Homework 3

Name: Zheyuan Hu

NetID: zh2095

Subgradients

Q1

We choose that

$$g \in \partial f_k(x)$$

then g is a subgradient of f_k, that is,

$$f_k(z) \geq f_k(x) + g^T(z - x) \text{ for all } z$$

and

$$f_k(x) = f(x)$$

so

$$f(z) \geq f(x) + g^T(z - x) \text{ for all } z$$

Therefore, by definition,

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

Q2

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

$$g = \begin{cases} -yx & \text{for } yw^T x < 1 \\ 0 & \text{for } yw^T x \geq 1 \end{cases}$$

Justify:

When $yw^T x < 1$,

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

$g = -yx \in \partial(1 - yw^T x)$, so

$$g = -yx \in \partial J(w)$$

Similarly when $yw^T x \geq 1$,

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

$g = 0 \in \partial(0)$, so

$$g = 0 \in \partial J(w)$$

SVM with the Pegasos algorithm

Q3

The gradient of $J_i(w)$ is not defined at $y_i w^T x_i = 1$

The expression for the gradient of $J_i(w)$ where it is defined is

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

Q4

When $y_i w^T x_i < 1$,

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

Let $f_1(w) = \frac{\lambda}{2}\|w\|^2$, $f_2(w) = 1 - y_i w^T x_i$, so $J_i = f_1 + f_2$

Since f_1 and f_2 are convex and differentiable,

$$\partial f_1(w) = \{\nabla f_1(w)\} = \{\lambda w\}$$

$$\partial f_2(w) = \{\nabla f_2(w)\} = \{-y_i x_i\}$$

Then

$$\partial J_i(w) = \partial f_1(w) + \partial f_2(w) = \{\lambda w - y_i x_i\}$$

$$gw = \lambda w - y_i x_i$$

When $y_i w^T x_i \geq 1$,

$$J_i(w) = \frac{\lambda}{2}\|w\|^2$$

Then $J_i(w)$ is convex and differentiable, that is,

$$\partial J_i(w) = \{\nabla J_i(w)\} = \{\lambda w\}$$

$$gw = \lambda w$$

Therefore,

$$gw = \begin{cases} \lambda w - y_i x_i & \text{for } y_i w^T x_i < 1 \\ \lambda w & \text{for } y_i w^T x_i \geq 1. \end{cases}$$

Dataset and sparse representation

Q5

In [12]:

```
from collections import Counter

def bag_of_words(list_words):
    '''
    converts a list of words into a sparse bag-of-words representation
    '''
```

```
    dict_words = Counter(list_words)
    return dict_words
```

Q6

In [13]:

```
from utils_svm_reviews import *
from sklearn.model_selection import train_test_split
```

In [14]:

```
review = load_and_shuffle_data()
X = [bag_of_words(x[:-1]) for x in review]
y = [x[-1] for x in review]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25)
```

Q7

In [4]:

```
def pegasos(X, y, lambda_reg, num_epochs):
    # initial values
    t = 0
    w = {}
    for x in X:
        w.update(x)
    w = {i: 0 for i in w}

    n = len(X)
    for i in range(num_epochs):
        for j in range(n):
            t = t+1
            eta = 1/(t*lambda_reg)

            if y[j] * dotProduct(w, X[j]) < 1:
                increment(w, -eta*lambda_reg, w)
                increment(w, eta*y[j], X[j])
            else:
                increment(w, -eta*lambda_reg, w)

    return w
```

Q8

We have

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

Therefore,

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

In [5]:
```
def pegasos_with_sW(X, y, lambda_reg, num_epochs):
    # initial values
    s = 1
    # start at t = 2 to avoid divided by 0
    t = 1
    W = {}
    for x in X:
        W.update(x)
    W = {i: 0 for i in W}

    n = len(X)
    for i in range(num_epochs):
        for j in range(n):
            t = t+1
            eta = 1/(t*lambda_reg)

            if y[j] * dotProduct(W, X[j]) < 1/s:
                s = (1-eta*lambda_reg) * s
                increment(W, eta*y[j]/s, X[j])
            else:
                s = (1-eta*lambda_reg) * s

    w = {k: W[k]*s for k in W}
    return w
```

Q9

Implement the Pegasos algorithm without (s, W) representation:

In [6]:
```
import time

# inputs
X = X_train
y = y_train
lambda_reg = 1e-2
num_epochs = 5

start_time = time.time()
w_1 = pegasos(X, y, lambda_reg, num_epochs)

time_taken = time.time() - start_time
print('The time taken for running %d epoches of the algorithm w/o (s,W) is'%num_epocl
```

```
The time taken for running 5 epoches of the algorithm w/o (s,W) is 64.32048034667969
```

Implement the Pegasos algorithm with (s, W) representation:

In [7]:
```
# inputs
X = X_train
y = y_train
lambda_reg = 1e-2
num_epochs = 5

start_time = time.time()
w_2 = pegasos_with_sW(X, y, lambda_reg, num_epochs)

time_taken = time.time() - start_time
print('The time taken for running %d epoches of the algorithm w/ (s,W) is'%num_epoch
```

```
The time taken for running 5 epoches of the algorithm w/ (s,W) is 0.6455962657928467
```

Check that the two approaches give essentially the same result:

In [8]:
```
diff = {k: w_1[k] - w_2[k] for k in w_1}
list(diff.items())[:20]
```

Out[8]:
```
[('at', -3.555081544498462e-05),
 ('the', 6.75465493578109e-05),
 ('outset', -2.3108030041407712e-05),
 ('of', 9.243212016607494e-05),
 ('swordfish', -3.199573390358368e-05),
 ('john', -5.3326223170946374e-05),
 ("travolta's", 3.555081544578953e-06),
 ('gabriel', -6.221392703187334e-05),
 ('shear', -5.332622317291702e-06),
 ('is', 5.5103763942521145e-05),
 ('pontificating', -1.7775407726051962e-06),
 ('about', -0.00014398080256827776),
 ('status', 1.066524463441687e-05),
 ('american', 0.00013687063947087063),
 ('cinema', 6.221392703409379e-05),
 ('today', 5.865884549105527e-05),
 ('basically', -3.0218193132147686e-05),
 ('he', 0.00010309736480240694),
 ('says', -1.0665244633695226e-05),
 ('it', 2.6663111586056054e-05)]
```

We can see that the two weights essentially have no difference

Q10

In [15]:
```
def classification_error(X, y, w):
    n = len(y)
    e = 0
    for i in range(n):
        y_pred_i = np.sign(dotProduct(w, X[i]))
        if y_pred_i != y[i]:
            e += 1

    err = e/n
    return err
```

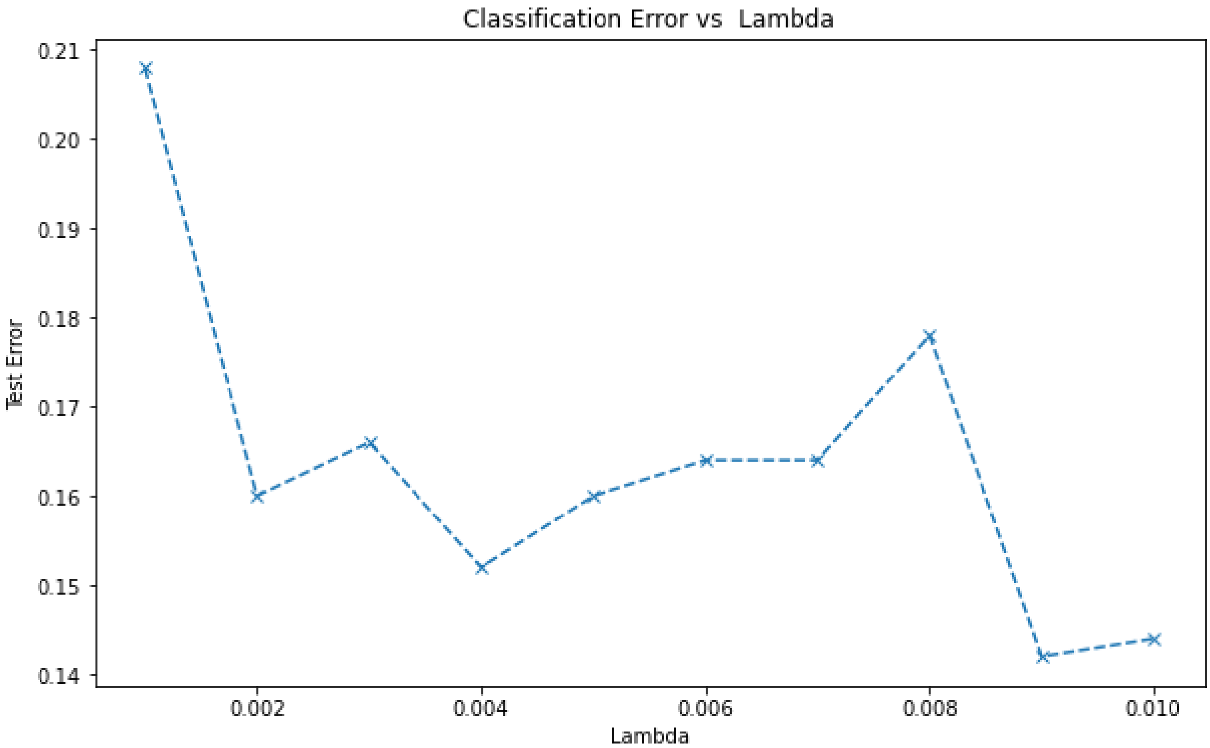
In [16]:
```
X = X_test
y = y_test
classification_error(X, y, w_2)
```

Out[16]:
```
0.186
```

Q11

In [17]:
```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(10,6))
lamb_set = [1e-3, 2e-3, 3e-3, 4e-3, 5e-3, 6e-3, 7e-3, 8e-3, 9e-3, 1e-2]
err_set = []

for lambda_reg in lamb_set:
    w = pegasos_with_sW(X_train, y_train, lambda_reg, num_epochs=20)
    err = classification_error(X_test, y_test, w)
    err_set.append(err)

ax.plot(lamb_set, err_set, 'x--')
ax.set_title('Classification Error vs  Lambda')
ax.set_xlabel('Lambda')
```

```
ax.set_ylabel('Test Error')

plt.show()

min_err = np.min(err_set)
best_lamb = lamb_set[np.argmin(err_set)]
print('The minimum test error %0.3f occurs when lambda = %0.3f'%(min_err, best_lamb))
```




```
The minimum test error 0.142 occurs when lambda = 0.009
```

Error Analysis

Q12

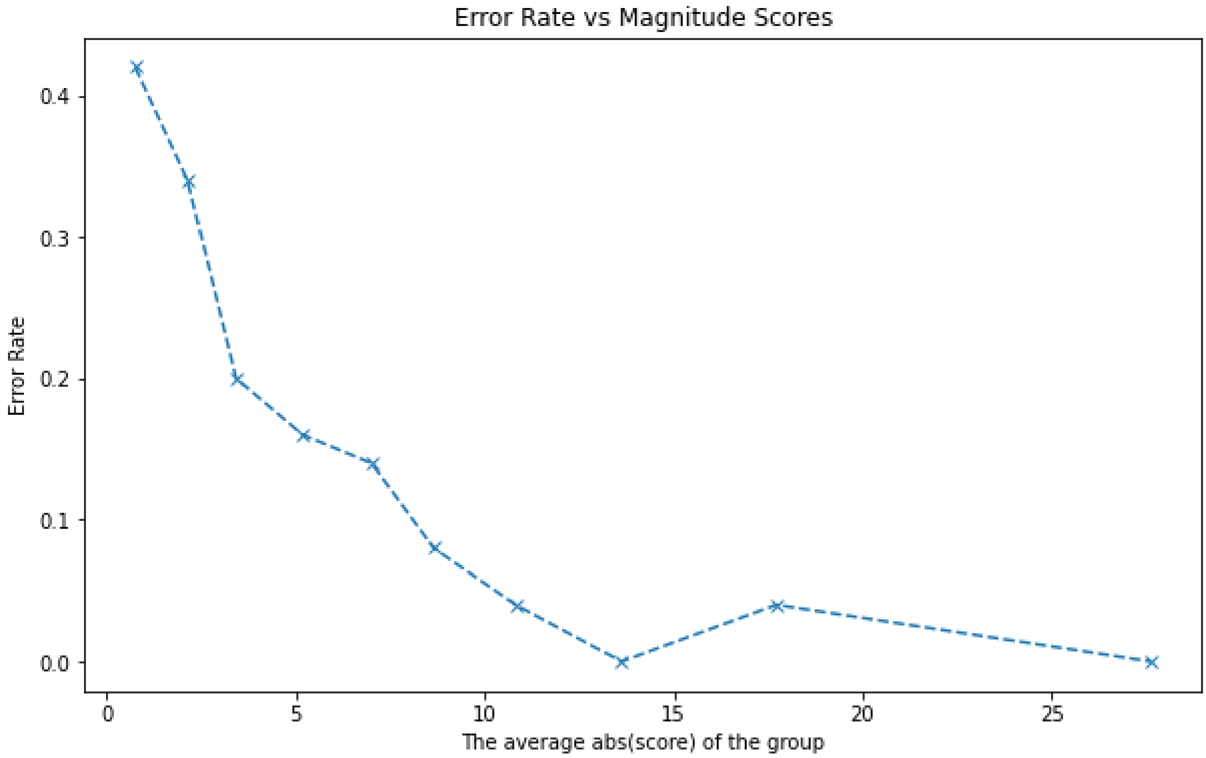
In [18]:

```
# choose the lambda that gives the minimal error rate
lambda_reg = 0.009
w = pegasos_with_sW(X_train, y_train, lambda_reg, num_epochs=20)

n = len(y_test)
# pairs of the score and its corresponding prediction(correct/wrong)
pairs = np.zeros([n,2])
for i in range(n):
    score = dotProduct(w, X_test[i])
    y_pred_i = np.sign(score)
    # ind indicates if the score corresponding to a correct prediction
    ind = 1 if y_pred_i == y_test[i] else 0

    pairs[i,0] = score
    pairs[i,1] = ind
# order by absolute value of scores
pairs = np.abs(pairs)
pairs = pairs[pairs[:,0].argsort()]
```

In [19]:

```
# each group have the same group size
num_groups = 10
group_size = int(n/num_groups)
```

```
avg_score_set = []
err_rate_set = []
for i in range(num_groups):
    group = pairs[i*group_size : (i+1)*group_size]
    avg_score = np.mean(group[:,0])
    err_rate = 1 - np.sum(group[:,1])/group_size
    avg_score_set.append(avg_score)
    err_rate_set.append(err_rate)

fig, ax = plt.subplots(figsize=(10,6))

ax.plot(avg_score_set, err_rate_set, 'x--')
ax.set_title('Error Rate vs Magnitude Scores')
ax.set_xlabel('The average abs(score) of the group')
ax.set_ylabel('Error Rate')

plt.show()
```



Essentially, the model predicts more accurate results as the magnitude score goes up.

Ridge Regression: Theory

Q14

To minimize $J(w)$, we must have

$$\nabla J(w) = 2X^TXw - 2X^Ty + 2\lambda Iw = 0$$

That is,

$$X^TXw + \lambda Iw = X^Ty$$

From Appendix A(2), we know that

$$X^TX \text{ is psd}$$

then from Appendix B(3)

$$X^TX + \lambda I \text{ is spd for any } \lambda > 0$$

Finally, from Appendix B(2), we prove that

$$X^TX + \lambda I \text{ is invertible}$$

Therefore, the minimizer of $J(w)$ is the solution to $X^TXw + \lambda Iw = X^Ty$:

$$w = (X^TX + \lambda I)^{-1}X^Ty$$

Q15

$$w = \frac{1}{\lambda}(X^Ty - X^TXw) = X^T[\frac{1}{\lambda}(y - Xw)]$$

Therefore, w can be rewritten in the form of $w = X^T\alpha$, where

$$\alpha = \frac{1}{\lambda}(y - Xw)$$

Q16

We know that w can be written in the form of $w = X^T\alpha$, that is, $w = \sum_{i=1}^n \alpha_i x_i$, which is a linear combination of the data set $\{x_i\}_{i=1}^n$.

Therefore, we say w is in the span of the data.

Q17

We have

$$\alpha = \frac{1}{\lambda}(y - Xw)$$

Replace w with $w = X^T\alpha$, and solve for α:

$$\begin{aligned} \alpha &= \frac{1}{\lambda}(y - XX^T\alpha) \\ \lambda I\alpha &= y - XX^T\alpha \\ (XX^T + \lambda I)\alpha &= y \\ \alpha &= (XX^T + \lambda I)^{-1}y \end{aligned}$$

Q18

Let K be the kernel matrix XX^T, then

$$\begin{aligned} Xw &= XX^T\alpha && (w = X^T\alpha) \\ &= XX^T(\lambda I + XX^T)^{-1}y && (\alpha = (\lambda I + XX^T)^{-1}y) \\ &= K(\lambda I + K)^{-1}y \end{aligned}$$

Q19

Define

$$k_x = \begin{pmatrix} x^T x_1 \\ \vdots \\ x^T x_n \end{pmatrix}$$

Then

$$k_x = x^T X^T$$

$$\begin{aligned} f(x) &= x^T w^* \\ &= x^T X^T \alpha^* \\ &= k_x \alpha^* \end{aligned}$$

Kernels and Kernel Machines

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
import sklearn
import scipy.spatial
import functools

%matplotlib inline
```

Q20

In [2]:

```
### Kernel function generators
def linear_kernel(X1, X2):
    """
    Computes the linear kernel between two sets of vectors.
    Args:
        X1 - an n1xd matrix with vectors x1_1,...,x1_n1 in the rows
        X2 - an n2xd matrix with vectors x2_1,...,x2_n2 in the rows
    Returns:
        matrix of size n1xn2, with x1_i^T x2_j in position i,j
    """
    return np.dot(X1,np.transpose(X2))

def RBF_kernel(X1,X2,sigma):
    """
    Computes the RBF kernel between two sets of vectors
    Args:
        X1 - an n1xd matrix with vectors x1_1,...,x1_n1 in the rows
        X2 - an n2xd matrix with vectors x2_1,...,x2_n2 in the rows
        sigma - the bandwidth (i.e. standard deviation) for the RBF/Gaussian kernel
    Returns:
        matrix of size n1xn2, with exp(-||x1_i-x2_j||^2/(2 sigma^2)) in position i,j
    """
    #TODO
    dist = scipy.spatial.distance.cdist(X1,X2,'sqeuclidean')
    return np.exp(-(dist)/(2*sigma**2))

def polynomial_kernel(X1, X2, offset, degree):
    """
    Computes the inhomogeneous polynomial kernel between two sets of vectors
    Args:
        X1 - an n1xd matrix with vectors x1_1,...,x1_n1 in the rows
        X2 - an n2xd matrix with vectors x2_1,...,x2_n2 in the rows
        offset, degree - two parameters for the kernel
    Returns:
```

```
        matrix of size n1xn2, with (offset + <x1_i,x2_j>)^degree in position i,j
    """
    #TODO
    inner = linear_kernel(X1, X2)
    return (offset + inner)**degree
```

Q21

In [3]:

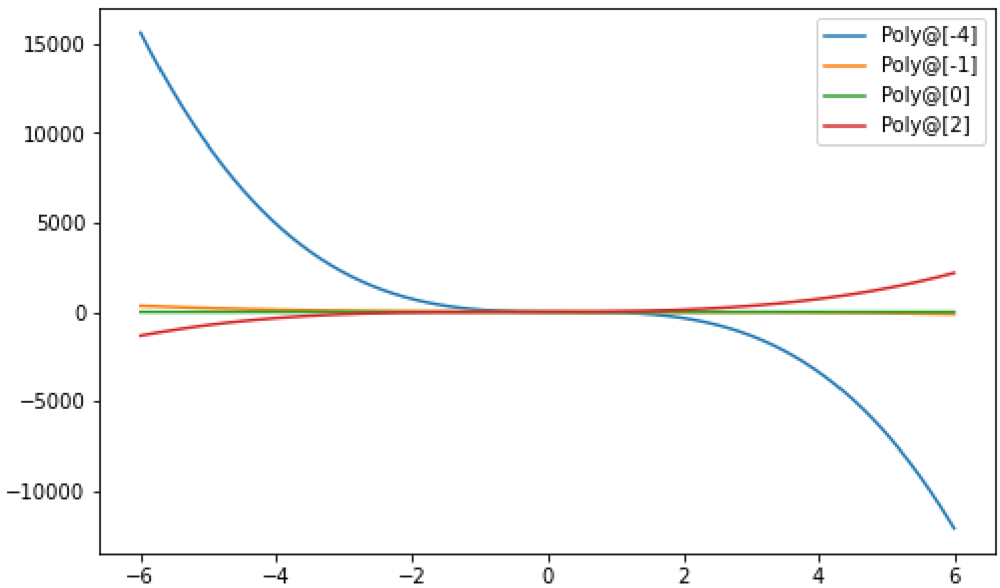
```
X0 = np.array([-4,-1,0,2]).reshape(-1,1)
K = linear_kernel(X0, X0)

print('The kernel matrix on X0 is:')
print(K)
```

```
The kernel matrix on X0 is:
[[16  4  0 -8]
 [ 4  1  0 -2]
 [ 0  0  0  0]
 [-8 -2  0  4]]
```

Q22(a)

In [4]:

```
# PLot kernel machine functions

plot_step = .01
xpts = np.arange(-6.0, 6, plot_step).reshape(-1,1)
prototypes = np.array([-4,-1,0,2]).reshape(-1,1)

# Polynomial kernel
plt.figure(figsize = (8,5))
y_poly = polynomial_kernel(prototypes, xpts, 1, 3)
for i in range(len(prototypes)):
    label = "Poly@"+str(prototypes[i,:])
    plt.plot(xpts, y_poly[i,:], label=label)

plt.legend(loc = 'best')
plt.show()
```

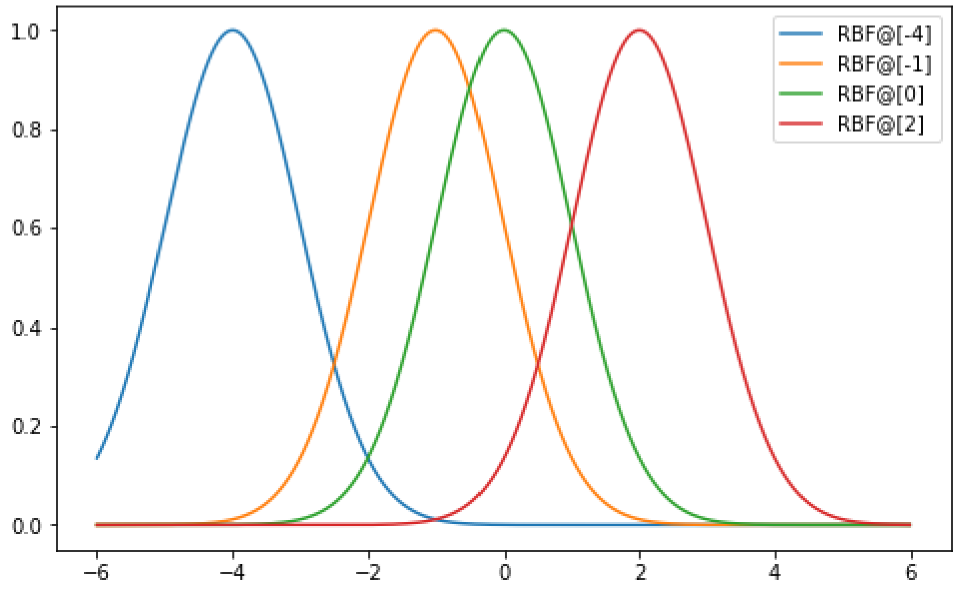


Q22(b)

In [5]:

```
# RBF kernel
```

```
plt.figure(figsize = (8,5))
y_RBF = RBF_kernel(prototypes, xpts, 1)
for i in range(len(prototypes)):
    label = "RBF@"+str(prototypes[i,:])
    plt.plot(xpts, y_RBF[i,:], label=label)

plt.legend(loc = 'best')
plt.show()
```



Q23

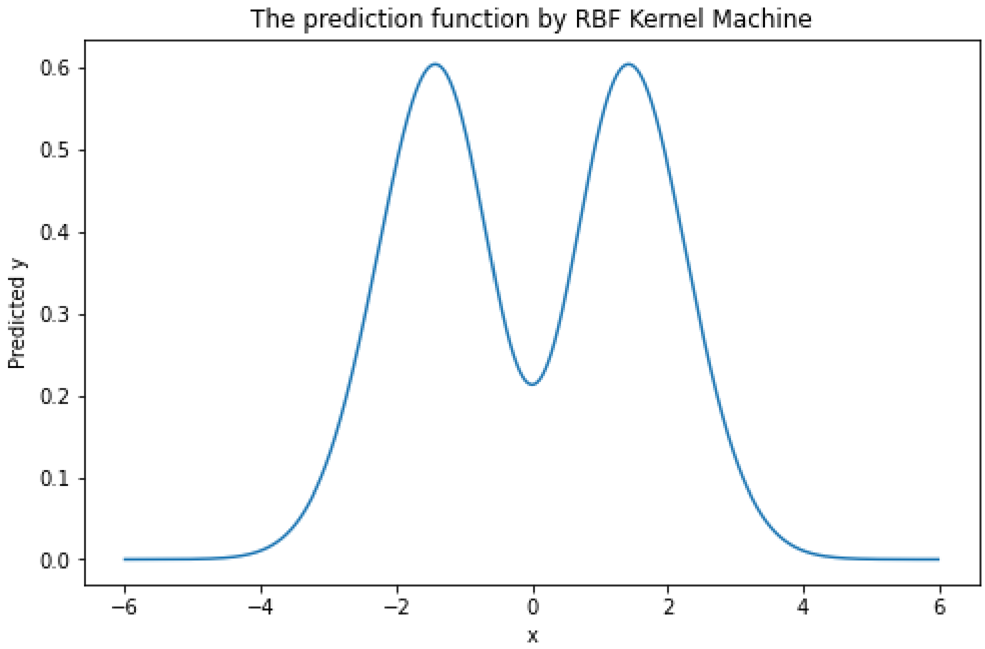
In [6]:

```
class Kernel_Machine(object):
    def __init__(self, kernel, training_points, weights):
        """
        Args:
            kernel(X1,X2) - a function return the cross-kernel matrix between rows of
            training_points - an nxd matrix with rows x_1,..., x_n
            weights - a vector of length n with entries alpha_1,...,alpha_n
        """

        self.kernel = kernel
        self.training_points = training_points
        self.weights = weights

    def predict(self, X):
        """
        Evaluates the kernel machine on the points given by the rows of X
        Args:
            X - an nxd matrix with inputs x_1,...,x_n in the rows
        Returns:
            Vector of kernel machine evaluations on the n points in X.  Specifically,
                Sum_{i=1}^R alpha_i k(x_j, mu_i)
        """
        # TODO
        K = self.kernel(self.training_points, X)
        return np.dot(K.T, self.weights)
```

In [7]:

```
from functools import partial

# Construct a Kernel Machine object with the RBF kernel
training_points = np.array([-1,0,1]).reshape(-1,1)
weights = np.array([1,-1,1])
```

```
kernel = partial(RBF_kernel,sigma=1)
machine = Kernel_Machine(kernel, training_points, weights)

xpts = np.arange(-6.0, 6, plot_step).reshape(-1,1)
y_pred = machine.predict(xpts)

#plot the resulting function
plt.figure(figsize = (8,5))
plot_step = .01
plt.plot(xpts, y_pred, label=label)
plt.title('The prediction function by RBF Kernel Machine')
plt.xlabel('x')
plt.ylabel('Predicted y')
plt.show()
```

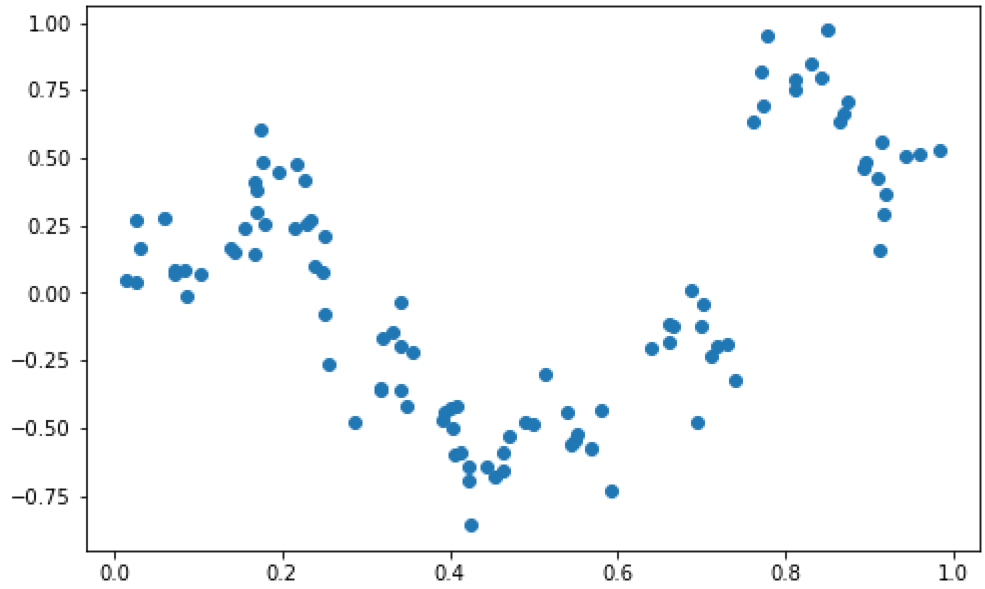


Kernel Ridge Regression: Practice

Q24

In [8]:
```
data_train,data_test = np.loadtxt("krr-train.txt"),np.loadtxt("krr-test.txt")
x_train, y_train = data_train[:,0].reshape(-1,1),data_train[:,1].reshape(-1,1)
x_test, y_test = data_test[:,0].reshape(-1,1),data_test[:,1].reshape(-1,1)
```

In [9]:
```
plt.figure(figsize=(8,5))
plt.plot(x_train, y_train, 'o')
plt.show()
```

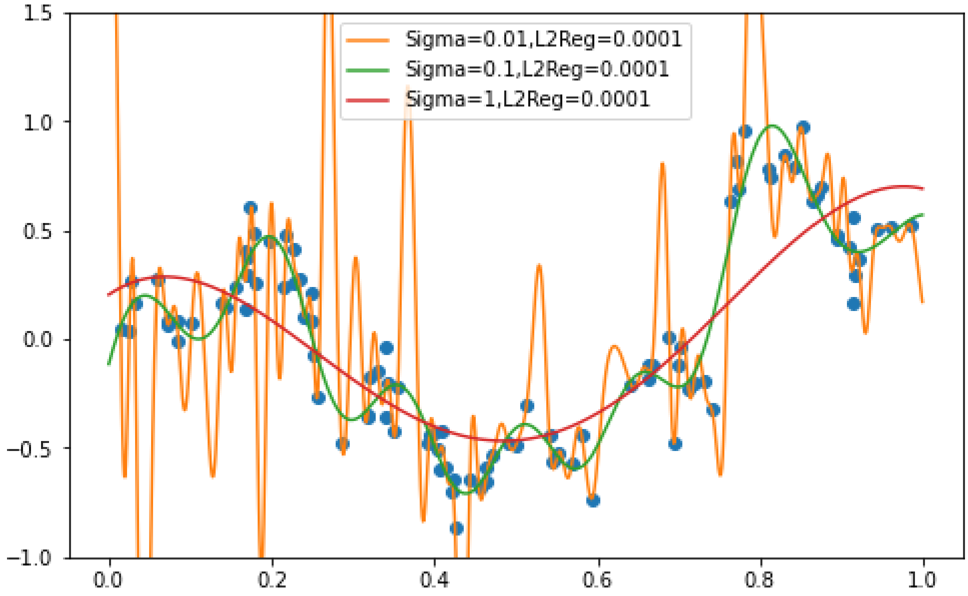


Q25

In [10]:
```
def train_kernel_ridge_regression(X, y, kernel, l2reg):
    # TODO
    K = kernel(X, X)
    n = K.shape[0]
    alpha = np.linalg.inv((np.identity(n)*l2reg+K)).dot(y)

    return Kernel_Machine(kernel, X, alpha)
```

Q26

In [11]:
```
plt.figure(figsize=(8,5))

plot_step = .001
xpts = np.arange(0 , 1, plot_step).reshape(-1,1)
plt.plot(x_train,y_train,'o')
l2reg = 0.0001
for sigma in [.01,.1,1]:
    k = functools.partial(RBF_kernel, sigma=sigma)
    f = train_kernel_ridge_regression(x_train, y_train, k, l2reg=l2reg)
    label = "Sigma="+str(sigma)+",L2Reg="+str(l2reg)
    plt.plot(xpts, f.predict(xpts), label=label)
plt.legend(loc = 'best')
plt.ylim(-1,1.5)
plt.show()
```

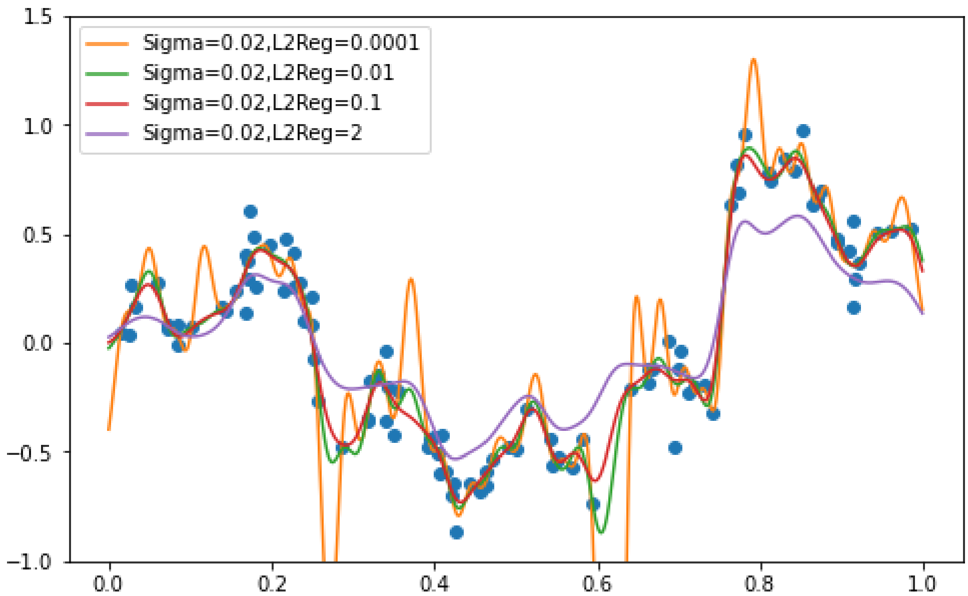


Small values of sigma(e.g. sigma=0.01) are more likely to overfit, while large sigmas(e.g. sigma=1) are less.

Q27

In [12]:

```
plt.figure(figsize=(8,5))

plot_step = .001
xpts = np.arange(0 , 1, plot_step).reshape(-1,1)
plt.plot(x_train,y_train,'o')
sigma= .02
for l2reg in [.0001,.01,.1,2]:
    k = functools.partial(RBF_kernel, sigma=sigma)
    f = train_kernel_ridge_regression(x_train, y_train, k, l2reg=l2reg)
    label = "Sigma="+str(sigma)+",L2Reg="+str(l2reg)
    plt.plot(xpts, f.predict(xpts), label=label)
plt.legend(loc = 'best')
plt.ylim(-1,1.5)
plt.show()
```



When λ goes up, the prediction fucntion becomes more flat, and as $\lambda \to \infty$, the prediction fucntion will be a constant function.

Q28

In [13]:

```
from sklearn.base import BaseEstimator, RegressorMixin, ClassifierMixin

class KernelRidgeRegression(BaseEstimator, RegressorMixin):
    """sklearn wrapper for our kernel ridge regression"""

    def __init__(self, kernel="RBF", sigma=1, degree=2, offset=1, l2reg=1):
        self.kernel = kernel
        self.sigma = sigma
        self.degree = degree
        self.offset = offset
        self.l2reg = l2reg

    def fit(self, X, y=None):
        """
        This should fit classifier. All the "work" should be done here.
        """
        if (self.kernel == "linear"):
            self.k = linear_kernel
        elif (self.kernel == "RBF"):
            self.k = functools.partial(RBF_kernel, sigma=self.sigma)
        elif (self.kernel == "polynomial"):
            self.k = functools.partial(polynomial_kernel, offset=self.offset, degree=
        else:
            raise ValueError('Unrecognized kernel type requested.')

        self.kernel_machine_ = train_kernel_ridge_regression(X, y, self.k, self.l2reg

        return self

    def predict(self, X, y=None):
        try:
            getattr(self, "kernel_machine_")
        except AttributeError:
            raise RuntimeError("You must train classifer before predicting data!")

        return(self.kernel_machine_.predict(X))

    def score(self, X, y=None):
        # get the average square error
        return(((self.predict(X)-y)**2).mean())
```

In [14]:

```
from sklearn.model_selection import GridSearchCV,PredefinedSplit
from sklearn.model_selection import ParameterGrid
from sklearn.metrics import mean_squared_error,make_scorer
import pandas as pd

test_fold = [-1]*len(x_train) + [0]*len(x_test)   #0 corresponds to test, -1 to train
predefined_split = PredefinedSplit(test_fold=test_fold)
```

In [15]:

```
param_grid = [{'kernel': ['RBF'],'sigma':[.1,1,10], 'l2reg': np.exp2(-np.arange(-5,5,
              {'kernel':['polynomial'],'offset':[-1,0,1], 'degree':[2,3,4],'l2reg':[1(
              {'kernel':['linear'],'l2reg': [10,1,.01]}]
kernel_ridge_regression_estimator = KernelRidgeRegression()
grid = GridSearchCV(kernel_ridge_regression_estimator,
                    param_grid,
                    cv = predefined_split,
                    scoring = make_scorer(mean_squared_error,greater_is_better = Fals
                    return_train_score=True
```

```
               )
grid.fit(np.vstack((x_train,x_test)),np.vstack((y_train,y_test)))
```

Out[15]:

```
GridSearchCV(cv=PredefinedSplit(test_fold=array([-1, -1, ...,  0,  0])),
             estimator=KernelRidgeRegression(),
             param_grid=[{'kernel': ['RBF'],
                          'l2reg': array([32.    , 16.    ,  8.    ,  4.    ,  2.    ,
1.    ,  0.5   ,
        0.25  ,  0.125 ,  0.0625]),
                          'sigma': [0.1, 1, 10]},
                         {'degree': [2, 3, 4], 'kernel': ['polynomial'],
                          'l2reg': [10, 0.1, 0.01], 'offset': [-1, 0, 1]},
                         {'kernel': ['linear'], 'l2reg': [10, 1, 0.01]}],
             return_train_score=True,
             scoring=make_scorer(mean_squared_error, greater_is_better=False))
```

In [16]:

```
pd.set_option('display.max_rows', None)
df = pd.DataFrame(grid.cv_results_)
# Flip sign of score back, because GridSearchCV likes to maximize,
# so it flips the sign of the score if "greater_is_better=FALSE"
df['mean_test_score'] = -df['mean_test_score']
df['mean_train_score'] = -df['mean_train_score']
cols_to_keep = ["param_degree", "param_kernel","param_l2reg" ,"param_offset","param_si
        "mean_test_score","mean_train_score"]
df_toshow = df[cols_to_keep].fillna('-')
```

Table for RBF kernels

In [17]:

```
# show the first ten rows order by ascending test scores
df_toshow_RBF = df_toshow[df_toshow['param_kernel']=='RBF']
df_toshow_RBF.sort_values(by=["mean_test_score"])[:10]
```

Out[17]:

	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_t
27	-	RBF	0.0625	-	0.1	0.021270	
24	-	RBF	0.1250	-	0.1	0.022885	
21	-	RBF	0.2500	-	0.1	0.024845	
18	-	RBF	0.5000	-	0.1	0.026609	
15	-	RBF	1.0000	-	0.1	0.027562	
12	-	RBF	2.0000	-	0.1	0.028041	
9	-	RBF	4.0000	-	0.1	0.030082	
6	-	RBF	8.0000	-	0.1	0.037650	
3	-	RBF	16.0000	-	0.1	0.055006	
28	-	RBF	0.0625	-	1	0.063632	

Table for polynomial kernels

In [18]:

```
# show the first ten rows order by ascending test scores
df_toshow_poly = df_toshow[df_toshow['param_kernel']=='polynomial']
df_toshow_poly.sort_values(by=["mean_test_score"])[:10]
```

Out[18]:

	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_t

	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_t
54	4	polynomial	0.01	-1	-	0.043454	
56	4	polynomial	0.01	1	-	0.060262	
33	2	polynomial	0.10	-1	-	0.065554	
38	2	polynomial	0.01	1	-	0.066532	
36	2	polynomial	0.01	-1	-	0.066915	
35	2	polynomial	0.10	1	-	0.067454	
44	3	polynomial	0.10	1	-	0.067508	
45	3	polynomial	0.01	-1	-	0.068156	
53	4	polynomial	0.10	1	-	0.068353	
42	3	polynomial	0.10	-1	-	0.068397	

Table for linear kernels

In [19]:

```
df_toshow_poly = df_toshow[df_toshow['param_kernel']=='linear']
df_toshow_poly.sort_values(by=["mean_test_score"])
```

Out[19]:

	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_t
58	-	linear	1.00	-	-	0.164540	
59	-	linear	0.01	-	-	0.164569	
57	-	linear	10.00	-	-	0.164591	

Best Settings

RBF: l2reg = 0.0625, sigma = 0.1

Polynomial: l2reg = 0.01, offset = -1, degree = 4

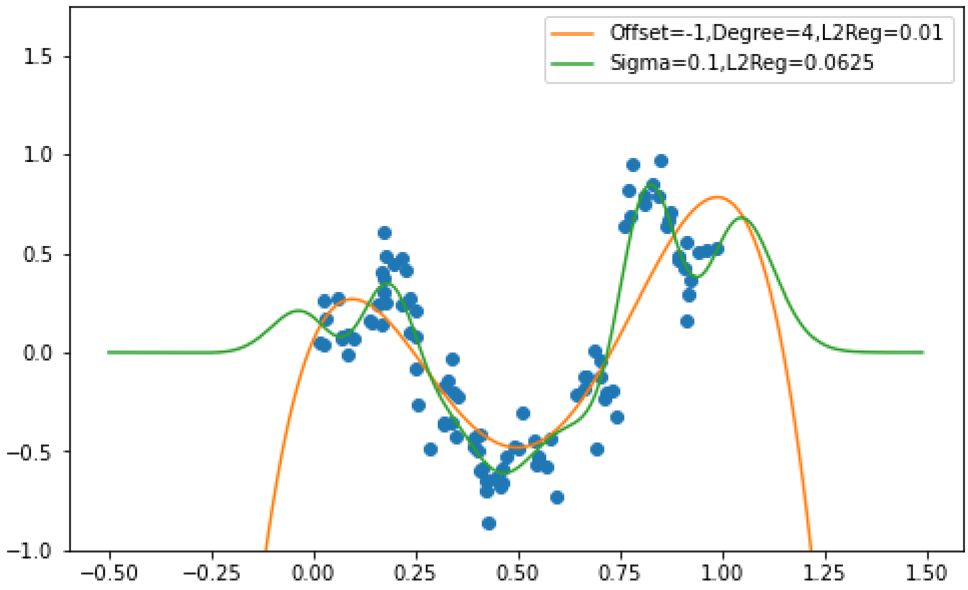
Linear: l2reg = 1

Q29

In [22]:

```
## Plot the best polynomial and RBF fits you found
plt.figure(figsize=(8,5))
plot_step = .01
xpts = np.arange(-.5 , 1.5, plot_step).reshape(-1,1)
plt.plot(x_train,y_train,'o')

#Plot best polynomial fit
offset= -1
degree = 4
l2reg = 0.01
k = functools.partial(polynomial_kernel, offset=offset, degree=degree)
f = train_kernel_ridge_regression(x_train, y_train, k, l2reg=l2reg)
label = "Offset="+str(offset)+",Degree="+str(degree)+",L2Reg="+str(l2reg)
plt.plot(xpts, f.predict(xpts), label=label)
```

```
#Plot best RBF fit
sigma = 0.1
l2reg= 0.0625
k = functools.partial(RBF_kernel, sigma=sigma)
f = train_kernel_ridge_regression(x_train, y_train, k, l2reg=l2reg)
label = "Sigma="+str(sigma)+",L2Reg="+str(l2reg)
plt.plot(xpts, f.predict(xpts), label=label)
plt.legend(loc = 'best')
plt.ylim(-1,1.75)
plt.show()
```



The prediction function trained with RBF kernel fits better on the test data than the one trained with polynomial kernel.

It is probably because the RBF kernel gives more flexibity to the prediction function while the polynomial kernel has a fixed degree that restricts its shape.

Q30

The Bayes decision function $f^*(x)$ is the best prediction fucntion we can get among all possible functions, and here y is generated by $y = f(x) + \epsilon$, where ϵ is independent of x.

Therefore, the best prediction for y is

$$\hat{y} = f(x)$$

That is, the Bayes decision function:

$$f^*(x) = f(x)$$

The Bayes risk is:

$$\begin{aligned} R(f^*) &= E[\ell(f^*(x), y)] \\ &= E[\ell(f(x), y)] \\ &= E[(f(x) - (f(x) + \epsilon))^2] \\ &= E(\epsilon^2) \\ &= var(\epsilon) + [E(\epsilon)]^2 \\ &= 0.1^2 + 0 \\ &= 0.01 \end{aligned}$$