y_Bi_Validation, y_pred = Logistic_Bi_Pred)))
Unigram baseline model accuracy: 87.04%
Bigram baseline model accuracy: 49.48%
```

The unigram model seems to perform always better so the bigram dataset will be excluded from the tuning phase.

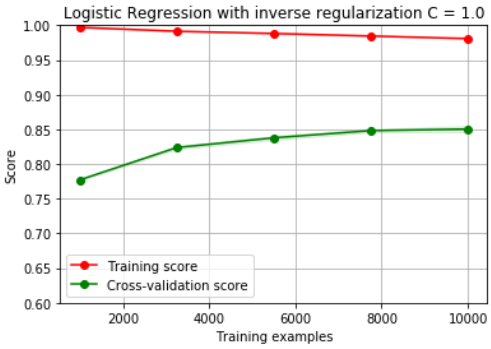
4.2.2. Tuning

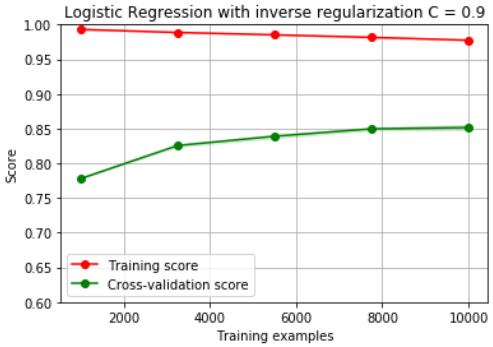
Regularization

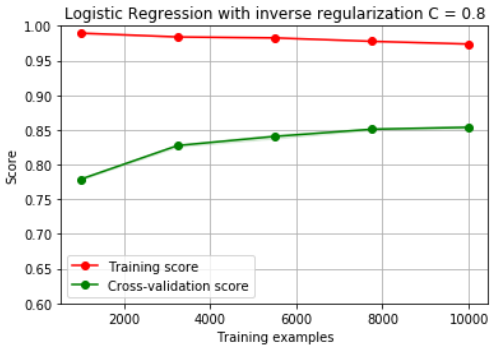
After deciding to use the Unigram set for training, we plotted the learning curves of the classifier to get a better idea of its performance:

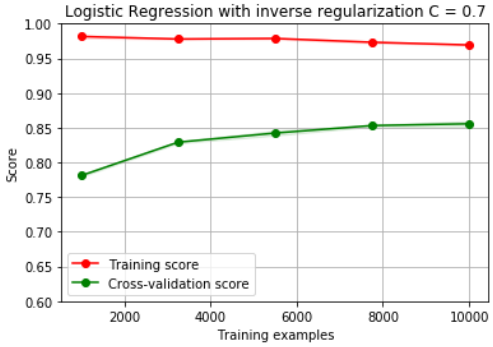


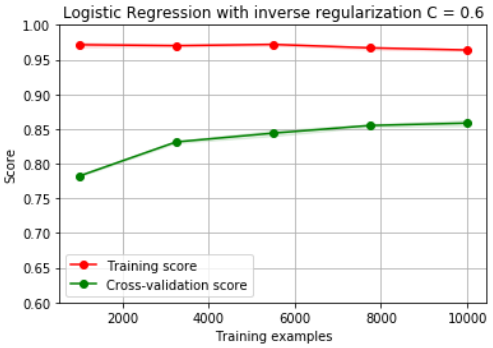
It seems that the classifier is greatly overfitting on the training data. In order to tackle this issue we decided to experiment with different values for the regularization term. The results are seen below:

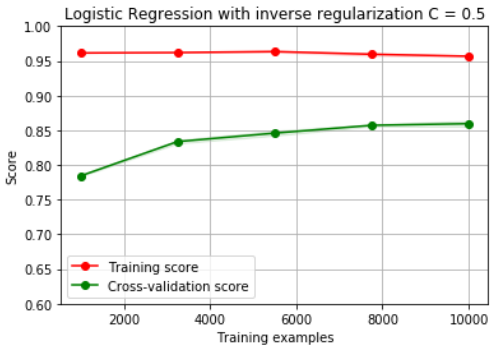


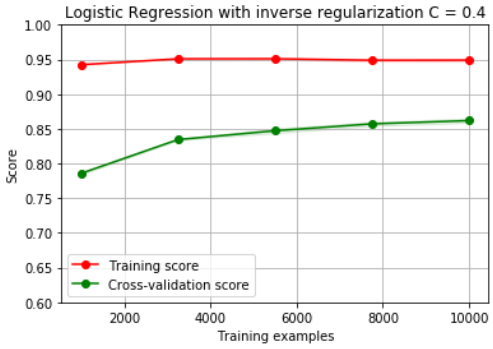


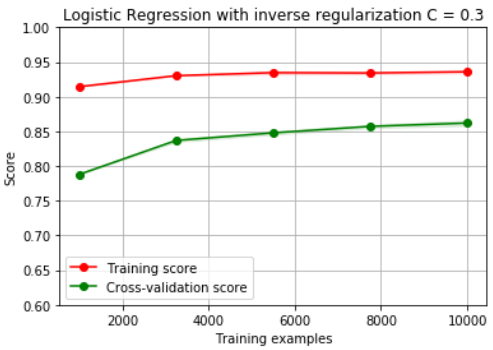


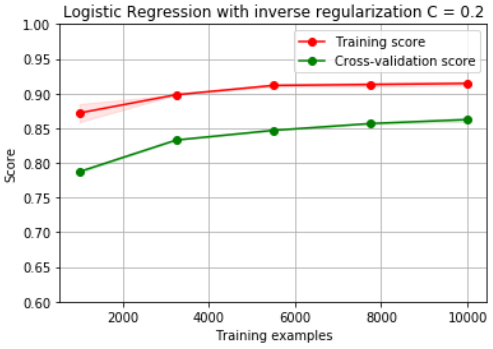


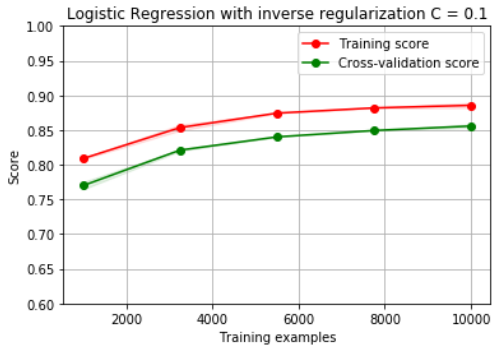








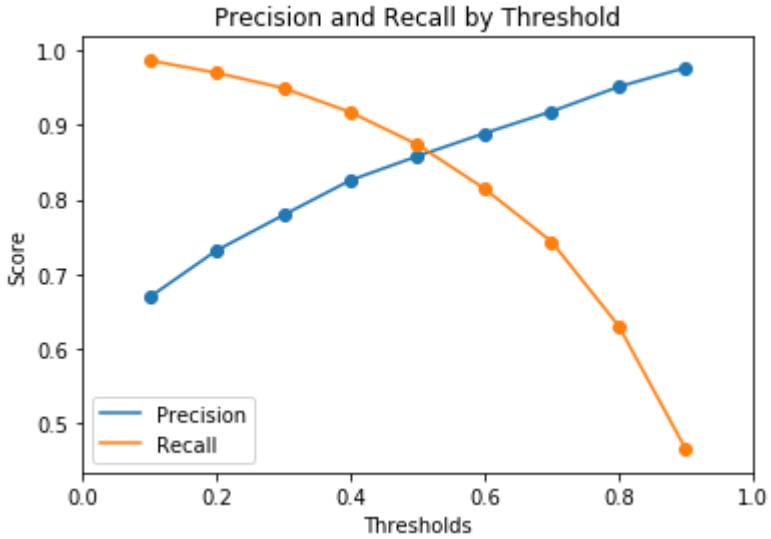




From the results we concluded that in order to avoid overfitting we will use the 0.1 as the final C value.

Decision Boundary

We also tuned the decision threshold in order to find the optimal value that maximized both precision and recall scores. In order to do that we implemented a Tune_Threshold() function that plots the resulting scores for different threshold values. The results can be seen below:



It seems that the default 0.5 threshold maximizes both scores so it will not be changed when using the final model.

4.2.3. Testing

So after the tuning step we concluded on the Logistic Regression classifier with an inverse regularization strength of 0.1, using the l1 penalty and utilizing the threshold 0.5 in order to assign samples to classes. The resulting classifier yielded the following results on the Unigram Test Set:

```
Accuracy:
88.2%

Precision:
87.26%

Recall:
89.42%

Confusion Matrix:
             Negative Positive
    Negative   1089.0    163.0
    Positive    132.0   1116.0
```