Q1: Perceptron

1

Since we know g is a subgradient of $f_k(x)$, by definition we have

$$f_k(z) \geq f_k(x) + g^T(z-x), \forall z,$$

and since f_k(x) = f(x), the above equation can be re-written as

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

, and by definition, this means g is a subgradient of f.

2

$$g = \begin{cases} 0, 1 - yw^T x < 0 \\ -yx, else \end{cases}$$

3

If w does define a separating hyperline, then all data points in $\mathbb{D}$ will be correctly classified. Thus, the perceptron loss

$$l(\tilde{y}, y) = \max\{0, -\tilde{y}y\} = 0, \forall y$$

because $\tilde{y}y = 1, \forall y$ (either $1 \times 1 or - 1 \times -1$). Since all single perceptron loss is 0, the average loss will be 0 too on $\mathbb{D}$.

4

SSGD to minimize empirical risk of perceptron is equivalent to perceptron algorithm. This is because of the gradient w.r.t w of the term $max\{0, 1 - yw^T x\}$,

$$\nabla max\{0, 1 - yw^T x\} = \begin{cases} 0, & \text{if } yw^T x \geq 1 \\ -yx, & \text{otherwise.} \end{cases}$$

As we can see, running gradient descent using the above equation is equivalent to using an if-else condition to update on data points that aren't correctly classified $(yw^T x < 1)$

5

Since in the perceptron algorithm update rule, for each point (x_i, y_i), according to last question, nothing gets updated if x_i is correctly classified or else $w+ = y_i x_i$, we can see that when the algorithm converges, all the updates were in the form $y_i x_i$. Thus, the final weight vector w will be a linear combination of the input points. More specifically, we can write w as $w = \sum_{i=0}^{n} \alpha_i y_i x_i$, where $\alpha_i = 0$ when x_i is correctly classified else $\alpha_i = 1$. Thus, the support vectors are closer to the separating hyperplane than the non support vectors.

Q2: Sparse Representations

1

In [3]:

```
from load import *
```

In [4]:

```
import zipfile
with zipfile.ZipFile("data.zip","r") as zip_ref:
    zip_ref.extractall()
```

In [5]:

```
! rm -rf data/neg/.ipynb_checkpoints
```

In [6]:

```
shuffle_data()
```

In [7]:

```
with open('review.pkl', 'rb') as f:
    review = pickle.load(f)
```

Since load.py has already shuffled the data, no need to shuffle again here

In [8]:

```
assert len(review) == 2000
review_train = review[:1500]
train_labels = [r[-1] for r in review_train]
review_train = [r[:-1] for r in review_train]

review_val = review[-500:]
val_labels = [r[-1] for r in review_val]
review_val = [r[:-1] for r in review_val]

assert len(review_train) == 1500
assert len(train_labels) == 1500
assert len(review_val) == 500
assert len(val_labels) == 500
```

2

In [9]:

```
from collections import Counter
def convert_sparse_representation(list_of_words):
    return Counter(list_of_words)
convert_sparse_representation(["Harry", "Potter", "and", "Harry", "Potter", "II"
])
```

Out[9]:

```
Counter({'Harry': 2, 'Potter': 2, 'and': 1, 'II': 1})
```

Q3: SVM with via Pegasos

1

$$\nabla J_i(w) = \begin{cases} \lambda w, & \text{if } y_i w^T x_i > 1 \\ -y_i x_i + \lambda w, \text{if } y_i w^T x_i < 1 \\ \text{undefined, if } y_i w^T x_i = 1 \end{cases}$$

2

This is true because g is just the real gradient of $\nabla J_i(w)$ with the point where it is undefined λw. Thus g is continuous and $g \leq \nabla J_i(w)$.

3

$$w = \begin{cases} w - \eta_t(\lambda w - y_i x_i) = (1 - \eta_t \lambda)w + \eta_t y_i x_i, & \text{if } y w^T x < 1 \\ w - \eta_t \lambda w = (1 - \eta_t \lambda)w, & \text{otherwise.} \end{cases}$$

As we can see, this is equivalent to the Pegaso update rule.

4

In [56]:

```
import collections
import math
import time
def pegasos_dict(X, y, lambda_reg = 0.1, max_epochs = 1000, verbose = True):
    epoch = 0
    t = 0.
    w = collections.defaultdict(float)
    times = []
    cur_loss = float("inf")
    while epoch < max_epochs:
        tic = time.perf_counter()
        epoch += 1
        for i, y_i in enumerate(y):
            t += 1
            eta = 1.0/(lambda_reg*t)
            x_i = convert_sparse_representation(X[i]) # counter

            temp = {k: v*(1 - eta*lambda_reg) for k, v in w.items()}

            w_dot_xi = sum(x_i.get(k, 0) * v for k, v in w.items())
            if y_i*w_dot_xi < 1:
                for k, v in x_i.items():
                    w[k] = temp.get(k,0) + v * eta * y_i
            else:
                w = temp.copy()
        toc = time.perf_counter()
        times.append(toc-tic)
        prev_loss = cur_loss
        cur_loss = svm_loss(X, y, w)
        if verbose:
            print('Epoch', str(epoch), "loss:", cur_loss)
        if cur_loss > prev_loss:
            return w, sum(times)/len(times)
    return w, sum(times)/len(times)

def svm_loss(X, y, w, lambda_reg = 0.1):
    reg_penalty = 0.
    for k,v in w.items():
        reg_penalty += v**2
    reg_penalty *= lambda_reg/2

    margin_loss = 0
    for i, y_i in enumerate(y):
        x_i = convert_sparse_representation(X[i]) # counter
        w_dot_xi = sum(x_i.get(k, 0) * v for k, v in w.items())
        reg_penalty += max(0,1-y_i*w_dot_xi)
    reg_penalty /= len(y)

    return margin_loss + reg_penalty
```

5

$$w_{t+1} = (1 - \eta_t \lambda) w_t + \eta_t y_i x_i = (1 - \eta_t \lambda) s_t W_t + \eta_t y_i x_i = s_{t+1} W_t + \eta_t y_i x_i = s_{t+1}(W_t + \frac{1}{s_{t+1}} \eta_t y_i x_i) = s$$

In [23]:

```
def pegasos_accerlated(X, y, lambda_reg = 0.1, max_epochs = 1000, verbose = True
):
    epoch = 0
    t = 1.
    s_t = 1.
    W = collections.defaultdict(float)
    w = collections.defaultdict(float)
    times = []
    cur_loss = float("inf")
    while epoch < max_epochs:
        tic = time.perf_counter()
        epoch += 1
        for i, y_i in enumerate(y):
            t += 1
            eta = 1.0/(lambda_reg*t)
            s_t *= (1-eta*lambda_reg)
            x_i = convert_sparse_representation(X[i]) # counter
            w_dot_xi = sum(w.get(k, 0) * v for k, v in x_i.items())
            if w_dot_xi*y_i < 1:
                for k, v in x_i.items():
                    W[k] = W[k] + (1/s_t)*eta*y_i*v
            w = {k: s_t*v for k, v in W.items()}
        toc = time.perf_counter()
        times.append(toc-tic)
        prev_loss = cur_loss
        cur_loss = svm_loss(X, y, w)
        if verbose:
            print('Epoch', str(epoch), "loss:", cur_loss)
        if cur_loss > prev_loss:
            return w, sum(times)/len(times)
    return w, sum(times)/len(times)
```

6

In [24]:

```
w1,t1 = pegasos_dict(review_train, train_labels, lambda_reg = 0.1, max_epochs =
1000)
w2,t2 = pegasos_accerlated(review_train, train_labels, lambda_reg = 0.1, max_epo
chs = 1000)
```

```
Epoch 1 loss: 1.2173642174031287
Epoch 2 loss: 0.7622658345683799
Epoch 3 loss: 0.25437708010300636
Epoch 4 loss: 0.1920115289110334
Epoch 5 loss: 0.5285685053689315
Epoch 1 loss: 1.3517782039747592
Epoch 2 loss: 0.7627414042407208
Epoch 3 loss: 0.2567636132699729
Epoch 4 loss: 0.21322456509243493
Epoch 5 loss: 0.38204383077619736
```

In [51]:

```
print("the first implementation takes on average {0:.2f} seconds per epoch".form
at(t1))
print("the accelerated implementation takes on average {0:.2f} seconds per epoc
h".format(t2))
```

```
the first implementation takes on average 32.76 seconds per epoch
the accelerated implementation takes on average 16.40 seconds per ep
och
```

In [47]:

```
word = "hi"
print(w1[word])
print(w2[word])
```

```
-0.0029994545391896295
-0.0026663111585121893
```

7

In [52]:

```
def evaluate(w, X, y):
    error = 0
    total = 0
    for i, y_i in enumerate(y):
        x_i = convert_sparse_representation(X[i])
        w_dot_xi = sum(x_i.get(k, 0) * v for k, v in w.items())
        if (w_dot_xi > 0 > y_i) or (w_dot_xi < 0 < y_i):
            error += 1
        total += 1
    return error/total
```

8

In [57]:

```
lambdas = [1e-5,1e-4,1e-3,1e-2,5e-2,1e-1,5e-1,1,10]
errors = []
for lambda_reg in lambdas:
    print("Testing: ", lambda_reg)
    w,t = pegasos_accerlated(review_train, train_labels, lambda_reg = lambda_reg
,\
                             max_epochs = 10, verbose=False)
    errors.append(evaluate(w, review_val, val_labels))
```

```
Testing:  1e-05
Testing:  0.0001
Testing:  0.001
Testing:  0.01
Testing:  0.05
Testing:  0.1
Testing:  0.5
Testing:  1
Testing:  10
```

In [58]:

```
import numpy as np
print("The best lambda is {}".format(lambdas[np.argmin(errors)]),\
      "with lowest error rate {0:.2f}".format(min(errors)))
```

```
The best lambda is 1e-05 with lowest error rate 0.18
```

Q5

1

Let V be the total vocabulary from both documents without duplicates. Then for any document x, for the k-th word w_k in vocabulary V, if w_k appears in x then $\phi(x)_k = 1$ else 0. Thus, both $\phi(x)$ and $\phi(z)$ will be vectors of length $|V|$. Then, $k(x, z) = \phi(x)^T \phi(z)$ will be the unique number of words that appear in both documents.

2

Let $k(x, z) = x^T z$ be a kernel, $\frac{1}{\|x\|_2} = f(x)$, $\frac{1}{\|z\|_2} = f(z)$. Then

$$k_1(x, z) = f(x)f(z)k(x, z) = \frac{1}{\|x\|_2}\frac{1}{\|z\|_2}x^T z = (\frac{x}{\|x\|_2})^T(\frac{z}{\|z\|_2})$$

is also a kernel. Since 1 is also a kernel by constant feature mapping $\phi(x) = 1$,

$$k_2(x, z) = 1 + k_1(x, z) = 1 + (\frac{x}{\|x\|_2})^T(\frac{z}{\|z\|_2})$$

is also a kernel. Finally, given $k_2(x, z)$ is a kernel, we can apply the product rule twice, thus

$$k_3(x, z) = (k_2(x, z))^3 = (1 + (\frac{x}{\|x\|_2})^T(\frac{z}{\|z\|_2}))^3$$

is a kernel.

Q6

1

This is equivalent to proving $(w^{(t)})^T x_j = K_{j.} a^{(t)}$

First, $(w^{(t)})^T = \begin{pmatrix} a_1 & a_2 & \dots & a_n \end{pmatrix} \begin{pmatrix} \vec{x_1} \\ \vec{x_2} \\ \vdots \\ \vec{x_n} \end{pmatrix}$.

Thus, we can write

$$(w^{(t)})^T x_j = \begin{pmatrix} a_1 & a_2 & \dots & a_n \end{pmatrix} \begin{pmatrix} \overrightarrow{x_1} \\ \overrightarrow{x_2} \\ \vdots \\ \overrightarrow{x_n} \end{pmatrix} \begin{pmatrix} x_{j_1} \\ x_{j_2} \\ \vdots \\ x_{j_d} \end{pmatrix} = \begin{pmatrix} a_1 & a_2 & \dots & a_n \end{pmatrix} \begin{pmatrix} k(x_1, x_j) \\ k(x_2, x_j) \\ \vdots \\ k(x_n, x_j) \end{pmatrix} = (a^t)^T (K_{j.})^T =$$

2

Since there is no margin violation, $w^{(t+1)} = (1 - \eta_t \lambda) w^{(t)}$. Thus,

$$a^{(t+1)} = (1 - \eta_t \lambda) a^{(t)}$$

3

First, write $w^{(t)}$ as $w^{(t)} = \begin{pmatrix} \overrightarrow{x_1} & \dots & \overrightarrow{x_n} \end{pmatrix} \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix} = X^T a^{(t)}$, and let $\overrightarrow{1_j} = \begin{pmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix}$ be the n-dimensional row vector with only the j-th entry being 1, rest is all 0.

Then, given the update on with margin violation example x_j:

$$w^{(t+1)} = (1 - \eta_t \lambda) w^{(t)} + \eta_t y_j x_j,$$

we can rewrite it as

$$X^T a^{(t+1)} = (1 - \eta_t \lambda) X^T a^{(t)} + \eta_t y_j X^T \overrightarrow{1_j}$$

Removing X^T, we get

$$a^{(t+1)} = (1 - \eta_t \lambda) a^{(t)} + \eta_t y_j \overrightarrow{1_j}$$

Algorithm 1: Kernelized Pegasos

Result: Return $a^{(t+1)}$
Kernel matrix K, $\lambda > 0$, $t = 0$, $y_1, ..., y_n \in \{-1, 1\}$, $a^0 = \vec{0} \in R^n$;
while *not converged* **do**
    $t \leftarrow t + 1$;
    $\eta_t \leftarrow 1/(t\lambda)$;
    for $i \leftarrow 1$ **to** n **do**
        $K_j \leftarrow$ j-th row of k;
        $a^{(t+1)} \leftarrow (1 - \eta_t\lambda)a^{(t)}$;
        if $y_j K_j^T a^{(t)} < 1$ **then**
            $a^{(t+1)} \leftarrow (1 - \eta_t\lambda)a^{(t)} + \eta_t y_j \vec{1_j}$;
        else
            $a^{(t+1)} \leftarrow (1 - \eta_t\lambda)a^{(t)}$;
        end
    end
end