Machine Learning and Computational Statistics
Homework 3: SVM and Sentiment Analysis

2.Calculating Subgradients

1

According to the definition of the subgradirnt, for $g \in \partial f_k(x)$

$$f_k(x) + g^T(z - x) \leq f_k(z) \leq f(z)$$

Note that $f_k(x) = f(x)$,
therefore,

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

Therefore, $g \in \partial f(x)$

2

Let $J_1(w) = 0$, $J_2(w) = 1 - yw^T x$, $J(w) = \max\{J_1(w), J_2(w)\}$
If $J_1(w) = J(w)$, the subgradient of $J(w)$ is 0
If $J_2(w) = J(w)$, the subgradient of $J(w)$ is $-xy$
If $J_1(w) = J_2(w)$, the subgradient of $J(w)$ can be either $-xy$ or 0

3 Perceptron

1

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

Also, $w^T x_i = \hat{y}$
Therefore,

$$y_i \hat{y}_i > 0 \;\forall i \in \{1, \ldots, n\}\,.$$

and the average perceptron loss is,

$$\frac{1}{n}\sum_{i=1}^{n} \ell(\hat{y}_i, y_i) = \frac{1}{n}\sum_{i=1}^{n} \max\{0, -\hat{y}_i y_i\} = 0$$

Thus any separating hyperplane of $\mathcal{D}$ is an empirical risk minimizer for perceptron loss.

2

$$\nabla_w \ell(\hat{y}_i, y_i) = \begin{cases} -y_i x_i \quad \ell(\hat{y}_i, y_i) > 0 \quad y_i x_i^T w^{(k)} < 0 \\ 0 \quad \ell(\hat{y}_i, y_i) = 0 \quad y_i x_i^T w^{(k)} > 0 \\ -y_i x_i \quad \ell(\hat{y}_i, y_i) = 0 \quad y_i x_i^T w^{(k)} = 0 \end{cases}$$

The updating rule for fixed step size 1 is $w^{(k+1)} = w^{(k)} + \nabla_w \ell(\hat{y}_i, y_i)$. We can see that the updating rule are the same for two algorithms. And so are the terminating rule.

Therefore, the SSGD described in the question is exactly doing the percepton algorithm.

3

Without the loss of generality, suppose the algorithm terminate after m loops. w can be expressed as,

$$w = \sum_{i=1}^{n}\sum_{l=1}^{m} 1_{(y_i x_i^T w^{(ln+i-1)} \leq 0)} x_i y_i$$

let $\alpha_i = \sum_{l=1}^{m} 1_{(y_i x_i^T w^{(ln+i-1)} \leq 0)} y_i$

Therefore, we can write $w = \sum_{i=1}^{n} \alpha_i x_i$.

The points that we have used to update the w is support vectors. The points that we have never used is not support vectors.

4. The Data

```
data_words = shuffle_data()
Y = []
for i in range(len(data_words)):
    Y.append(data_words[i][-1])
    del data_words[i][-1]
data = []
for entry in data_words:
    data.append(bow_rep(entry))

import pickle
with open('review.pkl', 'wb') as f:
    pickle.dump(data_words, f)


train_X, test_X, train_y, test_y = train_test_split(data, Y, test_size
 ↪  = 0.25)
```

5. Sparse Representations

```
def bow_rep(word_list):
    return Counter(word_list)
```

6. Support Vector Machine via Pegasos

1

$$\nabla_w J_i(w) = \begin{cases} \lambda w - y_i x_i & y_i w^T x_i < 1 \\ \lambda w & y_i w^T x_i > 1 \\ undefined & y_i w^T x_i = 1 \end{cases}$$

2

The subgradient of $\partial \frac{\lambda}{2}$

$$\partial \frac{\lambda}{2} \|w\|^2 = \lambda w$$

We know from question 2 that,

$$\partial \max\{0, 1 - y_i w^T x_i\} = \begin{cases} -y_i x_i & y_i w^T x_i < 1 \\ 0 & y_i w^T x_i \geq 1 \end{cases}$$

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

Therefore, the subgradient of $J_i(w)$ is the one given by the question.

3

Doing SGD with the subgradient direction gives the following update rule,

$$w_{t+1} = w_t - \eta_t \partial J_i(w) = w_t - \eta_t g = \begin{cases} (1 - \eta_t \lambda) w_t + \eta_t y_j x_j & y_j w_t^T x_j < 1 \\ (1 - \eta_t \lambda) w_t & y_j w_t^T x_j \geq 1 \end{cases}$$

4

```python
def Pegasos_64(X, y, X_t, y_t, lambda_reg = 1, num_iter = 10, stop_diff
 ↪ = 10**-6):
    start = time.time()
    w={}
    t = 0
    num_instances = len(X)
    loss_hist = np.zeros(num_iter + 1) #initialize loss_hist
    loss_hist[0] = loss_func(X_t, y_t, w, lambda_reg)
    index = np.arange(num_instances)
    for i in range(num_iter):
        #np.random.shuffle(index)
        for j in index:
            t += 1
            eta = 1/(t*lambda_reg)
            if y[j]*dotProduct(X[j], w) < 1:
                increment(w, -eta*lambda_reg , w)
                increment(w, eta*y[j], X[j])
            else:
                increment(w, -eta*lambda_reg , w)
        loss_hist[i+1] = loss_func(X_t, y_t, w, lambda_reg)
        if abs(loss_hist[i+1]-loss_hist[i]) < stop_diff:
            print ('when lambda = {0}, Pegasos converge in {1}
             ↪ iteration\n'.format(lambda_reg, i))
            break
        if(i == num_iter - 1):
            print ('don\'t converge')
    end = time.time()
    print ('the time to run {0} iteration is {1}'.format(num_iter, end
     ↪ - start))
    return w, loss_hist[i+1]
```

5

We know that W is in fact sW, and $W_t = s_t W_t$

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

```
def Pegasos_65(X, y, X_t, y_t, lambda_reg = 1, start_t = 2, num_iter =
↪ 100, stop_diff = 10**-8, loss_function = percent_loss_func):
    start = time.time()
    w={}
    w_test = {}
    t = start_t
    num_instances = len(X)
    loss_hist = np.zeros(num_iter + 1) #initialize loss_hist
    loss_hist[0] = loss_func(X_t, y_t, w, lambda_reg)
    index = np.arange(num_instances)
    s = 1
    w_small = {}
    for i in range(num_iter):
        start = time.time()
        #np.random.shuffle(index)

        for j in index:
            t += 1
            eta = 1/(t*lambda_reg)
            s *= (1 - eta * lambda_reg)
            if y[j]*dotProduct(X[j], w) < 1/s:
                increment(w, 1/s*eta*y[j], X[j])
        w_small = {}
        increment(w_small, s, w)
        loss_hist[i+1] = loss_function(X_t, y_t, w_small, lambda_reg)
        if abs(loss_hist[i+1]-loss_hist[i]) < stop_diff:
            print ('when lambda = {0}, Pegasos converge in {1}
            ↪ iteration\n'.format(lambda_reg, i))
            break
        if(i == num_iter - 1):
            print ('don\'t converge')
    end = time.time()
    print ('the time to run {0} iteration is {1}'.format(num_iter, end
    ↪ - start))
    return w_small, loss_hist[i+1]
```

6

The w given by two approach is as follows,

Key	Type	Size	Value
earth	float	1	0.0005555555555555532
goodies	float	1	0.00031111111111110793
if	float	1	0.0005555555555555532
ripley	float	1	0.0005555555555555532
suspend	float	1	0.00031111111111110793
white	float	1	0.00031111111111110793
	float	1	0.0017333333333333398
	float	1	0.0011111111111111063
!	float	1	0.022711111111111133
#05425	float	1	0.00046666666666666834

Key	Type	Size	Value
earth	float	1	0.0005777520999066706
goodies	float	1	0.0003333185191769242
if	float	1	0.0005777520999066706
ripley	float	1	0.0005777520999066706
suspend	float	1	0.0003333185191769242
white	float	1	0.0003333185191769242
	float	1	0.0018221412381671897
	float	1	0.0011555041998133411
!	float	1	0.022576774365583436
#05425	float	1	0.000444424692235899

We can see that they are essentially the same. The small difference is due to the shuffling of the index.

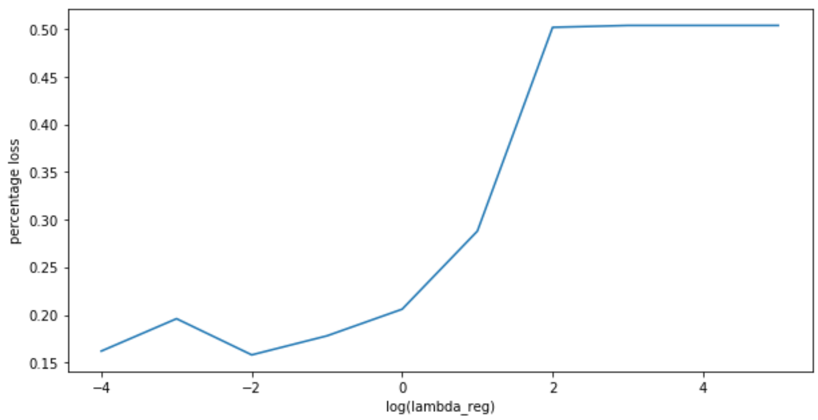
The time for each epoch using the first approach is 103. and the time for each epoch using the second approach is 0.0125.

7

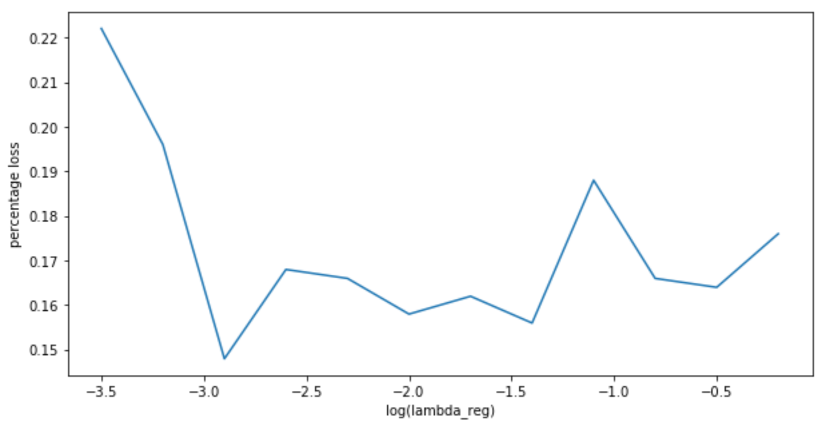
```
def percent_loss_func(X, y, w, lambda_reg):
    num_instances = len(X)
    loss = 0
    for i in range(num_instances):
        pred = dotProduct(X[i], w)
        if(pred * y[i] < 0):
            loss += 1
    loss /= num_instances
    return loss
```

8

My first search range is from $10^{-}4$ to 10^5, the plot is as follows,



Then I zoom in to $10^{-}3.5$ to 10^0, the plot is as follows,



When $\lambda = 0.00126$, the percentage is at its minimum 0.148.

9

```
lambda_reg_best = 0.04
w69, loss69 = Pegasos_65(train_X, train_y, test_X, test_y, lambda_reg = lambda_reg_best, start_t = 2, num_iter = 400, stop_diff = 10**-8, loss_function = percent_loss_func)
wxs69 = []
for i in range(len(test_y)):
    flag = 0
    pred = dotProduct(test_X[i], w69)
    if(pred * test_y[i] < 0):
        #misclassified
        flag = 1
    wxs69.append((abs(pred),flag))
wxs69_sorted = sorted(wxs69, key=lambda tup: tup[0], reverse = True)
fold = 5
num_each_fold =len(test_y) / fold
for i in range(fold):
    start_index = 100 * i
    end_index = 100 * i +100
    percentage_error = np.average([flag69 for flag69 in [tup[1] for tup in wxs69_sorted[start_index : end_index]]])
    print('for fold {}, the percentage error is {}'.format(i+1, percentage_error))
```

I divide the score into five groups. The first group has the largest absolute value of the score, the last group has the smallest.
The results is as follows,

```
for fold 1, the percentage error is 0.04
for fold 2, the percentage error is 0.15
for fold 3, the percentage error is 0.18
for fold 4, the percentage error is 0.29
for fold 5, the percentage error is 0.4
```

We can see significant negative relationship between the percentage error and the absolute value of the score.

10

```
if y[j]*dotProduct(X[j], w) == 1/s:
    print('not differentiable')
elif abs(y[j]*dotProduct(X[j], w) - 1/s) < diff_1:
    print('almost not differentiable')
```

There is no time when $y_i w^T x_i = 1$
When set the diff_1 to 10^{-9}, there is no instance of that, either.

I don't think we should skip the update. However, I also don't see how shortening the step size can help with this situation.

7. Error Analysis

The rank for the first misclassified review is as follows,

```
The result is as follows, the class for this review should be -1, however,
it has been misclassfied.
key      abs(wx)      w        x       xw
and     : 1.12763  -0.05638  20  -1.12763
horror  : 0.51062  -0.04255  12  -0.51062
do      : 0.47658  0.11915  4  0.47658
this    : 0.43438  0.06205  7  0.43438
he      : 0.37162  -0.04645  8  -0.37162
good    : 0.34325  -0.08581  4  -0.34325
on      : 0.33332  0.08333  4  0.33332
director: 0.27765  0.09255  3  0.27765
as      : 0.23262  -0.02908  8  -0.23262
what    : 0.22588  -0.03227  7  -0.22588
```

The rank for the second misclassified review is as follows,

```
The result is as follows, the class for this review should be -1, however,
it has been misclassfied.
key      abs(wx)      w        x       xw
and     : 0.95848  -0.05638  17  -0.95848
bad     : 0.85317  0.28439  3  0.85317
scream  : 0.74998  -0.08333  9  -0.74998
only    : 0.65246  0.16312  4  0.65246
also    : 0.617  -0.10283  6  -0.617
this    : 0.43438  0.06205  7  0.43438
have    : 0.40992  0.10248  4  0.40992
movies  : 0.39963  -0.05709  7  -0.39963
the     : 0.38722  -0.00993  39  -0.38722
do      : 0.35744  0.11915  3  0.35744
```

We can see from the two examples that the stop words like 'do', 'this','have' contribute a lot to the misclassification, which should not be in the feature in the first place. I would include tfidf and binary feature to fix this issue.

8. Features

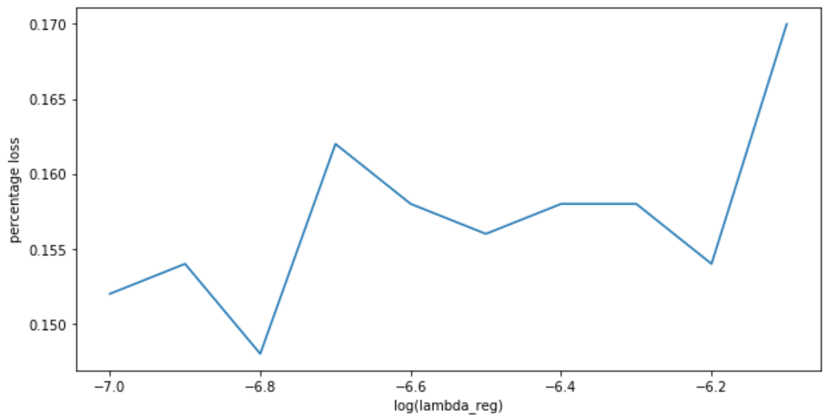
Because of the presense of the stop words in data, I use tf-idf instead of raw word counts.

```
data_words = shuffle_data()
Y = []
for i in range(len(data_words)):
    Y.append(data_words[i][-1])
    del data_words[i][-1]
data = []
for entry in data_words:
    data.append(bow_rep(entry))
data_tfidf = []
key_list = []
for entry in data:
    for key in entry:
        if key not in key_list:
            key_list.append(key)
for entry in data:
    data_tfidf.append(entry)
#data_tfidf = [{'1':1, '2':2}, {'1':4, '2':6, '3':8}]
for entry in data_tfidf:
    total_num_words = len(entry)
    for key in entry:
        entry[key] /= total_num_words
inv_log_num_time_words = {}
for key in key_list:
    num = 0.0
    for entry in data:
        if entry[key] != 0:
            num += 1
    if num == 1:
        DF = 1/2
    else:
        DF = np.log10(num)
    inv_log_num_time_words[key] = 1/DF

for entry in data_tfidf:
    for key in entry:
        entry[key] *= inv_log_num_time_words[key]

train_X_tfidf, test_X_tfidf, train_y_tfidf, test_y_tfidf =
 ↪ train_test_split(data_tfidf, Y, test_size = 0.25)
```

```
plot_Pegasos_lambda(train_X_tfidf, train_y_tfidf, test_X_tfidf,
 ↪  test_y_tfidf, -9, -3, 1, num_iter = 400)
plot_Pegasos_lambda(train_X_tfidf, train_y_tfidf, test_X_tfidf,
 ↪  test_y_tfidf, -7, -6, 0.1, num_iter = 400)
```



When lambda = 1.5848e-07, percentage loss is 0.148.h t statistics is,

$$t = \frac{|p_1 - p_0|}{\sqrt{\frac{p_0(1-p_0)}{n}}} = \frac{|0.148 - 0.15|}{\sqrt{\frac{0.148*(1-0.148)}{500}}} = 2.816 > 1.64$$

Therefore, the improvement is statistically significant.