Machine Learning and Computational Statistics, Spring 2015
Homework 3: SVM and Sentiment Analysis

Qingxian Lai(ql516)

Due: Monday, February 23, 2015, at 4pm (Submit via NYU Classes)

Instructions: Your answers to the questions below, including plots and mathematical work, should be submitted as a single PDF file. You may include your code inline or submit it as a separate file. You may either scan hand-written work or, preferably, write your answers using software that typesets mathematics (e.g. LaTeX, LyX, or MathJax via iPython).

1 Introduction

2 The Data

```
def shuffle_data():
   pos_path = "./data/pos"
   neg_path = "./data/neg"

   pos_review = folder_list(pos_path,1)
   neg_review = folder_list(neg_path,-1)

   review = pos_review + neg_review
   random.shuffle(review)

   train_data = review[:1500]
   test_data = review[1500:]

   train_output = open("train_data","a+")
   pickle.dump(train_data,train_output)
   train_output.close()

   test_output = open("test_data","a+")
   pickle.dump(test_data,test_output)
   test_output.close()
```

3 Sparse Representations

1. Write a function that converts an example (e.g. a list of words) into a sparse bag-of-words representation. You may find Python's Counter class to be useful here: `https://docs.python.org/2/library/collections.html`. Note that a Counter is also a dict.

```
def convert_to_word_bag(words_list):
   a = {}
   for words in words_list[:-1]:
      a[words] = a.get(words,0)+1
   return a
```

2. Write a version of `generic_gradient_checker` from Homework 1 that works with sparse vectors represented as dict types. See Homework 1 solutions if you didn't do that part. Since we'll be using it for stochastic methods, it should take a single (x, y) pair, rather than the entire dataset. Be sure to use the dotProduct and increment primitives we provide, or make your own.

```
def
    generic_gradient_checker(X,y,w,lamb,grad_fun,loss_fun,epsilon=0.01,tolerance=0.1):
   true_gradient = grad_fun(X,y,w,lamb)
   approx_grad = {}
   distance = 0
   for key in w.keys():
      test_w_p = w.copy()
      test_w_n = w.copy()
      test_w_p[key] += epsilon
      test_w_n[key] -= epsilon
      approx_grad[key] =
          (loss_fun(X,y,test_w_p,lamb)-loss_fun(X,y,test_w_n,lamb))/(2*epsilon)
      distance +=
          (true_gradient[key]-approx_grad[key])*(true_gradient[key]-approx_grad[key])
   distance = np.sqrt(distance)
   return distance <= tolerance
```

4 Support Vector Machine via Pegasos

1. [Written] Compute a subgradient for the "stochastic" SVM objective, which assumes a single training point. Show that if your step size rule is $\eta_t = 1/(\lambda t)$, then the the corresponding SGD update is the same as given in the pseudocode.
Answer: The SGD object function is (for pair(X_i, y_i)):

$$f(w) = \frac{\lambda}{2}\|w\|^2 + max\{0, 1 - y_i w^T X_i\}$$

This function is not differentiable, so I calculate its subgradient:
if $1 - y_i w^T X_i > 0$: $\frac{df(w)}{dw} = \lambda w + y_i X_i^T$

if $1 - y_i w^T X_i < 0$: $\frac{df(w)}{dw} = \lambda w$

if $1 - y_i w^T X_i = 0$: $\frac{df(w)}{dw} = [\lambda w,\ \lambda w + y_i X_i^T]$

so, with formula $w_{t+1} = w_t - \eta df(w)$, we have:

if $y_i w^T X_i < 1$: $\quad w_{t+1} = w_t - \eta_t(\lambda w + y_i X_i^T) = (1 - \eta_t \lambda)w - \eta_t y_i X_i^T$

if $1 - y_i w^T X_i \geq 1$: $w_{t+1} = w_t - \eta_t \lambda w = (1 - \eta_t \lambda) w$
these are the same as given in the peseudocode

2. Implement the Pegasos algorithm to run on a sparse data representation. The output should be a sparse weight vector w. [As should be your habit, please check your gradient using `generic_gradient_checker` while you are in the testing phase. That will be our first question if you ask for help debugging. Once you're convinced it works, take it out so it doesn't slow down your code.]

```
def pegasos_SGD(X,y,lamb,num_iter):
   w = {}
   t = 1
   s = 1
   for i in range(num_iter):

      for j in range(len(X)):

         t += 1
         alpha = 1.0/(t*lamb)
         tmp = y[j] * s * dotProduct(X[j], w)
         g = l_de(tmp)
         s *= (1 - alpha * lamb)
         w = increment(w, -(alpha*y[j]*g/s), X[j])


         # if tmp < 1:
         #    tmp = increment({},(1-alpha*lamb),w)
         #    w = increment(tmp,alpha*y[j],X[j])
         # else:
         #    w = increment({},1-alpha*lamb,w)
         # w = increment(w,-alpha,pegasos_grad(X[j],y[j],w,1))
         # print
             generic_gradient_checker(X[j],y[j],w,1,pegasos_grad,pegasos_loss)
      # loss.append(percent_error(X,y,w))
   return increment({},s,w)
```

3. Write a function that takes the sparse weight vector w and a collection of (x, y) pairs, and returns the percent error when predicting y using $\text{sign}(w^T x)$ (that is, report the 0-1 loss).

```
def percent_error(X,y,w):
```

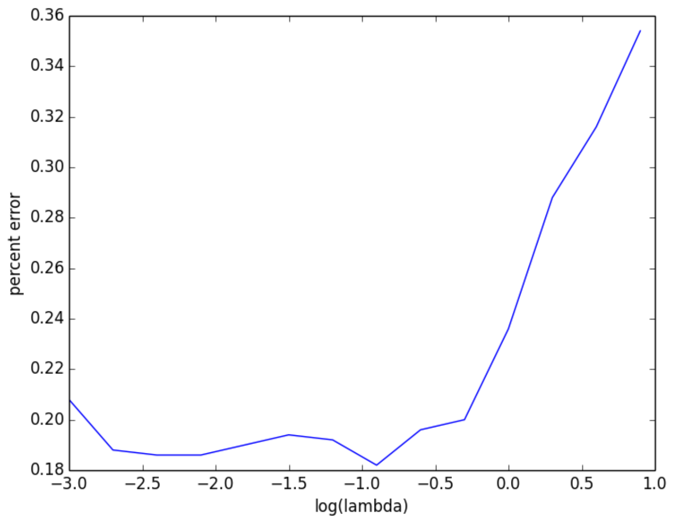

Figure 1: Percent error with different Lambda

```
correct = 0
total = len(y)
for i in range(len(y)):
   sign_value = np.sign(dotProduct(X[i],w))
   if y[i] == sign_value:
      correct += 1
return 1-float(correct)/total
```

4. Using the bag-of-words feature representation described above, search for the regularization parameter that gives the minimal percent error on your test set. A good search strategy is to start with a set of lambdas spanning a broad range of orders of magnitude. Then, continue to zoom in until you're convinced that additional search will not significantly improve your test performance. Once you have a sense of the general range of regularization parameters that give good results, you do not have to search over orders of magnitude every time you change something (such as adding new feature).

   Answer: As Figure 1 shows, the best lambda for this problem is about 0.1

5. Recall that the "score" is the value of the prediction $f(x) = w^T x$. We like to think that the magnitude of the score represents the confidence of the prediction. This is something we can directly verify or refute. Break the predictions into groups based on the score (you can play with the size of the groups to get a result you think is informative). For each group, examine the percentage error. You can make a table or graph. Summarize the results. Is there a correlation between higher magnitude scores and accuracy?
   Answer: As figure 2 shows, when the absolute value of the "score" become larger, the

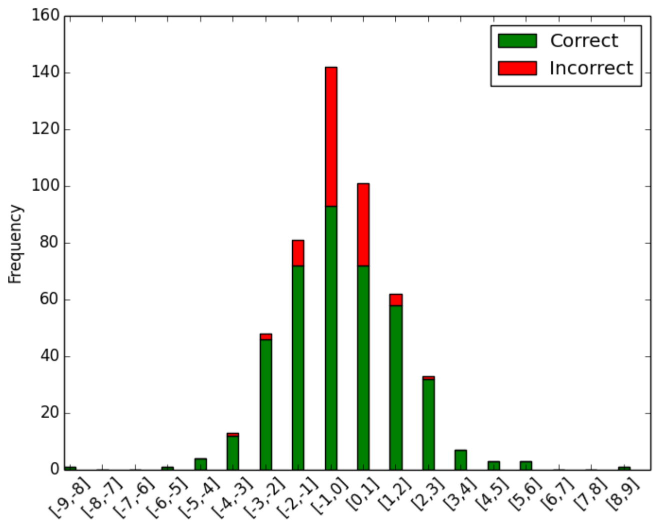

Figure 2: Number of correct predictions and incorrection predictions

ratio of correct prediction and total instance become larger, which means the increse of the confidence of prediction.

5 Error Analysis

The natural language processing domain is particularly nice in that one can often interpret why a model has performed well or poorly on a specific example, and sometimes it is not very difficult to come up with ideas for new features that might help fix a problem. The first step in this process is to look closely at the errors that our model makes.

1. Choose some examples that the model got wrong. List the features that contributed most heavily to the descision (e.g. rank them by $|w_i x_i|$), along with x_i, w_i, xw_i. Do you understand why the model was incorrect? Can you think of a new feature that might be able to fix the issue? Include a short analysis for at least 3 incorrect examples.
   Answer:
   Example 1:

```
sign of predicted score:  -1
true vote:  1
wx:
[('and', 0.9439790226883807), ('the', 0.6222083953689858),
("didn't", -0.5213217484055894), ('to', -0.4424346125641638),
('this', -0.4026577187173579), ('of', -0.3082153729917135),
 ('back', 0.28466034088131253), ('fun', 0.2733272593942343)]
```

```
abs_wx
[('and', 0.9439790226883807), ('the', 0.6222083953689858),
 ("didn't", 0.5213217484055894), ('to', 0.4424346125641638),
 ('this', 0.4026577187173579), ('of', 0.3082153729917135),
  ('back', 0.28466034088131253), ('fun', 0.2733272593942343)]

x
[18, 50, 6, 11, 12, 19, 3, 2]

w
[0.052443279038243376, 0.012444167907379716,
-0.08688695806759823, -0.04022132841492398,
-0.03355480989311316, -0.01622186173640597,
0.09488678029377084, 0.13666362969711715]
```

Example 2:

```
sign of predicted score:  1
true vote:  -1
wx:
[('to', -0.7239839114686316), ('and', 0.6817626274971639),
('the', 0.5599875558320871), ('will', 0.35376991622408277),
('you', 0.3533254816559622), ('on', -0.3437701384413655),
 ('not', 0.33599253349925406), ('is', 0.32665940756872)]

abs_wx
[('to', 0.7239839114686316), ('and', 0.6817626274971639),
('the', 0.5599875558320871), ('will', 0.35376991622408277),
('you', 0.3533254816559622), ('on', 0.3437701384413655),
 ('not', 0.33599253349925406), ('is', 0.32665940756872)]

x
[18, 13, 45, 4, 6, 7, 12, 21]

w
[-0.04022132841492398, 0.052443279038243376,
 0.012444167907379716, 0.08844247905602069,
 0.05888758027599371, -0.04911001977733793,
 0.027999377791604507, 0.01555520988422476]
```

Example 3:

```
sign of predictedscore:  1
true vote:  -1
```

```
wx:
[('and', 0.9964223017266242), ('the', 0.348436701406632),
('as', 0.31732628163818355), ('on', -0.2946601186640276),
('to', -0.24132797048954385), ('have', -0.21332859269793583),
('of', -0.21088420257327764), ('back', 0.1897735605875417)]

abs_wx
[('and', 0.9964223017266242), ('the', 0.348436701406632),
('as', 0.31732628163818355), ('on', 0.2946601186640276),
 ('to', 0.24132797048954385), ('have', 0.21332859269793583),
 ('of', 0.21088420257327764), ('back', 0.1897735605875417)]

x
[19, 28, 12, 6, 6, 2, 13, 2]

w
[0.052443279038243376, 0.012444167907379716,
0.026443856803181964, -0.04911001977733793,
-0.04022132841492398, -0.10666429634896792,
-0.01622186173640597, 0.09488678029377084]
```

In thses three examples, we can find that the most important features are the words that cannot show the reviewers' opinion, such as "and", "the", "of", "to". These words present in both positive reviews and negative reviews. So if these neutral words play an important role in predicting the score, the model would probably be wrong. And a good way to fix this bug is to eliminate the common words in both postive reviews and negative reviews

6 Features

For a problem like this, the features you use are far more important than the learning model you choose. Whenever you enter a new problem domain, one of your first orders of business is to beg, borrow, or steal the best features you can find. This means looking at any relevant published work and seeing what they've used. Maybe it means asking a colleague what features they use. But eventually you'll need to engineer new features that help in your particular situation. To get ideas for this dataset, you might check the discussion board on this Kaggle competition, which is using a very similar dataset `https://www.kaggle.com/c/sentiment-analysis-on-movie-reviews`. There are also a very large number of academic research papers on sentiment analysis that you can look at for ideas.

1. Based on your error analysis, or on some idea you have, find a new feature (or group of features) that improve your test performance. Describe the features and what kind of improvement they give. At this point, it's important to consider the standard errors ($\sqrt{p(1-p)/n}$) on your performance estimates, to know whether the improvement is statistically significant.
   Answer: I begin with a percent error of **20 %** and try several methods to improve the performance:

(a) First I tried to eliminate some meanless words, such as "and", "to", "the", "you", "on", "will", "is", "of", "have", "as", "back", "such", "been", "why", "not", "his", "an", "it's", "that", "most", "this", "a", "be", "then". But the percent error didn't change at all and keep fluctuating around **20 %**

(b) Then I tried only delete some of the words above and finaly choosed a word set: "and", "to", "the", "is", "of", "have", "his", "an", "it's", "that", "this", "a", "be", "then". Then I got a percent error of **19 %**

(c) Next, I added new features by combining the words after "not" with "not", and then eliminating the original words. However, this does no help for the performance.

(d) Keep some of the special characters, such as ().,:;+-*/—¡¿= ", these characters are usually used in some emojis. Then the percent decrease to **16 %**, this is the best performnce i achieved.

2. [Optional] Try to get the best performance possible by generating lots of new features, changing the pre-processing, or any other method you want, so long as you are using the same core SVM model. Describe what you tried, and how much improvement each thing brought to the model. To get you thinking on features, here are some basic ideas of varying quality: 1) how many words are in the review? 2) How many "negative" words are there? (You'd have to construct or find a list of negative words.) 3) Word n-gram features: Instead of single-word features, you can make every pair of consecutive words a feature. 4) Character n-gram features: Ignore word boundaries and make every sequence of n characters into a feature (this will be a lot). 5) Adding an extra feature whenever a word is preceded by "not". For example "not amazing" becomes its own feature. 6) Do we really need to eliminate those funny characters in the data loading phase? Might there be useful signal there? 7) Use tf-idf instead of raw word counts. The tf-idf is calculated as

$$\text{tfidf}(f_i) = \frac{FF_i}{\log(DF_i)} \tag{1}$$

where FF_i is the feature frequency of feature f_i and DF_i is the number of document containing f_i. In this way we increase the weight of rare words. Sometimes this scheme helps, sometimes it makes things worse. You could try using both! [Extra credit points will be awarded in proportion to how much improvement you achieve.]

7 Feedback (not graded)

1. Approximately how long did it take to complete this assignment?
2. Did you find the Python programming challenging (in particular, converting your code to use sparse representations)?
3. Any other feedback?