Course project report

CS18S012
CE16B034

Sunday 10th November, 2019

Introduction and brief summary

In this assignment we are asked to build an email spam classifier from scratch. We have used the Enron spam email data set to train our classifier. We used the NLTK library in Python to do the bulk of our prepossessing. Finally we modelled the data using Naive Bayes Classifier, Logistic Regression and SVM.

The Dataset

The Enron data set contains emails received and sent by the senior management of Enron which were made public due to the Enron scandal. We have used this data set because we couldn't find a more comprehensive set of emails which are classified into ham or spam. There are about 33,716 emails of which 16,545 are ham and 17,171 are spam. Most of the emails are from the late 1990's to the early 2000's, which might make this data set a little irrelevant today. The data is organised in 6 folders named from Enron1 through Enron6, each folder has again two folders named "Spam" and "Ham" which contain the Spam and Ham emails respectively in text files. Since the data set is huge the reading of the emails into memory itself takes a non trivial amount of time to run.

Preprocessing and Feature extraction

We read the data given in spam and ham folders in each of the six Enron folders separately into two lists spam_lst and ham_lst. Each element of the list contains one email in the form of a giant string, we have used "latin-1" encoding to read data because it can handle special characters and rarely gives errors. We now list the various preprcessing techniques we have used:

Tokenization

In very basic terms, word tokenization is the process of splitting a paragraph/sentence into individual words and punctuation marks. We can then run other preprocessing steps on these individual words to obtain better features at the end. As part of this step we also convert all the words to lower case letters.

Stop words

Stop words are the commonly used words in a sentence which add no additional meaning to it, some examples include "a, an, the, I ", etc. It makes sense to remove them completely since they carry no meaning and the feature size is reduced.

Stemming and Lemmatization

Both Stemming and Lemmatization are what are known as text normalization techniques. They reduce the "inflection" of the words in a given language. For example playing and played are the inflected words of play, therefore transforming played, playing and play to a common word makes more sense since they convey the same meaning and also reduce the features. Stemming(we used the Porter Stemmer algorithm) does this by using a complex set of rules to remove what it determines as suffix and maps a group of words to a stem, the stem might not even be a valid word in the language because after stripping the suffix, the word might not be in the dictionary. Lemmatization on the other hand is a more sophisticated technique that returns a stem which is a legal word in the language. After applying all the preprocessing steps above we again join the words into a sentence. A more authoritative resource is here.

Tfidf Vectorizer

We need to convert the sentences we obtained into numerical features so that our models can process them. The most basic of these "vectorizers" is the count vectorizer which just counts the number of words and outputs that as a feature. It might make more sense to somehow penalize the words which are common in all the documents since they are unlikely to be an important feature. So the the Tfidf vectorizer outputs a number for a feature(word) which is proportional to the number of times it occurs in the given document and is inversely proportional to the number of times it occurs in all the documents. A more concrete resource with mathematical details is here. After running the Tfidf vectorizer for the above sentences we obtained a feature vector in excess of 1,00,000 features(unique words in all documents)! Running models on such a huge data matrix was practically impossible(in fact our laptop crashed twice), so we reduced it to the top 1000 most common features(words), which is giving good accuracy for a reasonable amount of training time. The code in Python is given below:

```
ham_lst = []
spam_lst = []
# following code iterates through Enron1 ,..., Enron6 extracting the ↘
→spam and ham emails
for d,sd,f in os.walk(path):
    print(d)
    if os.path.split(d)[1] == "ham":
        for filename in f:
            with open(os.path.join(d,filename),encoding="latin-1") as ↘
            →s:
                # input the data
                data = s.read()
                # tokenize the data
                data = word_tokenize(data)
                # convert to lower case words
                data = [word.lower() for word in data]
                # remove the stop words
                data = [word for word in data if word not in stopwords↘
                →.words("english")]
                # use porter stemmer and lemmatization
                stemmer = PorterStemmer()
                lemmatizer = WordNetLemmatizer()
                data = [stemmer.stem(word) for word in data]
                data = [lemmatizer.lemmatize(word) for word in data]
                ham_lst.append(data)
    elif os.path.split(d)[1] == "spam":
        for filename in f:
            with open(os.path.join(d,filename),encoding="latin-1") as ↘
            →s:
                # input the data
                data = s.read()
                # tokenize the data
                data = word_tokenize(data)
                # convert to lower case words
                data = [word.lower() for word in data]
                # remove the stop words
                data = [word for word in data if word not in stopwords↘
                →.words("english")]
                # use porter stemmer and lemmatization
                stemmer = PorterStemmer()
                lemmatizer = WordNetLemmatizer()
                data = [stemmer.stem(word) for word in data]
                data = [lemmatizer.lemmatize(word) for word in data]
                spam_lst.append(data)
# contain all the data into one list
ham_lst.extend(spam_lst)
y = np.array([0]*(len(ham_lst)-len(spam_lst))+[1]*len(spam_lst))
# split into training and test data
txt_train, txt_test, y_train, y_test = train_test_split(ham_lst,y)
# combine words into sentences for running the tf-idf vectorizer
txt_train = [' '.join(words) for words in txt_train]
txt_test = [' '.join(words) for words in txt_test]
# define max features to limit the size of the matrix
vectorier = TfidfVectorizer(max_features=1000)
# fit and transform for train
X_train = vectorier.fit_transform(txt_train)
# only transform don't fit for test
X_test = vectorier.transform(txt_test)
# convert sparse matrix into ordinary matrix
```

```
X_train = X_train.toarray()
X_test = X_test.toarray()
```

Building Models

We ran three separate classifiers on our data set.

Naive Bayes Classifier

We used a Gaussian likelihood here, because the Tfidf vectoizer outputs continuous real values and hence it might not make sense to use the Multinomial Naive Bayes classifier here.

$$P(y \mid x_1, \dots, x_n) = \frac{P(y) \prod_{i=1}^{n} P(x_i \mid y)}{P(x_1, \dots, x_n)} \tag{1}$$

where y is the class value we want to predict
and $x_1, ..., x_n$ are the features for a particular data point. and each feature is estimated by a gaussian distribution as follows:

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right) \tag{2}$$

The mean and the variance for each class and each feature are just the MLE estimates and hence the sample mean and variances respectively. Instead of taking the product as shown in equation 1, we have equivalently taken sum of the log values for better numerical stability. Also we have added a small value to the standard deviation values for better numerical stability. The code in Python is given below:

```python
# training the classifier
# just get the means and standard deviations
k = len(np.unique(y_train))
mean = np.zeros((k,X_train.shape[1]))
std = np.zeros((k,X_train.shape[1]))
for i in range(k):
    mean[i] = np.mean(X_train[y_train == i],0)
    std[i] = np.std(X_train[y_train == i],0)
class_prob = []
for i in range(k):
    class_prob.append(np.sum(y_train == i) / len(y_train))
# add a small value for numerical stability
std += 1e-6
# store the predicted values
y_pred = []
for x in X_test:
    # here instead of product we are taking sum of log for numerical ↘
    → stability
    y_pred.append(np.argmax([-np.sum(np.log(std[i]) + (x-mean[i])↘
    →**2/(2*std[i]**2)) + np.log(class_prob[i]) for i in range(k)]))
```

After running the above code we got an accuracy of 0.97 on our test set.

Logistic Regression

Logistic regression assumes the sigmoid function for the posterior class probabilities. We used the loss function defined in class based on MLE to derive it's gradient and used the gradient descent update rule to minimize the loss, with learning rate = 1e-4 and number of iterations = 1500. We can see from Fig 1 that the loss function converges.The code in Python is given below:

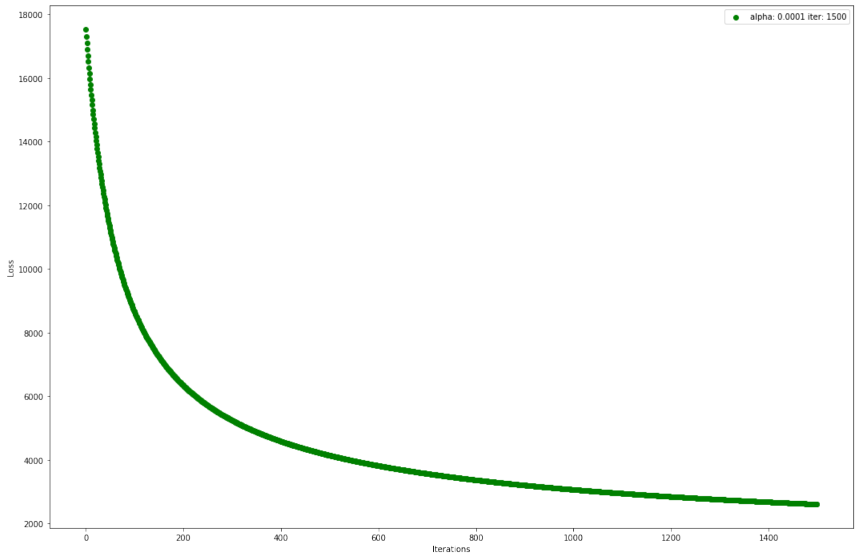

Figure 1: Logistic regression loss function v/s iterations

```
# we will now implement the above in code
# add a row of 1's to include a constant in linear function
X_train_affine = np.hstack([X_train,np.array([1]*len(X_train)).reshape↘
→(-1,1)])
X_test_affine = np.hstack([X_test,np.array([1]*len(X_test)).reshape↘
→(-1,1)])
y_train = y_train.reshape(-1,1)
min_loss = math.inf
alpha = 1e-4
itera = 1500
beta = np.random.randn(X_train_affine.shape[1],1) * 0.001
loss = []
for i in range(itera):
    # add the current loss to vector which is negative log liklihood
    loss.append( -(np.sum((X_train_affine @ beta) * y_train) - np.sum(↘
    →np.log(1+np.exp(X_train_affine @ beta)))) )
    # calulate the gradient
    grad = (np.sum(y_train * X_train_affine,0) - np.sum(X_train_affine↘
    → / (1 + np.exp(-X_train_affine @ beta)),0)).reshape(-1,1)
    # update using gradient ascent update rule, to maximize log ↘
    →liklihood
    beta = beta + alpha * grad
plt.figure(figsize=(18,12))
plt.scatter(list(range(len(loss))),loss,c='g',label="alpha: "+str(↘
→alpha)+" iter: "+str(itera))
plt.xlabel("Iterations")
plt.ylabel("Loss")
plt.legend()
y_pred = ((X_test_affine @ beta) > 0)
f1_score(y_test,y_pred)
```

Support Vector Machines

SVM finds the maximal margin separating hyperplane(along with a certain slack for misclassifications) as discussed in class. For this part we use Sklearn's SVC(Support Vector Classifier) function. We first tried rbf kernel, but it is giving a poorer result than linear kernel. So we stuck with linear kernel and did validation to find optimal value of C. We found that SVM gave us an accuracy of 97.9

```
# since we have already split the data, we can use X_test as a ↘
→validation code:
val_acc = []
for C in np.logspace(0,2,10):
    # fit the data
    clf = SVC(kernel="linear",C=C)
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    val_acc.append(np.sum(y_pred.ravel() == y_test) / len(y_test))
```

Results and Prediction

We found that out of the three SVM performs best. However for the final prediction we are taking a vote out of the three classifiers.