DS-GA 1003: Machine Learning (Spring 2020)

Homework 1: Perceptron algorithm

Solutions

In this problem set you will implement the Perceptron algorithm and apply it to the problem of e-mail spam classification.

Instructions. You are **not** permitted to use any machine learning code or packages such as `sklearn`. You should upload your questions along with plots and code to Gradescope. You must follow the policies for **submission** detailed in Homework 0.

Dataset. We have provided you with two files in the shared/ directory on JupyterHub: `spam_train.txt`, `spam_test.txt`. Each row of the data files corresponds to a single email. The first column gives the label (1=spam, 0=not spam).

> Anyone knows how much it costs to host a web portal ?
> Well, it depends on how many visitors youre expecting. This can be anywhere from less than 10 bucks a month to a couple of $100. You should checkout http://www.rackspace.com/ or perhaps Amazon EC2 if youre running something big..

To unsubscribe yourself from this mailing list, send an email to: groupname-unsubscribe@egroups.com

Figure 1: Sample e-mail in SpamAssassin corpus before pre-processing.

The dataset is based on a subset of the SpamAssassin Public Corpus. Figure 1 shows a sample email that contains a URL, an email address (at the end), numbers, and dollar amounts. While many emails would contain similar types of entities (e.g., numbers, other URLs, or other email addresses), the specific entities (e.g., the specific URL or specific dollar amount) will be different in almost every email. Therefore, one method often employed in processing emails is to "normalize" these values, so that all URLs are treated the same, all numbers are treated the same, etc.

For example, we could replace each URL in the email with the unique string "httpaddr" to indicate that a URL was present. This has the effect of letting the spam classifier make a classification decision based on whether any URL was present, rather than whether a specific URL was present. This typically improves the performance of a spam classifier, since spammers often randomize the URLs, and thus the odds of seeing any particular URL again in a new piece of spam is very small.

We have already implemented the following email preprocessing steps: lower-casing; removal of HTML tags; normalization of URLs, e-mail addresses, and numbers. In addition, words are reduced to their stemmed form. For example, "discount", "discounts", "discounted" and "discounting" are all replaced with "discount". Finally, we removed all non-words and punctuation. The result of these preprocessing steps is shown in Figure **??**.

1 **(3 points)** This problem set will involve your implementing several variants of the Perceptron algorithm discussed in Section 1 and Lecture 2. Before you can build these models and

measure their performance, split your training data (i.e. `spam_train.txt`) into a training and validate set, putting the last 1000 emails into the validation set. Thus, you will have a new training set with 4000 emails and a validation set with 1000 emails. You will not use `spam_test.txt` until problem 10.

Explain why measuring the performance of your final classifier would be problematic had you not created this validation set.

Answer: Since the training error is always 0 if training samples are linearly separable, if we had not splitted the data, we could not estimate the test error or the generalization ability of our model.

```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from tqdm import tqdm

def read_data(filename):
    '''
    Takes filename as input.
    Returns two arrays for text and labels.
    '''
    with open(filename) as fp:
        lines = fp.readlines()
        text,labels = [],[]
        for line in lines:
            text.append(line[2:].split())
            labels.append(float(line[0]))
    return text,labels

#Loading train data
text, labels = read_data("spam_train.txt")

#Splitting into train and val sets
train_data, train_labels = text[:4000],labels[:4000]
val_data, val_labels = text[4000:],labels[4000:]
```

2 **(6 points)** Transform all of the data into feature vectors. Build a vocabulary list using only the 4000 e-mail training set by finding all words that occur across the training set. Note that we assume that the data in the validation and test sets is completely unseen when we train our model, and thus we do not use any information contained in them. Ignore all words that appear in fewer than $X = 30$ e-mails of the 4000 e-mail training set – this is both a means of preventing overfitting and of improving scalability. For each email, transform it into a feature vector $\vec{x}$ where the ith entry, x_i, is 1 if the ith word in the vocabulary occurs in the email, and 0 otherwise.

```python
def build_vocab(df):
    '''
    Return a vocabulary list based on data provided
    '''
    vocab = {}
    for email in df:
        for word in set(email):
            if word in vocab:
                vocab[word]+=1
            else:
                vocab[word] = 1

    vocab = [k for k,v in sorted(vocab.items()) if v >=30]
    return vocab

def featurize_data(df,vocab):
    '''
    Takes a dataset and a dictionary of word_to_index as input
    Returns featurized dataset with 0s and 1s
    '''
    featurized_df = []
    for i in tqdm(df):
        featurized_df.append([1 if word in i else 0 for word in vocab])
    featurized_df = np.stack(featurized_df)
    return featurized_df

#Building voabulary on train data (4000 examples)
vocab = build_vocab(train_data)
word_to_index = {vocab[i]:i for i in range(len(vocab))}
index_to_word = {i:vocab[i] for i in range(len(vocab))}

train_featurized = featurize_data(train_data, word_to_index)
val_featurized = featurize_data(val_data, word_to_index)
```

3 **(12 points)** Implement the functions `perceptron_train(data)` and `perceptron_test(w, data)`.

The function `perceptron_train(data)` trains a perceptron classifier using the examples provided to the function, and should return $\vec{w}$, k, and $iter$, the final classification vector, the number of updates (mistakes) performed, and the number of passes through the data, respectively. You may assume that the input data provided to your function is linearly separable (so the stopping criterion should be that all points are correctly classified).

For this exercise, you do not need to add an offset feature to the feature vector (it turns out not to improve classification accuracy, possibly because a frequently occurring word already serves this purpose). Your implementation should cycle through the data points in the order as given in the data files (rather than randomizing), so that results are consistent for grading purposes.

The function `perceptron_test(w, data)` should take as input the weight vector $\vec{w}$ (the classification vector to be used) and a set of examples. The function should return the test error, i.e. the fraction of examples that are misclassified by $\vec{w}$.

```python
def perceptron_train(x,y,num_iters=None):
    '''
    Implementation of the perceptron algorithm
    Takes data as input
    Return w, k and iters
    '''
    y = [-1 if i == 0 else 1 for i in y]
    w = np.zeros(x.shape[-1])
    k, iters = 0,0

    while(True):
        n_incorrect = 0
        for i in range(len(x)):
            if y[i]*(np.dot(w,x[i])) <= 0 and np.sum(x[i])!=0:
                w += y[i]*x[i]
                k+=1
                n_incorrect+=1
        iters+=1

        if num_iters!=None:
            if iters==num_iters:
                break
        else:
            if n_incorrect == 0:
                break
    return w,k,iters

def perceptron_test(w,x,y):
    preds = np.dot(x,w)
    preds = np.array([1 if i >= 0 else 0 for i in preds])
    correct_preds = np.sum(preds != y)
    return float(correct_preds)/len(y)
```

4 **(6 points)** Train the linear classifier using your training set. How many mistakes are made before the algorithm terminates? Test your implementation of `perceptron_test` by running it with the learned parameters and the training data, making sure that the training error is zero. Next, classify the emails in your validation set. What is the validation error?

```python
w,k,iters = perceptron_train(train_featurized,train_labels)
print("Train error = {}".format(perceptron_test(w,train_featurized,train_labels)))
print("Number of mistakes made = {}".format(k))
print("Validation error =  {}".format(perceptron_test(w,val_featurized,val_labels)))
```

Train error = 0.0
Number of mistakes made = 437
Validation error = 0.013

5 **(3 points)** To better understand how the spam classifier works, we can inspect the parameters to see which words the classifier thinks are the most predictive of spam. Using the vocabulary list together with the parameters learned in the previous question, output the 15 words with the *most positive* weights. What are they? Which 15 words have the most *negative* weights?

```python
word_weights = {vocab[i]:w[i] for i in range(len(vocab))}
sorted_words = [(k,v) for k,v in sorted(word_weights.items(),key=lambda item: item[1])]

print("Bottom 15 words with negative weights :")
sorted_words[:15]

print("Top 15 words with positive weights :")
sorted_words[-15:][::-1]
```

Bottom 15 words with negative weights :

('wrote', -16.0), ('reserv', -15.0), ('prefer', -14.0), ('copyright', -13.0), ('i', -12.0), ('still', -12.0), ('technolog', -12.0), ('but', -11.0), ('comput', -11.0), ('recipi', -11.0), ('someth', -11.0), ('which', -11.0), ('coupl', -10.0), ('date', -10.0), ('url', -10.0)

Top 15 words with positive weights :

('sight', 22.0), ('click', 18.0), ('these', 16.0), ('remov', 16.0), ('market', 16.0), ('our', 15.0), ('deathtospamdeathtospamdeathtospam', 14.0), ('most', 13.0), ('yourself', 12.0), ('present', 12.0), ('parti', 12.0), ('ever', 12.0), ('pleas', 11.0), ('guarante', 11.0), ('check', 11.0)

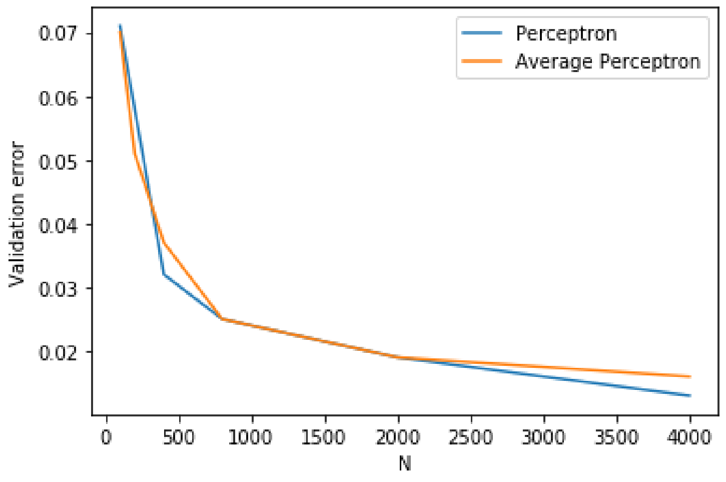
6 **(6 points)** Implement the *averaged* perceptron algorithm, which is the same as your current implementation but which, rather than returning the final weight vector, returns the average of all weight vectors considered during the algorithm (including examples where no mistake was made). Averaging reduces the variance between the different vectors, and is a powerful means of preventing the learning algorithm from overfitting (serving as a type of regularization).

```python
def average_perceptron(x,y, num_iters = None):
    y = [-1 if i == 0 else 1 for i in y]
    w = np.zeros(x.shape[-1])
    avg_weight = np.zeros(x.shape[-1])
    k, iters = 0,0

    while(True):
        n_incorrect = 0
        for i in range(len(x)):
            if y[i]*(np.dot(w,x[i])) <= 0 and np.sum(x[i])!=0:
                w += y[i]*x[i]
                k+=1
                n_incorrect+=1
                avg_weight += w
            else:
                avg_weight += w
        iters+=1

        if num_iters!=None:
            if iters==num_iters:
                break
        else:
            if n_incorrect == 0:
                break

    return avg_weight/(iters*len(x)),k,iters

w,k,iters = average_perceptron(train_featurized,train_labels)
print("Training error: {}".format(perceptron_test(w,train_featurized,train_labels)))
print("Validation error: {}".format(perceptron_test(w,val_featurized,val_labels)))
```

Train error: 0.00075
Val error: 0.016

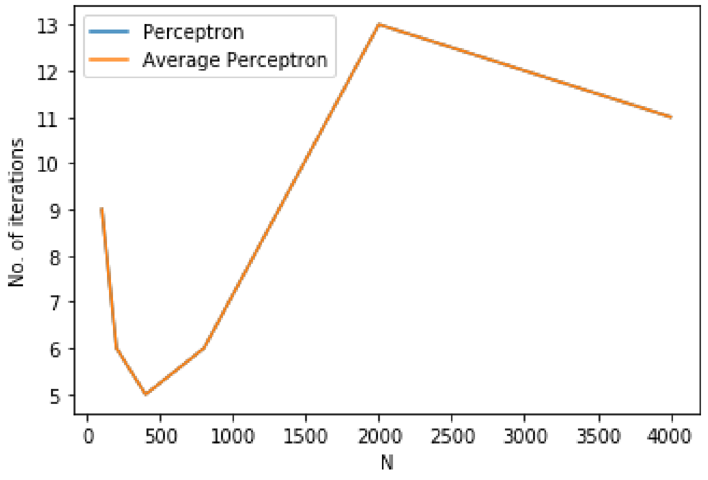
7 **(3 points)** One should expect that the test error decreases as the amount of training data increases. Using only the first N rows of your training data, run both the perceptron and the averaged perceptron algorithms on this smaller training set and evaluate the corresponding validation error (using all of the validation data). Do this for $N = 100, 200, 400, 800, 2000, 4000$, and create a plot of the validation error of both algorithms as a function of N.

```
percep_iters, avg_percep_iters = [], []
percep_err, avg_percep_err = [],[]
N = [100,200,400,800,2000,4000]

for n in N:
    w,k,iters = perceptron_train(train_featurized[:n],train_labels[:n])
    percep_iters.append(iters)
    percep_err.append(perceptron_test(w, val_featurized,val_labels))

    w,k,iters = average_perceptron(train_featurized[:n],train_labels[:n])
    avg_percep_iters.append(iters)
    avg_percep_err.append(perceptron_test(w, val_featurized,val_labels))

#Plotting validation error vs N curve
plt.plot(N, percep_err, label="Perceptron")
plt.plot(N, avg_percep_err, label="Average Perceptron")
plt.xlabel("N")
plt.ylabel("Validation error")
plt.legend()
plt.show()
```



8 **(3 points)** Also for $N = 100, 200, 400, 800, 2000, 4000$, create a plot of the number of perceptron iterations as a function of N, where by iteration we mean a complete pass through the training data. As the amount of training data increases, the margin of the training set decreases, which generally leads to an increase in the number of iterations perceptron takes to converge (although it need not be monotonic).

```python
#Plotting iterations vs N curve
plt.plot(N, percep_iters, label="Perceptron")
plt.plot(N, avg_percep_iters, label="Average Perceptron")
plt.xlabel("N")
plt.ylabel("No. of iterations")
plt.legend()
plt.show()
```



9 **(3 points)** One consequence of this is that the later iterations typically perform updates on only a small subset of the data points, which can contribute to overfitting. A way to solve this is to control the maximum number of iterations of the perceptron algorithm. Add an argument to both the perceptron and averaged perceptron algorithms that controls the maximum number of passes over the data.

Try various configurations of the algorithms on your own using all 4000 training points, and find a good configuration having a low error on your validation set. In particular, try changing the choice of perceptron algorithm and the maximum number of iterations. Report the validation error for several of the configurations that you tried; which configuration works best?

```
best_val_error = 1
for model in [perceptron_train, average_perceptron]:
    for n in [1,2,5,10,20,50,100]:
        w,k,iters = model(train_featurized,train_labels,n)
        error = perceptron_test(w,val_featurized,val_labels)
        print("Model: {} | Num of iters: {} | Val err: {}".format(model.__name__, iters, error))

        if error < best_val_error:
            best_val_error = error
            best_model = {"model":model, "iters":iters}

print("\nBest model = {} and num_iters = {}".format(best_model["model"].__name__, best_model["iters"]))
```

Model: perceptron_train — Num of iters: 1 — Val err: 0.03
Model: perceptron_train — Num of iters: 2 — Val err: 0.023
Model: perceptron_train — Num of iters: 5 — Val err: 0.023
Model: perceptron_train — Num of iters: 10 — Val err: 0.013
Model: perceptron_train — Num of iters: 20 — Val err: 0.013
Model: perceptron_train — Num of iters: 50 — Val err: 0.013
Model: perceptron_train — Num of iters: 100 — Val err: 0.013
Model: average_perceptron — Num of iters: 1 — Val err: 0.023
Model: average_perceptron — Num of iters: 2 — Val err: 0.02
Model: average_perceptron — Num of iters: 5 — Val err: 0.016
Model: average_perceptron — Num of iters: 10 — Val err: 0.017
Model: average_perceptron — Num of iters: 20 — Val err: 0.012
Model: average_perceptron — Num of iters: 50 — Val err: 0.012
Model: average_perceptron — Num of iters: 100 — Val err: 0.013

Best model = average_perceptron and num_iters = 20

10 **(Optional)** You could additionally change X from question 2.

This is an exploratory question. You can try changing the way you featurize the dataset by using counts of words instead or 0 and 1s, or try a TF-IDF vectorization strategy.

11 **(3 points)** You are ready to train on the full training set, and see if it works on completely new data. Combine the training set and the validation set (i.e. use all of `spam_train.txt`) and learn using the best of the configurations previously found. What is the error on the test set (i.e., now you finally use `spam_test.txt`)?

```python
combined_train_data = np.concatenate((train_featurized,val_featurized))
combined_train_labels = np.concatenate((train_labels,val_labels))
test_data, test_labels = read_data("spam_test.txt")
test_featurized = featurize_data(test_data, word_to_index)

model = best_model["model"]
num_iters = best_model["iters"]
w,k,iters = model(combined_train_data,combined_train_labels,num_iters)
test_err = perceptron_test(w,test_featurized,test_labels)
print("Test error = {}".format(test_err))
```

Test error $= 0.02$