COMP9318 (18S1) FINAL PROJECT
DUE ON 23:59 27 MAY, 2018 (SUN)

z5136414 Andrew Wirjaputra
z5119751 Shiqing Zhang

SHORT ABSTRACT

This report briefly describes an algorithm devised to fool a binary SVM text classifier. The commonly used bag-of-words model is chosen to represent the training examples. Considering the fact that some words appear more frequently in general, the algorithm also takes into account the frequency of the word in the corpus. An SVM will then be trained in order to mimic the target classifier. The algorithm will then use the weights obtained from the trained SVM to decide on which words to modify in the test samples to fool the target classifier.

1. INTRODUCTION

This project focuses on the design and implementation of fool_classifier(), an algorithm devised to fool a binary SVM text classifier named target-classifier. We were given access to part of classifiers' training data (a sample of 540 paragraphs, 180 for class-1, and 360 for class-0, provided in the files: class-1.txt and class-0.txt respectively), and a test sample of 200 paragraphs from class-1 (in the file: test_data.txt). The goal is to make exactly 20 distinct modifications in each test sample, so that the target-classifier misclassifies it as class-0.

2. METHODOLOGY WITH JUSTIFICATIONS

There are three main tasks that we need to do for this project. First, we need to decide on a way to represent the training examples into feature vectors. Secondly, we then need to select proper SVM parameters to mimic the target-classifier. And finally, we need to modify the test data so that the target-classifier misclassifies them as class-0 data.

2.1 Vectorization

Since the project specification requires us to classify paragraphs of varying length and content, we need a way of representing them as feature vectors. The bag-of-words model is one of the most commonly used models in text processing, where the frequency of each word is used as a feature for training a classifier. However, given the constraint that we should not preprocess the data, we need to consider the fact that some words appear more frequently in general (i.e. punctuations, prepositions). This can be offset by also considering the frequency of the word in the corpus. So all in all, we will use the term frequency-inverse document frequency (TF-IDF) of each word as a feature for training the classifier.

```
strategy_instance = helper.strategy()

training_data = []
for index in range(len(strategy_instance.class0)):
    training_data.append((get_freq_of_tokens(strategy_instance.class0[index]), "class0"))
for index in range(len(strategy_instance.class1)):
    training_data.append((get_freq_of_tokens(strategy_instance.class1[index]), "class1"))
x_train, y_train, vocabulary = analyze(training_data)
```

As can be seen in the code snippet above, we first create an instance of class strategy(), in which its __init__() function will read and extract the words that made up each training example (from 'class-0.txt' and 'class-1.txt'). By utilizing get_freq_of_tokens(), we will then build up a training_data that contain each of these training examples represented as a dictionary of words and their corresponding frequency, along with the label of the training example. By calling analyze() on this training_data, we can then extract the x_train and y_train data that will be used to train the SVM, and vocabulary of the training data. The function analyze() will use the TF-IDF of each word in the vocabulary as feature, so each training example will be represented by a vector of the same length (but with different values).

2.2 Parameter Selection

When selecting a Kernel for SVM, we should consider how the number of features is compared to the number of training examples. If the number of features far outweigh the number of training examples, mapping the data to a higher dimensional space shouldn't be necessary. That is, using non-linear kernel does not improve performance. Since the number of features (5718) extracted from the training data is many times higher than the number of training examples (540), we will be using a linear kernel with default parameters.

```
parameters = {'gamma':'auto', 'C':1, 'kernel':'linear', 'degree':3, 'coef0':0.0}
clf = strategy_instance.train_svm(parameters, np.array(x_train), np.array(y_train))
word_coef = clf.coef_[0].tolist()
word_coef = {vocabulary[idx]:word_coef[idx] for idx in range(len(vocabulary))}
```

As we can see in the code snippet above, we first initialize the parameters used for train_svm(), and supply it to the function along with the x_train and y_train data obtained from analyze(). We can then map the weights of the trained SVM classifier (clf.coef_) with their corresponding word in the vocabulary. These weights show the significance of each word to a specific class. The more positive the weight is, the more important the word is to class-1. And the more negative the weight is, the more important the word is to class-0.

2.3 Test Sample Modification

Since we are only allowed to modify each test sample by exactly 20 distinct tokens, we cannot simply consider the 20 most important words in the training vocabulary. Instead, we will need to create a list of words for each test sample, sorted based on their importance obtained during training. Words that are not seen in the training data will be padded with weights of zero. Using these lists, we can then modify each test sample based on the 20 most important words found in it.

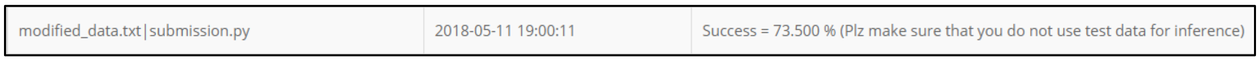
Modifications allowed on the test data can be classified as either insertion or deletion (each considered as one modification). Replacement will be considered as two modifications, since it involves deletion followed by insertion. Considering the rather large amount of features, the weight of each word are not that much different from each other. Based on this, we can safely assume that doing 2 deletions would often be more efficient than doing 1 replacement.

```
modified_test_documents = []
for document in test_documents:
    word_coef_test = []
    for word in set(document):
        if word in vocabulary:
            word_coef_test.append((word_coef[word], word))
        else:
            word_coef_test.append((0, word))
    word_coef_test.sort(reverse=True)
    to_delete = [word_coef_test[idx][1] for idx in range(20)]

    modified_document = []
    for word in document:
        if not word in to_delete:
            modified_document.append(word)
    modified_test_documents.append(modified_document)
```

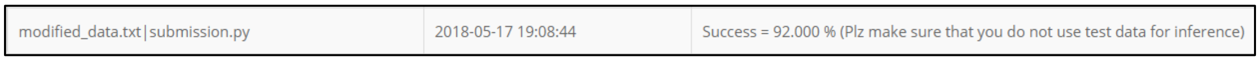
3. RESULTS AND CONCLUSIONS

At first, we tried to do 10 replacements on each test sample, and we only get a maximum success percentage of 73.5%.



modified_data.txt|submission.py   2018-05-11 19:00:11   Success = 73.500 % (Plz make sure that you do not use test data for inference)

We then tried to do 20 deletions on each test sample, and as expected the success percentage shot up to 92%. The reason behind this is because the number of features are fairly large, which cause the weight of each word to be not that much different from each other. Based on this, doing 2 deletions would often be more efficient than doing 1 replacement.



modified_data.txt|submission.py   2018-05-17 19:08:44   Success = 92.000 % (Plz make sure that you do not use test data for inference)

Since the number of features are fairly large compared to the number of training examples, and also considering the fact we can already achieve a high success rate using the linear kernel, we don't think that it's necessary to try on the non-linear kernels, as they are not necessarily more accurate than the linear one in high dimensions.