DS-GA 1003: Machine Learning and Computational Statistics
Homework 3: SVM and Sentiment Analysis

February 23, 2017

Yidi Zhang

Instructions: Your answers to the questions below, including plots and mathematical work, should be submitted as a single PDF file. It's preferred that you write your answers using software that typesets mathematics (e.g. LaTeX, LyX, or MathJax via iPython), though if you need to you may scan handwritten work. You may find the minted package convenient for including source code in your LaTeX document.

1 Introduction

In this assignment, we'll be working with natural language data. In particular, we'll be doing sentiment analysis on movie reviews. This problem will give you the opportunity to try your hand at feature engineering, which is one of the most important parts of many data science problems. From a technical standpoint, this homework has two new pieces. First, you'll be implementing Pegasos. Pegasos is essentially stochastic subgradient descent for the SVM with a particular schedule for the step-size. Second, because in natural langauge domains we typically have huge feature spaces, we work with sparse representations of feature vectors, where only the non-zero entries are explicitly recorded. This will require coding your gradient and SGD code using hash tables (dictionaries in Python), rather than numpy arrays. We begin with some practice with subgradients and an easy problem that introduces the Perceptron algorithm.

2 Calculating Subgradients

Recall that a vector $g \in \mathbf{R}^d$ is a **subgradient** of $f : \mathbf{R}^d \to \mathbf{R}$ at x if for all z,

$$f(z) \geq f(x) + g^T(z - x).$$

As we noted in lecture, there may be 0, 1, or infinitely many subgradients at any point. The **subdifferential** of f at a point x, denoted $\partial f(x)$, is the set of all subgradients of f at x.

Just as there is a calculus for gradients, there is a calculus for subgradients[1]. For our purposes, we can usually get by using the definition of subgradient directly. However, in the first problem below we derive a property that will make our life easier for finding a subgradient of the hinge loss and perceptron loss.

[1] A good reference for subgradients are the course notes on Subgradients by Boyd et al.

1. [Subgradients for pointwise maximum of functions] Suppose $f_1, \ldots, f_m : \mathbf{R}^d \to \mathbf{R}$ are convex functions, and

$$f(x) = \max_{i=1,\ldots,m} f_i(x).$$

Let k be any index for which $f_k(x) = f(x)$, and choose $g \in \partial f_k(x)$. [We are using the fact that a convex function on $\mathbf{R}^d$ has a non-empty subdifferential at all points.] Show that $g \in \partial f(x)$.

Answer 2.1:

For $\forall a \in R^d$

$$f(a) = max f_i(x) \geq f_k(a) \geq f_k + g^T(a - x) = f(x) + g^T(a - x)$$

Thus we can get $f(a) \geq f(x) + g^T(a - x)$, for $\forall a \in R^d$. So we can conclude that $g \in \partial f(x)$.

2. [Subgradient of hinge loss for linear prediction] Give a subgradient of

$$J(w) = \max\left\{0, 1 - yw^T x\right\}.$$

Answer 2.2:

Suppose $g(w) = w^T x$, and $f(z) = max\{0, 1 - yz\}$.

Using the chain rule, we get:

$$\frac{\partial f(g(w))}{\partial w_i} = \frac{\partial f}{\partial z}\frac{\partial g}{\partial w_i}$$

$$g^{'}(w) = \begin{cases} -y & w^T x < 1 \\ 0 & otherwise \end{cases}$$

Thus we can get

$$\begin{aligned} \frac{\partial f(g(w))}{\partial w_i} &= \begin{cases} -yx_i & w^T x < 1 \\ 0 & otherwise \end{cases} \\ \frac{\partial J(w)}{\partial w} &= \sum_i \frac{\partial J(w)}{\partial w_i} \\ &= \sum_i \frac{\partial f(g(w))}{\partial w_i} \end{aligned}$$

3 Perceptron

The perceptron algorithm is often the first classification algorithm taught in machine learning classes. Suppose we have a labeled training set $(x_1, y_1), \ldots, (x_n, y_n) \in \mathbf{R}^d \times \{-1, 1\}$. In the perceptron algorithm, we are looking for a hyperplane that perfectly separates the classes. That is, we're looking for $w \in \mathbf{R}^d$ such that

$$y_i w^T x_i > 0 \; \forall i \in \{1, \ldots, n\}.$$

Visually, this would mean that all the x's with label $y = 1$ are on one side of the hyperplane $\left\{x \mid w^T x = 0\right\}$, and all the $x's$ with label $y = -1$ are on the other side. When such a hyperplane exists, we say that the data are **linearly separable**. The perceptron algorithm is given in Algorithm 1.

Algorithm 1: Perceptron Algorithm

input: Training set $(x_1, y_1), \ldots, (x_n, y_n) \in \mathbf{R}^d \times \{-1, 1\}$
$w^{(0)} = (0, \ldots, 0) \in \mathbf{R}^d$
$k = 0$ # step number
repeat
  all_correct = TRUE
  for $i = 1, 2, \ldots, n$ # loop through data
    if $(y_i x_i^T w^{(k)} \leq 0)$
      $w^{(k+1)} = w^{(k)} + y_i x_i$
      all_correct = FALSE
    else
      $w^{(k+1)} = w^{(k)}$
    end if
    $k = k + 1$
  end for
until (all_correct == TRUE)
return $w^{(k)}$

There is also something called the **perceptron loss,** given by

$$\ell(\hat{y}, y) = \max\{0, -\hat{y}y\}.$$

In this problem we will see why this loss function has this name.

1. Show that if $\left\{x \mid w^T x = 0\right\}$ is a separating hyperplane for a training set $\mathcal{D} = ((x_1, y_1), \ldots, (x_n, y_n))$, then the average perceptron loss on $\mathcal{D}$ is 0.

   Answer 3.1:

   if $\left\{x \mid w^T x = 0\right\}$ is a separating hyperplane, for $\forall i$, $max(0, -y_i w^T x_i) = 0$.

   Thus we can conclude that $\ell(\hat{y}, y) = \frac{1}{n} \sum_{i=1}^{n} \max\{0, -\hat{y_i} y_i\} = 0$.

2. Let $\mathcal{H}$ be the linear hypothesis space consisting of functions $x \mapsto w^T x$. Consider running stochastic subgradient descent (SSGD) to minimize the empirical risk with the perceptron loss. We'll use the version of SSGD in which we cycle through the data points in each epoch. Show that if we use a fixed step size 1, we terminate when our training loss is 0, and we make the right choice of subgradient, then we are exactly doing the Perceptron algorithm.

Answer 3.2:

$$J(w) = \frac{1}{n}\sum_{i}^{n} \max(0, y_i w^T x_i)$$
$$J_i(w) = \max(0, y_i w^T x_i)$$

Then we can get

$$\frac{\partial J_i}{\partial w_j} = \begin{cases} 0 & if y_i w^T x_i > 0 \\ -y_i x_{ij} & otherwise \end{cases}$$

Then we get the gradient of the objective function:

$$\nabla J_i = \begin{cases} 0 & if y_i w^T x_i > 0 \\ -y_i x_i & otherwise \end{cases}$$

So

$$w^{(k+1)} = w^{(k)} - \alpha \sum_{i=1}^{n} \nabla J_i$$

If training loss is 0, and we make the right choice of subgradient, then we can get

$$w^T x^{(i)} = 0$$
$$and\ 0 \in \partial L$$

Thus $w^{(k+1)} = w^{(k)}$, and we are exactly doing the Perceptron algorithm.

3. Suppose the perceptron algorithm returns w. Show that w is a linear combination of the input points. That is, we can write $w = \sum_{i=1}^{n} \alpha_i x_i$ for some $\alpha_1, \ldots, \alpha_n \in \mathbf{R}$. The x_i for which $\alpha_i \neq 0$ are called support vectors. Give a characterization of points that are support vectors and not support vectors.

Answer 3.3:

$$w^{(k+1)} = \begin{cases} w^{(k)} + y_i x_i & if y_i w^{(k)} x_i^T \leq 0 \\ w^{(k)} & otherwise \end{cases}$$

Thus the combination of w can only be 0 and $y_i x_i$, which means that w is a linear combination of the input points.

The points that are support vectors are the points which are wrongly classified, while the points that are not support vectors are the points that are classified correctly.

4 The Data

We will be using the Polarity Dataset v2.0, constructed by Pang and Lee. It has the full text from 2000 movies reivews: 1000 reviews are classified as "positive" and 1000 as "negative." Our goal is to predict whether a review has positive or negative sentiment from the text of the review. Each review is stored in a separate file: the positive reviews are in a folder called "pos", and the negative reviews are in "neg". We have provided some code in `load.py` to assist with reading these files. You can use the code, or write your own version. The code removes some special symbols from the reviews. Later you can check if this helps or hurts your results.

1. Load all the data and randomly split it into 1500 training examples and 500 validation examples.

   Answer 4.1:

```
review = pos_review + neg_review
    random.seed(0)
    random.shuffle(review)
    pickle.dump(review, open("/Users/twff/Downloads/machine_learning/hw
    /hw3-sentiment/data/allreview.p", "wb" ) )
```

```
reviews = pickle.load(open("/Users/twff/Downloads/machine_learning/hw
/hw3-sentiment/data/allreview.p", "rb" ) )
training_data = reviews[0:1500]
validation_data = reviews[1500:2000]
len(reviews)
```

5 Sparse Representations

The most basic way to represent text documents for machine learning is with a "bag-of-words" representation. Here every possible word is a feature, and the value of a word feature is the number of times that word appears in the document. Of course, most words will not appear in any particular document, and those counts will be zero. Rather than store a huge number of zeros, we use a sparse representation, in which we only store the counts that are nonzero. The counts are stored in a key/value store (such as a dictionary in Python). For example, "Harry Potter and Harry Potter II" would be represented as the following Python dict: x={'Harry':2, 'Potter':2, 'and':1, 'II':1}. We will be using linear classifiers of the form $f(x) = w^T x$, and we can store the w vector in a sparse format as well, such as w={'minimal':1.3,'Harry':-1.1,'viable':-4.2,'and':2.2,'product':9.1}. The inner product between w and x would only involve the features that appear in both x and w, since whatever doesn't appear is assumed to be zero. For this example, the inner product would be x[Harry] * w[Harry] + x[and] * w[and] = 2*(-1.1) + 1*(2.2). To help you along, we've included two functions for working with sparse vectors: 1) a dot product between two vectors represented as dict's and 2) a function that increments one sparse vector by a scaled multiple of another vector, which is a very common operation. These functions are located in `util.py`. It is worth reading the code, even if you intend to implement it yourself. You may get some ideas on how to make things faster.

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.

Answer 5.1:

```
from collections import Counter
from nltk.corpus import stopwords
import nltk

def bag_of_words(words):
    '''
    words is a list of words
    -------------------
    count sparse bag of words representation.
    '''
    stop = set(stopwords.words('english'))
    words_stopfree = [w for w in words if w not in stop]
    count = Counter(words_stopfree[:-1])
    label = words[-1]
    return count, label
```

The results are shown as below

```
words=['hello','world','Hello','world','it','is','a','new','start', 1]
words, l = bag_of_words(words)
words
Counter({'Hello': 1, 'hello': 1, 'new': 1, 'start': 1, 'world': 2})
```

2. [Optional] 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. **Note**: SVM loss is not differentiable everywhere. Yet our method for checking the gradient doesn't extent to subgradients. Thus in certain situations, the gradient checker may indicate that the gradient is not correct, when in fact you have provided a perfectly fine subgradient. This is an interesting opportunity to see how often we actually end up in those places.

6 Support Vector Machine via Pegasos

In this question you will build an SVM using the Pegasos algorithm. To align with the notation used in the Pegasos paper[2], we're considering the following formulation of the SVM objective function:

$$\min_{w \in \mathbf{R}^n} \frac{\lambda}{2}\|w\|^2 + \frac{1}{m}\sum_{i=1}^{m} \max\left\{0, 1 - y_i w^T x_i\right\}.$$

Note that, for simplicity, we are leaving off the unregularized bias term b, and the expression with "max" is just another way to write $\left(1 - y_i w^T x\right)_+$. Pegasos is stochastic subgradient descent using a step size rule $\eta_t = 1/(\lambda t)$. The pseudocode is given below:

Input: $\lambda > 0$. Choose $w_1 = 0, t = 0$
While termination condition not met
  For $j = 1, \ldots, m$ (assumes data is randomly permuted)
    $t = t + 1$
    $\eta_t = 1/(t\lambda)$;
    If $y_j w_t^T x_j < 1$
      $w_{t+1} = (1 - \eta_t \lambda) w_t + \eta_t y_j x_j$
    Else
      $w_{t+1} = (1 - \eta_t \lambda) w_t$

---

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. [You may use the following facts without proof: 1) If $f_1, \ldots, f_m : \mathbf{R}^d \to \mathbf{R}$ are convex functions and $f = f_1 + \cdots + f_m$, then $\partial f(x) = \partial f_1(x) + \cdots + \partial f_m(x)$. 2) For $\alpha \geq 0$, $\partial(\alpha f)(x) = \alpha \partial f(x)$.]

   Answer 6.1:

   The subgradient for the SVM objective is

$$\partial f = \begin{cases} \lambda w_t - y_{it} x_{it} & if\, y_{it} w_t x_{it} < 1 \\ \lambda w_t & otherwise \end{cases}$$

   We know

$$w_{t+1} := w_t - \eta_t \partial f$$
$$\eta_t = \frac{1}{\lambda t}$$

   So now the update can be written as:

[2] Shalev-Shwartz et al.'s "Pegasos: Primal Estimated sub-GrAdient SOlver for SVM".

If $y_i w^T x_i < 1$

$$\begin{aligned} w_{t+1} &:= w_t - \eta_t \partial f \\ &= w_t - \eta_t(\lambda w_t - y_i x_i) \\ &= (1 - \eta_t) w_t + \eta_t y_i x_i) \end{aligned}$$

If $y_i w^T x_i \geq 1$

$$w_{t+1} := (1 - \eta_t) w_t$$

2. Implement the Pegasos algorithm to run on a sparse data representation. The output should be a sparse weight vector w. Note that our Pegasos algorithm starts at $w = 0$. In a sparse representation, this corresponds to an empty dictionary. **Note:** With this problem, you will need to take some care to code things efficiently. In particular, be aware that making copies of the weight dictionary can slow down your code significantly. If you want to make a copy of your weights (e.g. for checking for convergence), make sure you don't do this more than once per epoch.

Answer 6.2:

```
def pegasos_svm(X, y, Lambda, num_iter):
    '''
    X is input sparse data representation
    y is pos and neg label of the review
    Lambda is parameter of step size
    num_iter is number of epoch
    ----------------------------
    w is sparse weight vector
    '''
    w = {}
    t = 0.0

    for i in range(num_iter):
        start = timeit.default_timer()
        for j in range(len(X)):
            #w_old = w.copy()
            t = t + 1
            alpha = 1 / (t * Lambda)
            if (y[j] * dotProduct(w, X[j])) < 1:
                #for k, v in w.items():
                 #   w[k] = w.get(k, 0) + v * (1-alpha * Lambda)
                increment(w, -alpha * Lambda, w)
                increment(w, alpha*y[j], X[j])
            else:
                increment(w, -alpha * Lambda, w)
        stop = timeit.default_timer()
        print('The time of the {} epoch is {}'.format(i+1,(stop-start)))
    return w
```

3. Note that in every step of the Pegasos algorithm, we rescale every entry of w_t by the factor $(1-\eta_t\lambda)$. Implementing this directly with dictionaries is relatively slow. We can make things significantly faster by representing w as $w = sW$, where $s \in \mathbf{R}$ and $W \in \mathbf{R}^d$. You can start with $s = 1$ and W all zeros (i.e. an empty dictionary). Note that both updates start with rescaling w_t, which we can do simply by setting $s_{t+1} = (1-\eta_t\lambda)\, s_t$. If the update is $w_{t+1} = (1-\eta_t\lambda)w_t + \eta_t y_j x_j$, then **verify that the Pegasos update step is equivalent to**:

$$\begin{aligned} s_{t+1} &= (1-\eta_t\lambda)\, s_t \\ W_{t+1} &= W_t + \frac{1}{s_{t+1}}\eta_t y_j x_j. \end{aligned}$$

There is one subtle issue with the approach described above: if we ever have $1-\eta_t\lambda = 0$, then $s_{t+1} = 0$, and we'll have a divide by 0 in the calculation for W_{t+1}. This only happens when $\eta_t = 1/\lambda$. With our step-size rule of $\eta_t = 1/(\lambda t)$, it happens exactly when $t = 1$. So one approach is to just start at $t = 2$. More generically, note that if $s_{t+1} = 0$, then $w_{t+1} = 0$. Thus an equivalent representation is $s_{t+1} = 1$ and $W = 0$. Thus if we ever get $s_{t+1} = 0$, simply set it back to 1 and reset W_{t+1} to zero, which is an empty dictionary in a sparse representation. **Implement the Pegasos algorithm with this change**. [See section 5.1 of Leon Bottou's Stochastic Gradient Tricks for a more generic version of this technique, and many other useful tricks.]

Answer 6.3:

If the update is $w_{t+1} = (1-\eta_t\lambda)w_t + \eta_t y_j x_j$,

$$\begin{aligned} &w_{t+1} = (1-\eta_t\lambda)w_t + \eta_t y_j x_j \\ &s_{t+1}W_{t+1} = (1-\eta_t\lambda)W_t s_t + \eta_t y_j x_j) \\ &W_{t+1} = \frac{(1-\eta_t\lambda)s_t}{s_{t+1}}W_t + \frac{\eta_t y_j x_j}{s_{t+1}} \\ &W_{t+1} = W_t + \frac{1}{s_{t+1}}\eta_t y_j x_j \end{aligned}$$

Thus the Pegasos update step is equivalent to

$$\begin{aligned} s_{t+1} &= (1-\eta_t\lambda)\, s_t \\ W_{t+1} &= W_t + \frac{1}{s_{t+1}}\eta_t y_j x_j. \end{aligned}$$

And the code shows as follow:

```
def pegasos_svm_change(X, y, Lambda, num_iter):
    '''
    X is input sparse data representation
    y is pos and neg label of the review
    Lambda is parameter of step size
```

```
num_iter is number of epoch
----------------------------
w is sparse weight vector
'''
s =1
W = {}
t = 0.0
w = {}
#print(type(W))

for i in range(num_iter):
    start = timeit.default_timer()
    for j in range(len(X)):
        t = t + 1
        alpha = 1 / (t * Lambda)
        s = (1 - Lambda * alpha)*s
        if s == 0:
            s = 1
            W = {}
        if y[j] * s * dotProduct(W, X[j]) < 1:
            increment(W, (1/s)*alpha*y[j], X[j])
        else:
            W = W
    stop = timeit.default_timer()
    print('The time of the {} epoch is {}'.format(i+1,(stop-start)))
print(s)
increment(w,s,W)
return w
```

4. Run both implementations of Pegasos to train an SVM on the training data (using the bag-of-words feature representation described above). Make sure your implementations are correct by verifying that the two approaches give essentially the same result. Report on the time taken to run each approach.

Answer 6.4:

After running both implementations of Pegasos on training data, I get the essentially the same result. And the time taken to run each method is:

Method of 6.2:

```
The time of the 1 epoch is 16.7350112109998
The time of the 2 epoch is 26.042331717995694
The time of the 3 epoch is 24.68780592099938
The time of the 4 epoch is 24.562127457000315
The time of the 5 epoch is 24.870430830997066
The time of the 6 epoch is 24.874173132993747
The time of the 7 epoch is 24.774511045994586
The time of the 8 epoch is 24.73381142700964
The time of the 9 epoch is 24.740555381999002
The time of the 10 epoch is 24.77694539199001
The time of the 11 epoch is 24.742912389003322
The time of the 12 epoch is 25.24772353300068
The time of the 13 epoch is 29.86519856999803
The time of the 14 epoch is 25.101484796003206
The time of the 15 epoch is 26.183572648995323
The time of the 16 epoch is 26.148424128987244
The time of the 17 epoch is 27.446792246002587
The time of the 18 epoch is 27.99856451099913
The time of the 19 epoch is 29.490715571999317
The time of the 20 epoch is 26.211274907007464
```

Method of 6.3 *AfterChange*

```
The time of the 1 epoch is 0.3885569639969617
The time of the 2 epoch is 0.37763888500921894
The time of the 3 epoch is 0.43433662499592174
The time of the 4 epoch is 0.36520389399083797
The time of the 5 epoch is 0.3710629899869673
The time of the 6 epoch is 0.3578811479965225
The time of the 7 epoch is 0.3565807289996883
The time of the 8 epoch is 0.39485841400164645
The time of the 9 epoch is 0.3906846549944021
The time of the 10 epoch is 0.3692717780068051
The time of the 11 epoch is 0.4183165559952613
The time of the 12 epoch is 0.38175947799754795
The time of the 13 epoch is 0.3617392349988222
The time of the 14 epoch is 0.3653940619988134
The time of the 15 epoch is 0.41751101799309254
The time of the 16 epoch is 0.392883830005303
The time of the 17 epoch is 0.36920433399791364
```

```
The time of the 18 epoch is 0.3847044510039268
The time of the 19 epoch is 0.3717086569959065
The time of the 20 epoch is 0.39212347198917996
```

5. Write a function that takes a sparse weight vector w and a collection of (x, y) pairs, and returns the percent error when predicting y using $\text{sign}(w^T x)$. In other words, the function reports the 0-1 loss of the linear predictor $x \mapsto w^T x$.

Answer 6.5

```
def percent_error(w, X, y):
    '''
    w is the sparse weight vector
    X is the vector needed to be classified
    y is the true label
    -------------------------
    error is the percent of the wrong classifiation
    '''
    count = 0
    N = len(X)

    for i in range(N):
        yy = dotProduct(w, X[i])
        #print(yy)
        if yy > 0:
            yy = 1
        else:
            yy = -1
        #print(yy)
        if yy !=y[i]:
            count +=1
    #print(count)
    error = count/N
    return error
```

6. 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 regularization parameters 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 6.6

The search process and results are shown as follows:

```
errors = []
for i in range(-5,2):
    Lambda = 10**i;
    w = pegasos_svm_change(X_train, y_train, Lambda=Lambda, num_iter=20)
    errors.append(percent_error(w, X_valid, y_valid))
    print(Lambda, percent_error(w, X_valid, y_valid))

plt.plot(range(-5,2), errors)
plt.xlabel("lambda")
plt.ylabel("error")
plt.savefig('/Users/twff/Downloads/machine_learning/hw/hw3-sentiment/img66')
```

Then we get a figure showing the variation of error with λ, so keep zooming in

```
Lambda = np.linspace(0,1,10)
errors = []
for i in range(len(Lambda)):
    w = pegasos_svm_change(X_train, y_train, Lambda=Lambda[i], num_iter=20)
    errors.append(percent_error(w, X_valid, y_valid))
    print(Lambda[i], percent_error(w, X_valid, y_valid))

plt.plot(Lambda, errors)
plt.xlabel("lambda")
plt.ylabel("error")
plt.savefig('/Users/twff/Downloads/machine_learning/hw/hw3-sentiment/img66b')
```

From the images shown, we can get when $\lambda = 0.1$, we get minimal percent error, which is 0.15.

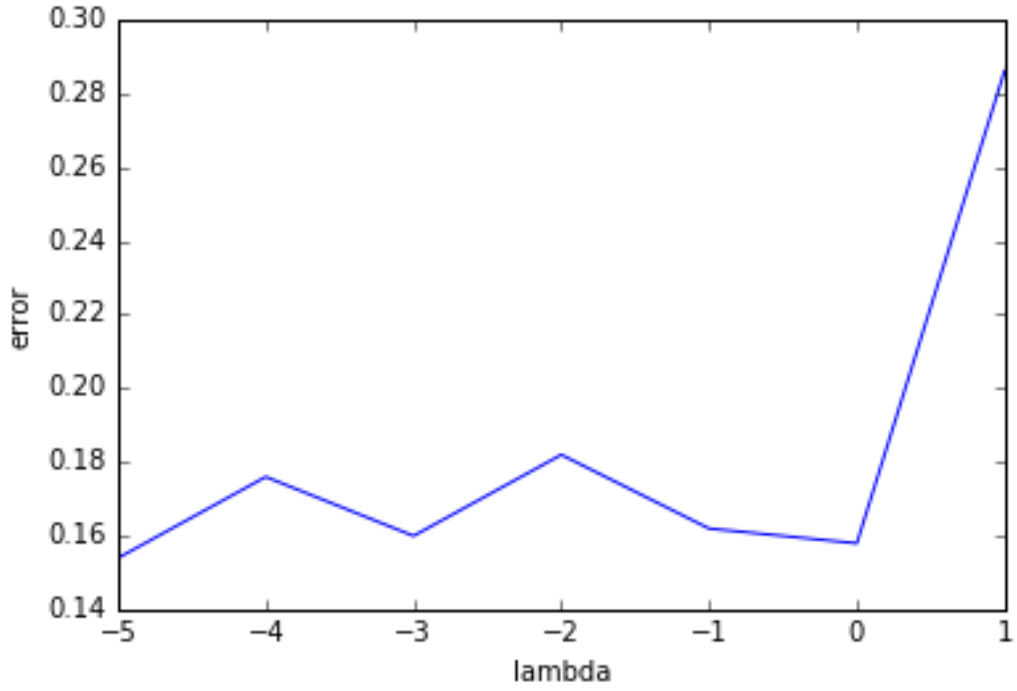

Figure 1: Lambda range giving the minimal percent error

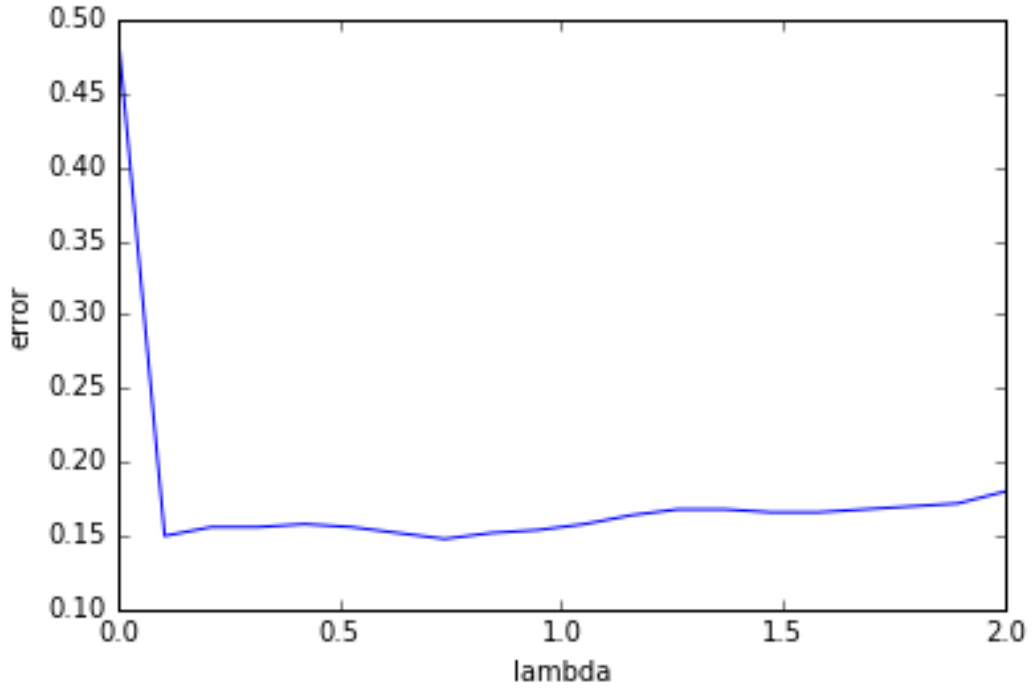

Figure 2: Lambda giving the minimal percent error after zooming in

7. [Optional] 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?

8. [Optional] Our objective is not differentiable when $y_i w^T x_i = 1$. Investigate how often and when we have $y_i w^T x_i = 1$ (or perhaps within a small distance of 1 – this is for you to explore) . Describe your findings. If we didn't know about subgradients, one might suggest just skipping the update when $y w^T x_i = 1$. Does this seem reasonable?

7 Error Analysis

The natural language processing domain is particularly nice in that often 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 2 incorrect examples.

   Answer 7.1

   First of all, we list the features that contributed most heavily to the decision.

```
w_wrong = {}
for k,v in w.items():
    if abs(v)>=0.01:
        w_wrong[k] = w.get(k, 0) + v

w_high = pd.DataFrame(w_wrong,index=range(len(w_wrong))).T
w_high = w_high[0]
w_high
```

   The results are shown as follows:

```
also     0.041859
bad     -0.066707
best     0.029917
film     0.049779
great    0.029275
life     0.055275
love     0.022173
many     0.024715
movie   -0.059741
plot    -0.027611
story    0.022152
well     0.024776
world    0.028056
worst   -0.021296
```

   We get **'also', 'bad', 'best', 'film', 'great', 'life', 'love', 'many', 'movie', 'plot', 'story', 'well', 'world', 'worst'** contribute most to the decision.

   Then we choose several examples that the model got wrong:

```
def list_incorrect(w, X, y):
    '''
    w is the sparse weight vector
    X is the vector needed to be classified
    y is the true label
```

```
    ---------------------------
    incorrect_list is the percent of the wrong classifiation
    '''
    incorrect_list = []
    weight_list = []
    N = len(X)
    index = []
    y_p = []
    y_true = []
    wx = []
    for i in range(N):
        yy = dotProduct(w, X[i])
        if yy > 0:
            yy = 1
        else:
            yy = -1

        if yy !=y[i]:
            incorrect_list.append(X[i])
            index.append(i)
            y_p.append(yy)
            y_true.append(y[i])
            wx.append(dotProduct(w, X[i]))

    return incorrect_list, index, y_p, y_true,wx

import pandas as pd
summ = pd.DataFrame(index)
summ['incorr'] = incorrect
summ['y_predicted'] = yy
summ['y_true'] = yt
summ['w*x'] = wx
summ.to_csv('/Users/twff/Downloads/machine_learning/hw/hw3-sentiment/wronglist.csv')
```

Table 1: Example 1 (y-predict=1, y-true=-1)

Words	Weight	w*x
perfect	0.0066	0.0132
well	0.0124	0.0124
film	0.0249	0.2241
better	-0.0067	-0.0134
interesting	-0.0024	-0.0048
lovely	1e-5	0
boring	-0.0083	-0.0083

The prediction of label is incorrect may due to several reasons. First of all, it may due to some negative reviews also include some positive words, which is sarcasm. But actually, the true meaning is negative. Secondly, in some positive reviews, there are some phases, such as 'It can't be better', in this case, 'can't' takes big negative weight, it is likely that the reviews will

Table 2: Example 2 (y-predict=-1, y-true=1)

Words	Weight	w*x
new	0.0055	0.022
doesn't	-0.0039	-0.0234
sandler	-0.0013	-0.0143
robbie	0.0001	0.0009
wedding	0.0008	0.0064

be predicted negative if the phase appears for many times. The same thing also happens on 'doesn't'. Finally, some positive words are identified as negative, such as 'better', 'interesting', or have low positive weight, like 'lovely', thus there are chances that some positive instances including plenty of this kind of words my be identified as negative incorrectly.

So the solution is that we can use bigram as features instead of using a gram. Furthermore, we can use n-gram features. Or whenever 'not' appears, it will combine with its following words as a new feature.

8 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. 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, construct a new feature (or group of features) that you hope will 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 $\sqrt{p(1-p)/n}$ (where p is the proportion of the test examples you got correct, and n is the size of the test set) on your performance estimates, to know whether the improvement is statistically significant.

   Answer 8.1 In Biagram features:

   I am going to use bigram as feature instead of a single word, that is to make every pair of consecutive words a feature.

   The code list as below:

```
def bag_of_words_bigram(words):
    '''
    words is a list of words
    -------------------
    count sparse bag of words representation.
    '''
    stop = set(stopwords.words('english'))
    words_stopfree = [w for w in words if w not in stop]
    count = Counter(words_stopfree[:-1])
    label = words[-1][1]
    return count, label

Xb_train  = []
yb_train = []
bigram_list = []
i=0
j=0
while i < len(training_data)-1:
    bigram_list = []
    j= 0
    i += 1
    while j < len(training_data[i])-1:
        bigram_list.append((training_data[i][j],training_data[i][j+1]))
        j += 1

    words, label = bag_of_words_bigram(bigram_list)
    Xb_train.append(words)
```

```
    yb_train.append(label)

Xb_valid  = []
yb_valid = []
bigram_data_test = []
i=0
while i < len(validation_data)-1:
    bigram_list = []
    j= 0
    i += 1
    while j < len(validation_data[i])-1:
        bigram_list.append((validation_data[i][j],validation_data[i][j+1]))
        j += 1

    words, label = bag_of_words(bigram_list)
    Xb_valid.append(words)
    yb_valid.append(label)


n = len(Xb_valid)
w = pegasos_svm_change(Xb_train, yb_train, Lambda=0.1, num_iter=20)
error = percent_error(w, Xb_valid, yb_valid)
std_error = ((1-error)*error)/n
```

So that the single words are turned to be bigram

```
          ('ben', 'kingsley'): 3,
          ('ben', "kingsley's"): 1,
          ('beyond', 'an'): 1,
          ('bird', 'uses'): 1,
          ('boss', 'teddy'): 1,
          ('boulder', 'that'): 1,
          ('break', 'into'): 1,
          ('british', 'crime'): 1,
          ('british', 'gangster'): 1,
          ('buddy', 'and'): 1,
```

And the std error computed is 0.017852406801892162 in bigram examples, while in single word example, it is 0.015880554146502572, so we can conclude from the standard error that the new features is not statistically significant.

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.]

Answer 8.2 The code list as below:

```
def bag_of_words_bigram(words):
    '''
    words is a list of words
    -------------------
    count sparse bag of words representation.
    '''
    stop = set(stopwords.words('english'))
    words_stopfree = [w for w in words if w not in stop]
    count = Counter(words_stopfree[:-1])
    label = words[-1][1]
    return count, label

Xb_train  = []
yb_train = []
bigram_list = []
i=0
j=0
while i < len(training_data)-1:
    bigram_list = []
    j= 0
    i += 1
    while j < len(training_data[i])-1:
        bigram_list.append((training_data[i][j],training_data[i][j+1]))
        j += 1

    words, label = bag_of_words_bigram(bigram_list)
    Xb_train.append(words)
    yb_train.append(label)

Xb_valid  = []
yb_valid = []
bigram_data_test = []
i=0
while i < len(validation_data)-1:
    bigram_list = []
    j= 0
    i += 1
    while j < len(validation_data[i])-1:
```

```
            bigram_list.append((validation_data[i][j],validation_data[i][j+1]))
            j += 1

        words, label = bag_of_words(bigram_list)
        Xb_valid.append(words)
        yb_valid.append(label)


n = len(Xb_valid)
w = pegasos_svm_change(Xb_train, yb_train, Lambda=0.1, num_iter=20)
error = percent_error(w, Xb_valid, yb_valid)
std_error = ((1-error)*error)/n
```

So that the single words are turned to be bigram

```
            ('ben', 'kingsley'): 3,
            ('ben', "kingsley's"): 1,
            ('beyond', 'an'): 1,
            ('bird', 'uses'): 1,
            ('boss', 'teddy'): 1,
            ('boulder', 'that'): 1,
            ('break', 'into'): 1,
            ('british', 'crime'): 1,
            ('british', 'gangster'): 1,
            ('buddy', 'and'): 1,
```

And the std error computed is 0.017852406801892162 in bigram examples, while in single word example, it is 0.015880554146502572, so we can conclude from the standard error that the new features is not statistically significant.

In other cases:

Table 3: Example 1 (y-predict=1, y-true=-1)

methods	std error	w*x
words number	0.01733	
negative words	0.01746	
bigrams	0.015880	
not	0.01955	
tfidf	0.01832	