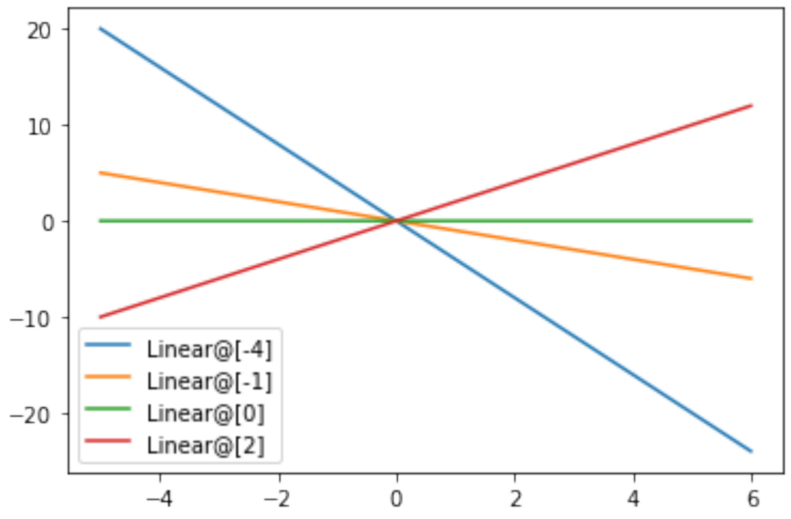
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import sklearn
import scipy.spatial
import functools

%matplotlib inline
```

Question 21

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/Ga
    Returns:
        matrix of size n1xn2, with exp(-||x1_i-x2_j||^2/(2 sigma^2)) i
    """
    #TODO
    mat = scipy.spatial.distance.cdist(X1,X2, metric='sqeuclidean')
    return np.exp(-mat/(2*sigma**2))

def polynomial_kernel(X1, X2, offset, degree):
    """
    Computes the inhomogeneous polynomial kernel between two sets of v
    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 po
    """
    return (offset + linear_kernel(X1,X2))**degree
```

Question 22

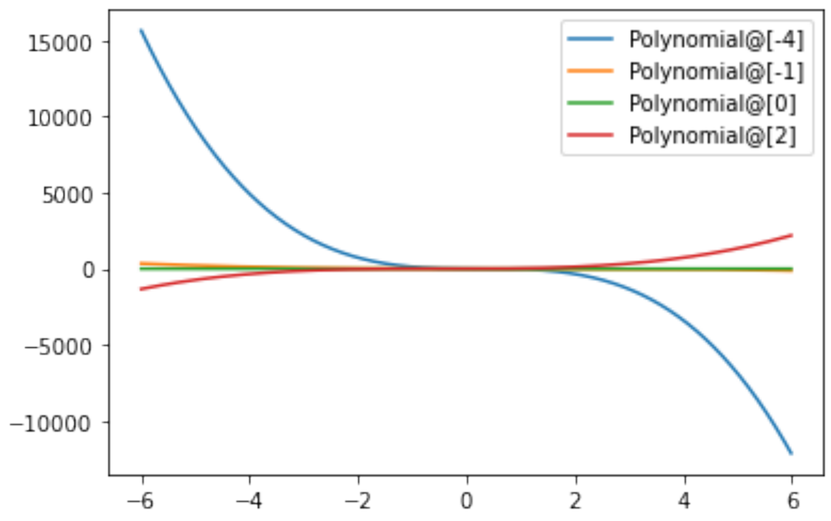
In [3]:
```
# PLot kernel machine functions
plot_step = .01
xpts = np.arange(-5.0, 6, plot_step).reshape(-1,1)
prototypes = np.array([-4,-1,0,2]).reshape(-1,1)

# Linear kernel
y = linear_kernel(prototypes, xpts)
for i in range(len(prototypes)):
    label = "Linear@"+str(prototypes[i,:])
    plt.plot(xpts, y[i,:], label=label)
plt.legend(loc = 'best')
plt.show()
```



Question 23

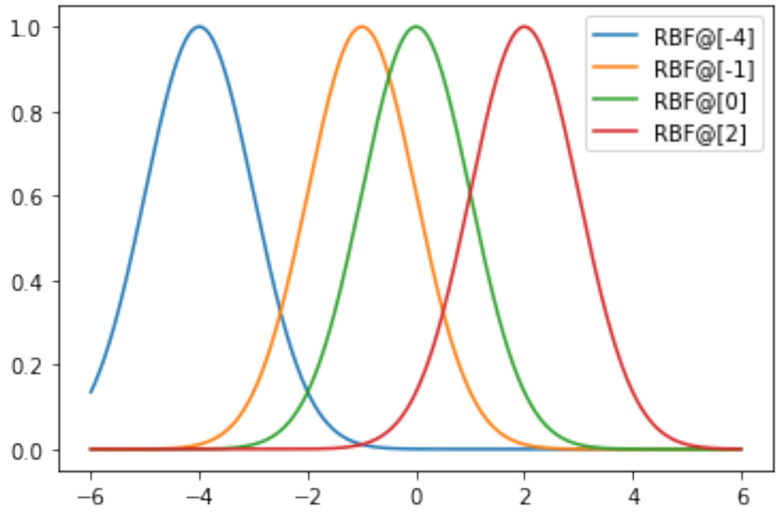
Part a)

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)

# Linear kernel
y = polynomial_kernel(prototypes, xpts,1,3)
for i in range(len(prototypes)):
    label = "Polynomial@"+str(prototypes[i,:])
    plt.plot(xpts, y[i,:], label=label)
plt.legend(loc = 'best')
plt.show()
```



Part b

In [5]:
```python
# 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)

# Linear kernel
y = RBF_kernel(prototypes, xpts, 1)
for i in range(len(prototypes)):
    label = "RBF@"+str(prototypes[i,:])
    plt.plot(xpts, y[i,:], label=label)
plt.legend(loc = 'best')
plt.show()
```



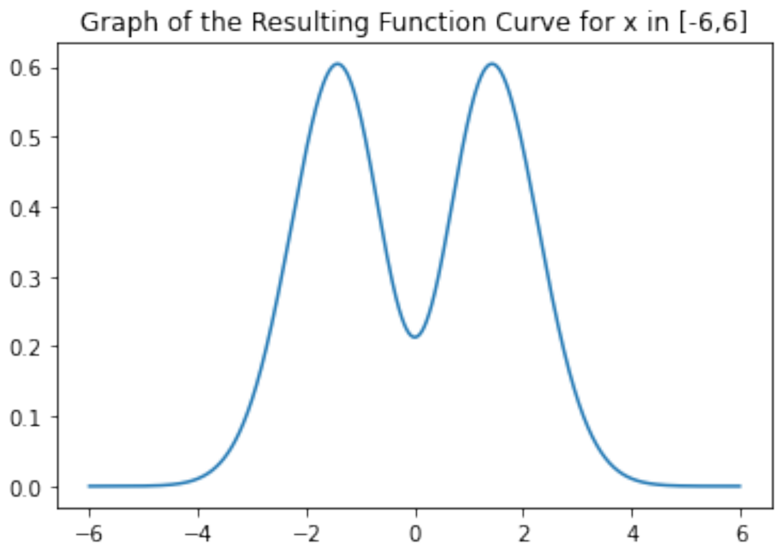
Question 24

In [6]:
```
class Kernel_Machine(object):
    def __init__(self, kernel, training_points, weights):
        """
        Args:
            kernel(X1,X2) - a function return the cross-kernel matrix
            training_points - an nxd matrix with rows x_1,..., x_n
            weights - a vector of length n with entries alpha_1,...,al
        """

        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 c
        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.
                Sum_{i=1}^R alpha_i k(x_j, mu_i)
        """

        kernel_matrix = self.kernel(X, self.training_points)
        return (kernel_matrix @ self.weights)
```

In [7]:
```
k = functools.partial(RBF_kernel, sigma=1)
train_points = np.array([-1,0,1]).reshape(-1,1)
weights = np.array([1,-1,1]).reshape(-1,1)
X = np.array([-4,-1,0,1]).reshape(-1,1)
test_kernel_object = Kernel_Machine(k, train_points, weights)
test_kernel_object.predict(X)
```

Out[7]:
```
array([[0.01077726],
       [0.52880462],
       [0.21306132],
       [0.52880462]])
```

In [8]:
```
# PLot kernel machine functions
plot_step = .01
xpts = np.arange(-6.0, 6, plot_step).reshape(-1,1)
# Linear kernel
y = test_kernel_object.predict(xpts).reshape(-1,1)

plt.plot(xpts, y)
plt.title('Graph of the Resulting Function Curve for x in [-6,6]')
plt.show()
```

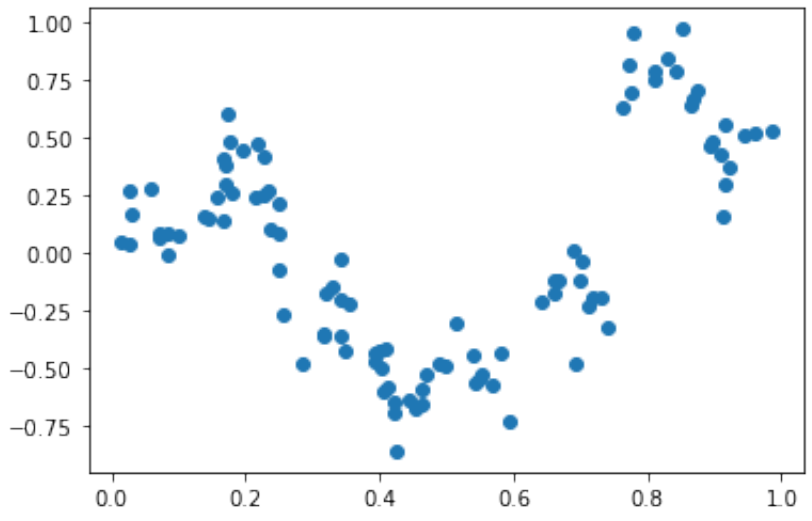


Load train & test data; Convert to column vectors so it generalizes well to data in higher dimensions.

Question 25

In [9]:
```
data_train,data_test = np.loadtxt("krr-train.txt"),np.loadtxt("krr-tes
x_train, y_train = data_train[:,0].reshape(-1,1),data_train[:,1].resha
x_test, y_test = data_test[:,0].reshape(-1,1),data_test[:,1].reshape(-
plt.scatter(x_train,y_train)
```

Out[9]:
```
<matplotlib.collections.PathCollection at 0x7fc8315edf10>
```



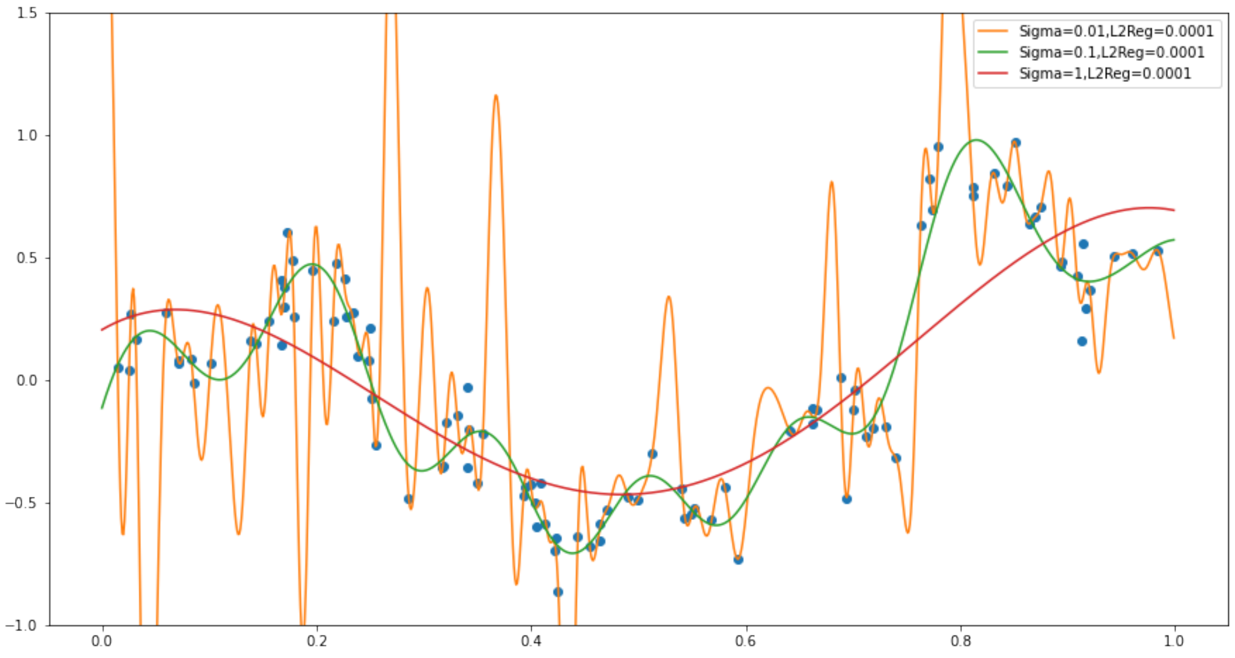
Looks like there is not a linear relationship, but rather a potential polynomial, or sinusoidal relationship, alternatively it could be a uniform + noise distribution.

Question 26

In [10]:
```
def train_kernel_ridge_regression(X, y, kernel, l2reg):
    kernel_matrix = kernel(X,X)
    matrix = ((np.identity(X.shape[0])*l2reg)+kernel_matrix)
    alpha = np.linalg.inv(matrix)@y
    return Kernel_Machine(kernel, X, alpha)
```

Question 27

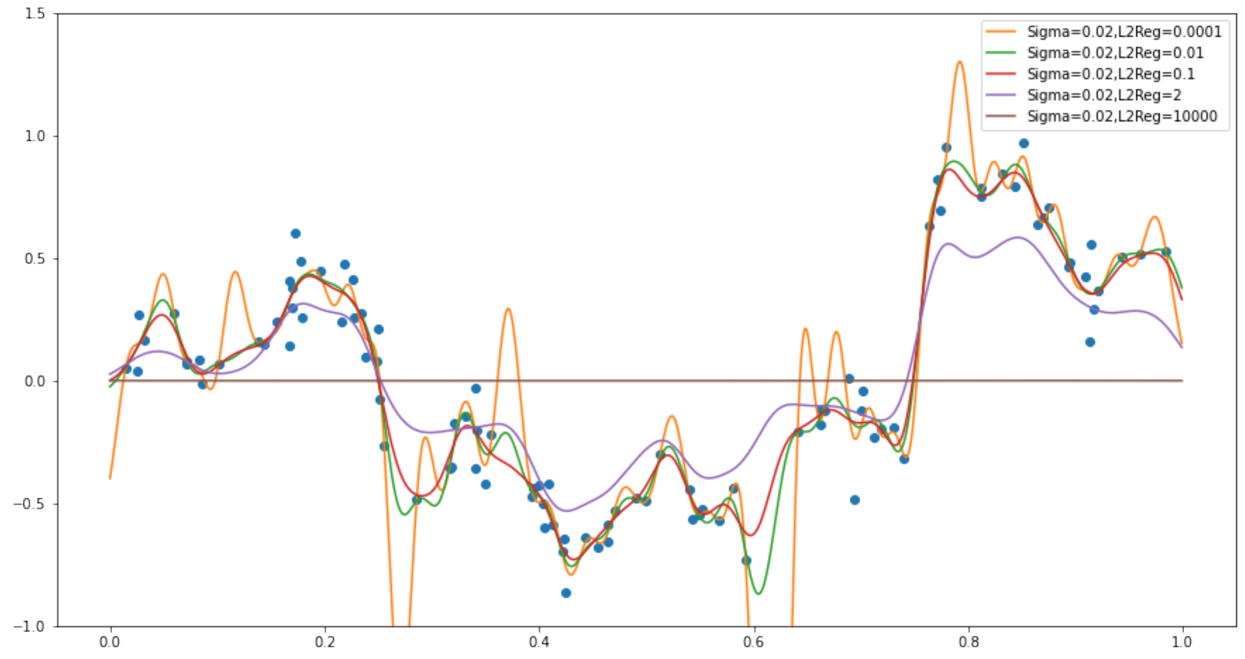
In [11]:
```
plt.figure(figsize=(15,8))
plot_step = .001
xpts = np.arange(0 , 1, plot_step).reshape(-1,1)
plt.plot(x_train,y_train,'o')
l2reg = 0.0001
for sigma in [.01,.1,1]:
    k = functools.partial(RBF_kernel, sigma=sigma)
    f = train_kernel_ridge_regression(x_train, y_train, k, l2reg=l2reg
    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()
```



The smaller values of sigma, $\sigma \in 0.01$ for instance, the curve overfits the training data, and does not generalize well (least bias). With a lower sigma, say $\sigma \in 1$ the curve is much smoother with lower amplitude, and still does not fit the training data well (high bias). With $\sigma = .1$ we have the best fit of the curve.

Problem 28

In [12]:
```
plt.figure(figsize=(15,8))
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,.1,2,10000]:
    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()
```



As $\lambda \rightarrow \infty$ the curve loses its amplitude, becomes less volatile, and becomes a line. Specifically, its the line that fits all points with least euclidean distance loss, which in this case approaches 0.

In [13]:
```
from sklearn.base import BaseEstimator, RegressorMixin, ClassifierMixi

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

    def __init__(self, kernel="RBF", sigma=1, degree=2, offset=1, l2re
        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=self
        else:
            raise ValueError('Unrecognized kernel type requested.')

        self.kernel_machine_ = train_kernel_ridge_regression(X, y, sel

        return self

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

        return(self.kernel_machine_.predict(X))

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

Question 29

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 te
predefined_split = PredefinedSplit(test_fold=test_fold)
```

In [15]:
```
param_grid = [{'kernel': ['RBF'],'sigma':[0.05, 0.055, 0.06],
               'l2reg': [0.271, 0.27, 0.269]},
              {'kernel':['polynomial'],'offset':[1.75, 1.8, 1.85],
               'degree':[5,6,7],'l2reg':[0.033, 0.034, 0.035]},
              {'kernel':['linear'],'l2reg': [3.2, 4, 4.5]}]
kernel_ridge_regression_estimator = KernelRidgeRegression()
grid = GridSearchCV(kernel_ridge_regression_estimator,
                    param_grid,
                    cv = predefined_split,
                    scoring = make_scorer(mean_squared_error,greater_i
                    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': [0.271, 0.27, 0
.269],
                          'sigma': [0.05, 0.055, 0.06]},
                         {'degree': [5, 6, 7], 'kernel': ['polynomial
'],
                          'l2reg': [0.033, 0.034, 0.035],
                          'offset': [1.75, 1.8, 1.85]},
                         {'kernel': ['linear'], 'l2reg': [3.2, 4, 4.5
]}],
             return_train_score=True,
             scoring=make_scorer(mean_squared_error, greater_is_bette
r=False))
```

In [16]:
```
pd.set_option('display.max_rows', 20)
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_sigma",
        "mean_test_score","mean_train_score"]
df_toshow = df[cols_to_keep].fillna('-')
df_toshow.sort_values(by=["mean_test_score"])
```

Out[16]:

	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	ı
1	-	RBF	0.271	-	0.055	0.013821	
4	-	RBF	0.270	-	0.055	0.013821	
7	-	RBF	0.269	-	0.055	0.013821	
8	-	RBF	0.269	-	0.06	0.013979	
5	-	RBF	0.270	-	0.06	0.013982	
...	...	...	...	...	...	...	
12	5	polynomial	0.034	1.75	-	0.046631	
15	5	polynomial	0.035	1.75	-	0.046974	
37	-	linear	4.000	-	-	0.164510	
38	-	linear	4.500	-	-	0.164511	
36	-	linear	3.200	-	-	0.164511	

39 rows × 7 columns

In [17]:
```
rbf = df_toshow[df_toshow['param_kernel']=='RBF']
rbf_min = rbf['mean_test_score'].min()
rbf_min_row = rbf[rbf['mean_test_score']==rbf_min]
print('Best parameters for RBF kernel:')
rbf_min_row
```

```
Best parameters for RBF kernel:
```

Out[17]:

	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	m
1	-	RBF	0.271	-	0.055	0.013821	

In [18]:
```
polynomial = df_toshow[df_toshow['param_kernel']=='polynomial']
poly_min = polynomial['mean_test_score'].min()
poly_min_row = polynomial[polynomial['mean_test_score']==poly_min]
print('Best parameters for polynomial kernel:')
poly_min_row
```

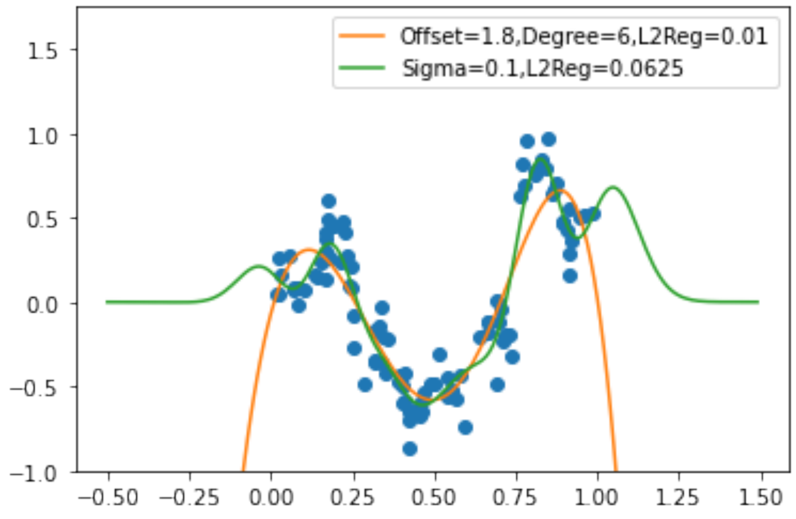
```
Best parameters for polynomial kernel:
```

Out[18]:

	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score
22	6	polynomial	0.034	1.8	-	0.032405

Question 30

In [19]:
```python
## Plot the best polynomial and RBF fits you found
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= 1.8
degree = 6
l2reg = .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(l2r
plt.plot(xpts, f.predict(xpts), label=label)
#Plot best RBF fit
sigma = .1
l2reg= .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 best hyperparameters for the polynomial kernel were $Offset = 1.8,\ Degree = 6,\ l2reg = .034$ and for the RBF kernel the best parameters were $\sigma = 0.055,\ l2reg = 0.271$. Both curves seem to fit the graph reasonably well, though the RBF performed better with a lower min mean_test_score ($RBF = 0.013821, Polynomial = 0.032405$). This could be because the best fitting polynomial curve could not capture the intricacies of the data without increasing its degree substantially, at the risk of over fitting on the train set, while the RBF kernel with its smoother curves could actually capture structure within the data that would generalize well.

Question 31

Bayes function is defined in the following way:

$$E(y|x) = E(f(x) + \epsilon|x) \rightarrow E(f(x)|x) + E(\epsilon|x) = E(f(x))$$

We can now calculate the risk of the bayes decision function:

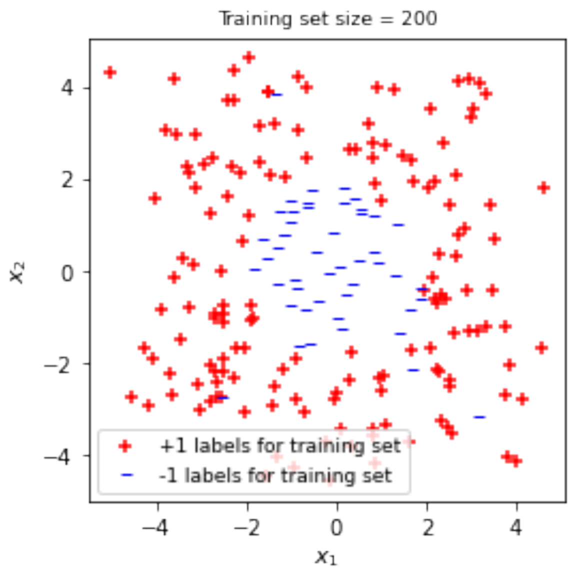
$$E((f(x) - y)^2) = E((f(x) - f(x) + epsilon)^2) = E(\epsilon^2) = .1^2$$

As

$$Var(\epsilon) = E(\epsilon^2) - E^2(\epsilon) = E(\epsilon^2) - 0 = E(\epsilon^2)$$

Question 32

In [20]:
```
# Load and plot the SVM data
#load the training and test sets
data_train,data_test = np.loadtxt("svm-train.txt"),np.loadtxt("svm-tes
x_train, y_train = data_train[:,0:2], data_train[:,2].reshape(-1,1)
x_test, y_test = data_test[:,0:2], data_test[:,2].reshape(-1,1)

#determine predictions for the training set
yplus = np.ma.masked_where(y_train[:,0]<=0, y_train[:,0])
xplus = x_train[~np.array(yplus.mask)]
yminus = np.ma.masked_where(y_train[:,0]>0, y_train[:,0])
xminus = x_train[~np.array(yminus.mask)]

#plot the predictions for the training set
figsize = plt.figaspect(1)
f, (ax) = plt.subplots(1, 1, figsize=figsize)

pluses = ax.scatter (xplus[:,0], xplus[:,1], marker='+', c='r',
                     label = '+1 labels for training set')
minuses = ax.scatter (xminus[:,0], xminus[:,1], marker=r'$-$',
                      c='b', label = '-1 labels for training set')

ax.set_ylabel(r"$x_2$", fontsize=11)
ax.set_xlabel(r"$x_1$", fontsize=11)
ax.set_title('Training set size = %s'% len(data_train), fontsize=9)
ax.axis('tight')
ax.legend(handles=[pluses, minuses], fontsize=9)
plt.show()
```



The data appears that it could be seperated by a quadratic boundary, (imagine a circle in the middle classifying the points). Additionally, a RBF boundary could work as well, as there is a circular distribution and you could imagine a guassian kernel coming out (as in the height extends to the z dimension) of the origin of the graph and decreasing otherwise.

Question 33

```
In [21]: class train_soft_svm():
             def __init__(self, x_train, y_train, y_test, k, lambda_reg, epochs
                 self.x_train = x_train
                 self.y_train = y_train
                 self.y_test = y_test
                 self.k = k
                 self.lambda_reg = lambda_reg
                 self.epochs = epochs
                 self.alpha = None

             def predict(self, values):
                 return self.k(values,self.x_train) @ self.alpha

             def fit(self):
                 #Initialize helper variables
                 alpha = np.zeros(self.x_train.shape[0])
                 epoch, t = 0, 2

                 #Iterate over the epochs
                 while epoch < self.epochs:
                     for i in range(len(self.x_train)):

                         alpha = alpha * (1-(1/t))
                         y_hat = self.k(self.x_train[i].reshape(1,2),self.x_tra
                         value = self.y_train[i] * y_hat

                         #If we have a missclassification, subtract the second
                         if value < 1:
                             step = y_train[i]/(t*self.lambda_reg)
                             alpha[i]+=1*step
                         t += 1
                     #Increment epoch counter variable
                     epoch += 1
                 self.alpha = alpha
                 return True

             def classification_error(self,y_hats):
                 error = 0
                 for i in range(len(y_hats)):
                     if y_hats[i] >= 0 and self.y_test[i] != 1:
                         error += 1
                     elif y_hats[i] < 0 and self.y_test[i] != -1:
                         error += 1
                 return error / len(y_hats)
```

Question 34

Took a comprehensive iterative approach, started ide then honed in by restriciting the range I was searching over to get the best hyper parameters

In [22]:
```
sigmas = np.arange(-3,0,.1)
lambda_regs = np.logspace(-5,-2, num=40)
rbf_test_accuracy = []
rbf_lambda_reg = []
rbf_sigma_list = []
for sigma in sigmas:
    for lambda_reg in lambda_regs:
        k = functools.partial(RBF_kernel, sigma=sigma)
        f = train_soft_svm(x_train, y_train, y_test, k, lambda_reg,20)
        f.fit()
        y_bar = f.predict(x_test)
        rbf_test_accuracy.append(f.classification_error(y_bar))
        rbf_lambda_reg.append(lambda_reg)
        rbf_sigma_list.append(sigma)

index = rbf_test_accuracy.index(min(rbf_test_accuracy))
print("Best min mean test error at",rbf_test_accuracy[index]*100,
      "% with sigma = ",rbf_sigma_list[index],
      "lambda =", rbf_lambda_reg[index])
```

```
Best min mean test error at 3.375 % with sigma =  -0.9999999999999982
lambda = 0.0004124626382901352
```

In [23]:
```
degrees = np.arange(1,5,1)
offsets = np.arange(1,10,20)
lambda_regs = np.logspace(-5,-3, num=40)
poly_offset_list = []
poly_lambda_list = []
poly_degree_list = []
polynomial_test_accuracy = []
for degree in degrees:
    for offset in offsets:
        for lambda_reg in lambda_regs:
            k = functools.partial(polynomial_kernel, offset=offset, de
            f = train_soft_svm(x_train, y_train, y_test, k, lambda_reg
            f.fit()
            y_bar = f.predict(x_test)
            polynomial_test_accuracy.append(f.classification_error(y_b
            poly_lambda_list.append(lambda_reg)
            poly_offset_list.append(offset)
            poly_degree_list.append(degree)

index = polynomial_test_accuracy.index(min(polynomial_test_accuracy))
print("Best min mean test error at",polynomial_test_accuracy[index]*10
      "% with offset = ",poly_offset_list[index],
      "degree = ", poly_degree_list[index],
      "lambda =", poly_lambda_list[index])
```

```
Best min mean test error at 5.625 % with offset =  1 degree =  2 lamb
da = 0.0004375479375074184
```

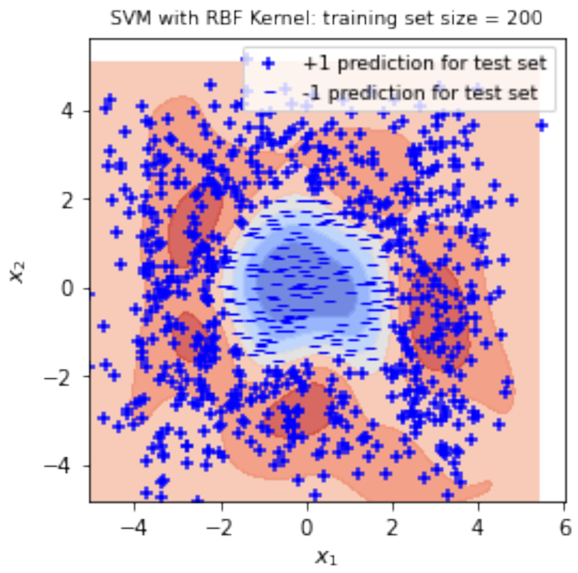
Question 35

RBF Kernel, Optimal Fit, $\sigma = -.99999$, $\lambda = .00412$

In [24]:
```
# Code to help plot the decision regions
# (Note: This ode isn't necessarily entirely appropriate for the quest
So think about what you are doing.)

sigma=1
k = functools.partial(RBF_kernel, sigma=-0.9999999999999982)
f = train_soft_svm(x_train, y_train, y_test, k, 0.0004124626382901352,
f.fit()
#determine the decision regions for the predictions
x1_min = min(x_test[:,0])
x1_max= max(x_test[:,0])
x2_min = min(x_test[:,1])
x2_max= max(x_test[:,1])
h=0.1
```

```
xx, yy = np.meshgrid(np.arange(x1_min, x1_max, h),
                     np.arange(x2_min, x2_max, h))

Z = f.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

#determine the predictions for the test set
y_bar = f.predict(x_test)
yplus = np.ma.masked_where(y_bar<=0, y_bar)
xplus = x_test[~np.array(yplus.mask)]
yminus = np.ma.masked_where(y_bar>0, y_bar)
xminus = x_test[~np.array(yminus.mask)]

print('Classification Error:',f.classification_error(y_bar)*100, '%')
#plot the learned boundary and the predictions for the test set
figsize = plt.figaspect(1)
f, (ax) = plt.subplots(1, 1, figsize=figsize)
decision =ax.contourf(xx, yy, Z, cmap=plt.cm.coolwarm, alpha=0.8)
pluses = ax.scatter (xplus[:,0], xplus[:,1], marker='+', c='b',
                     label = '+1 prediction for test set')
minuses = ax.scatter (xminus[:,0], xminus[:,1], marker=r'$-$',
                      c='b', label = '-1 prediction for test set')
ax.set_ylabel(r"$x_2$", fontsize=11)
ax.set_xlabel(r"$x_1$", fontsize=11)
ax.set_title('SVM with RBF Kernel: training set size = %s'% len(data_t
ax.axis('tight')
ax.legend(handles=[pluses, minuses], fontsize=9)
plt.show()
```

```
Classification Error: 3.375 %
```

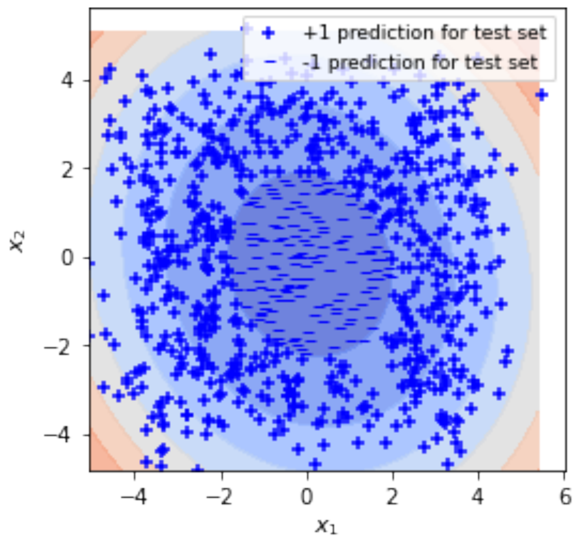


Polynomial Kernel, Optimal Fit, $Offset = 1$, $Degree = 2$, $\lambda = 0.00043754793750741\,84$

In [25]:
```
# Code to help plot the decision regions
# (Note: This ode isn't necessarily entirely appropriate for
the questions asked. So think about what you are doing.)

k = functools.partial(polynomial_kernel, offset=1,degree=2)
f = train_soft_svm(x_train, y_train, y_test, k, 0.00043754793750741 84,
f.fit()
#determine the decision regions for the predictions
x1_min = min(x_test[:,0])
x1_max= max(x_test[:,0])
x2_min = min(x_test[:,1])
x2_max= max(x_test[:,1])
h=0.1
xx, yy = np.meshgrid(np.arange(x1_min, x1_max, h),
                     np.arange(x2_min, x2_max, h))

Z = f.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

#determine the predictions for the test set
y_bar = f.predict(x_test)
yplus = np.ma.masked_where(y_bar<=0, y_bar)
xplus = x_test[~np.array(yplus.mask)]
yminus = np.ma.masked_where(y_bar>0, y_bar)
xminus = x_test[~np.array(yminus.mask)]

print('Classification Error:',f.classification_error(y_bar)*100, '%')
#plot the learned boundary and the predictions for the test set
figsize = plt.figaspect(1)
f, (ax) = plt.subplots(1, 1, figsize=figsize)
decision =ax.contourf(xx, yy, Z, cmap=plt.cm.coolwarm, alpha=0.8)
pluses = ax.scatter (xplus[:,0], xplus[:,1], marker='+', c='b',
                     label = '+1 prediction for test set')
minuses = ax.scatter (xminus[:,0], xminus[:,1], marker=r'$-$',
                      c='b', label = '-1 prediction for test set')
ax.set_ylabel(r"$x_2$", fontsize=11)
ax.set_xlabel(r"$x_1$", fontsize=11)
ax.set_title('SVM with RBF Kernel: training set size = %s'% len(data_t
ax.axis('tight')
ax.legend(handles=[pluses, minuses], fontsize=9)
plt.show()
```

```
Classification Error: 5.625 %
```

SVM with RBF Kernel: training set size = 200

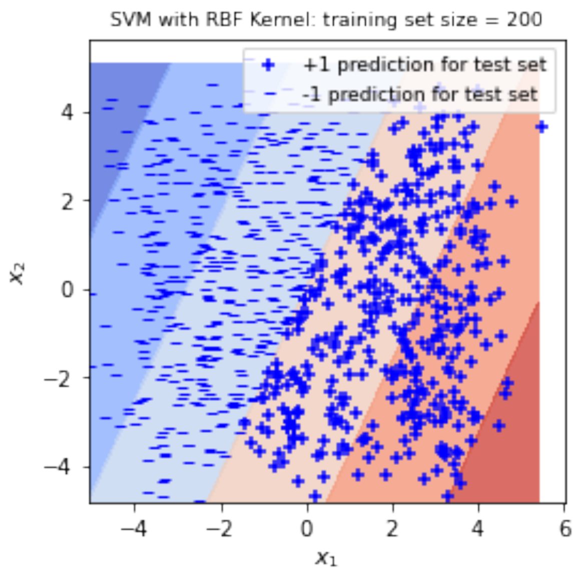


Linear Kernel, Optimal Fit

In [26]:
```
# Code to help plot the decision regions
# (Note: This ode isn't necessarily entirely appropriate for the quest

sigma=1
k = functools.partial(linear_kernel)
f = train_soft_svm(x_train, y_train, y_test, k, 1e-05,50)
f.fit()
#determine the decision regions for the predictions
x1_min = min(x_test[:,0])
x1_max= max(x_test[:,0])
x2_min = min(x_test[:,1])
x2_max= max(x_test[:,1])
h=0.1
xx, yy = np.meshgrid(np.arange(x1_min, x1_max, h),
                     np.arange(x2_min, x2_max, h))

Z = f.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

#determine the predictions for the test set
y_bar = f.predict(x_test)
yplus = np.ma.masked_where(y_bar<=0, y_bar)
xplus = x_test[~np.array(yplus.mask)]
yminus = np.ma.masked_where(y_bar>0, y_bar)
xminus = x_test[~np.array(yminus.mask)]

print('Classification Error:',f.classification_error(y_bar)*100, '%')
#plot the learned boundary and the predictions for the test set
figsize = plt.figaspect(1)
f, (ax) = plt.subplots(1, 1, figsize=figsize)
```

```
decision =ax.contourf(xx, yy, Z, cmap=plt.cm.coolwarm, alpha=0.8)
pluses = ax.scatter (xplus[:,0], xplus[:,1], marker='+', c='b',
                     label = '+1 prediction for test set')
minuses = ax.scatter (xminus[:,0], xminus[:,1], marker=r'$-$',
                      c='b', label = '-1 prediction for test set')
ax.set_ylabel(r"$x_2$", fontsize=11)
ax.set_xlabel(r"$x_1$", fontsize=11)
ax.set_title('SVM with RBF Kernel: training set size = %s'% len(data_t
ax.axis('tight')
ax.legend(handles=[pluses, minuses], fontsize=9)
plt.show()
```

```
Classification Error: 49.875 %
```



```
In [ ]:
```

In [1]:
```
import os
import numpy as np
import random
import time
import matplotlib.pyplot as plt
import pandas as pd
def folder_list(path,label):
    '''
    PARAMETER PATH IS THE PATH OF YOUR LOCAL FOLDER
    '''
    filelist = os.listdir(path)
    review = []
    for infile in filelist:
        file = os.path.join(path,infile)
        r = read_data(file)
        r.append(label)
        review.append(r)
    return review

def read_data(file):
    '''
    Read each file into a list of strings.
    Example:
    ["it's", 'a', 'curious', 'thing', "i've", 'found', 'that', 'when',
    ...'to', 'carry', 'the', 'whole', 'movie', "he's", 'much', 'better
    '''
    f = open(file)
    lines = f.read().split(' ')
    symbols = '${}()[].,:;+-*/&|<>=~" '
    words = map(lambda Element: Element.translate(str.maketrans("", ""
    words = filter(None, words)
    return list(words)


def load_and_shuffle_data():
    '''
    pos_path is where you save positive review data.
    neg_path is where you save negative review 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)
    return review
```

```
# Taken from http://web.stanford.edu/class/cs221/ Assignment #2 Suppor
def dotProduct(d1, d2):
    """
    @param dict d1: a feature vector represented by a mapping from a f
    @param dict d2: same as d1
    @return float: the dot product between d1 and d2
    """
    if len(d1) < len(d2):
        return dotProduct(d2, d1)
    else:
        return sum(d1.get(f, 0) * v for f, v in d2.items())

def increment(d1, scale, d2):
    """
    Implements d1 += scale * d2 for sparse vectors.
    @param dict d1: the feature vector which is mutated.
    @param float scale
    @param dict d2: a feature vector.

    NOTE: This function does not return anything, but rather
    increments d1 in place. We do this because it is much faster to
    change elements of d1 in place than to build a new dictionary and
    return it.
    """
    for f, v in d2.items():
        d1[f] = d1.get(f, 0) + v * scale
```

In [2]:
```
data = load_and_shuffle_data()
```

Question 6

In [3]:
```
#Create bag of words function
def bag_of_words_func(words):
    """
    Inputs:
    words: (list) a list of words

    Output:
    bag_of_words: (dictionary) Key: Word in words - Value: Count of Wo
    """
    bag_of_words = {}
    #Exclude the last character as it is the label we need to predict
    for word in words[:-1]:
        bag_of_words[word] = bag_of_words.get(word,0) + 1

    #Return our bag of words
    return bag_of_words
```

Question 7

In [4]:
```
#Grab the first 1500 Reviews / Labels for training
X_train = [bag_of_words_func(f) for f in data[:1500]]
y_train = [f[-1] for f in data[:1500]]

#Grab 500 more for Testing
X_test = [bag_of_words_func(f) for f in data[1500:2000]]
y_test = [f[-1] for f in data[1500:2000]]
```

Question 8

In [5]:
```
def pegasos(x,y, lambda_reg, total_epochs):
    """
    Input:
    x: (dictionary) review data that has been manipulated into sparse
    y: (list) label values of ith position of data at x_i
    lambda_reg
    total_epochs: (int):

    Output:
    w: dictionary of key value pairs, key: word in review, value: floa
    """

    #Initialize helper variables
    w = dict()
    epoch, t = 0, 0

    while epoch < total_epochs:
        for review in range(len(x)):

            #Update counter variable
            t += 1
            #Update step size
            step_size_t = 1/(lambda_reg*t)

            #Scale w variables
            increment(w, -step_size_t*lambda_reg, w)

            #If we have a missclassification, subtract the second port
            if y[review]*dotProduct(w,x[review]) < 1:
                increment(w, step_size_t*y[review], x[review])
        #Increment epoch counter variable
        epoch += 1

    return w
```

In [6]:
```
def fast_pegasos_algo(x,y,lambda_reg,total_epochs):
    """
    Input:
    x: (dictionary) review data that has been manipulated into sparse
    y: (list) label values of ith position of data at x_i
    lambda_reg
    total_epochs: (int):

    Output:
    w: dictionary of key value pairs, key: word in review, value: floa
    """

    #Initialize helper variables
    W = dict()
    epoch, t, s = 0, 1, 1

    while epoch < total_epochs:
        for review in range(len(x)):

            #Update counter variable
            t += 1
            #Update step size
            step_size_t = 1/(lambda_reg*t)

            #Update s
            s += (s * lambda_reg * -step_size_t)
            #If we have a missclassification, subtract the second port
            if y[review]*dotProduct(W,x[review])*s < 1:
                increment(W, (1/s)*step_size_t*y[review], x[review])
        #Increment epoch counter variable
        epoch += 1
    W.update((x,y*s) for x,y in W.items())
    return W
```

Problem 10

In [7]:
```
lambda_reg = .1
epochs = 6
```

In [8]:
```
#Time test each approach
start = time.time()
w_slow = pegasos(X_train,y_train,lambda_reg, epochs)
end = time.time()
print('Slow pegasos algorithm run speed:', end-start)

start = time.time()
w_fast = fast_pegasos_algo(X_train,y_train,lambda_reg, epochs)
end = time.time()
print('Fast pegasos algorithm run speed:', end-start)
```

```
Slow pegasos algorithm run speed: 44.98774600028992
Fast pegasos algorithm run speed: 0.7436118125915527
```

Problem 11

In [9]:
```
def classification_error(x,y,w):
    total_error = 0

    #Iterate over data
    for row in range(len(x)):
        #Get prediction
        if dotProduct(x[row],w)<0:
            y_hat = -1
        else:
            y_hat = 1

        if y_hat != y[row]:
            total_error += 1
    return total_error / len(y)
```

In [10]:
```
print('Slow loss:', classification_error(X_test,y_test,w_slow) , 'Fast
```

```
Slow loss: 0.304 Fast loss: 0.364
```

Problem 12

In [ ]:
```
error_list = list()
lambda_list = list()
lambda_regs = np.logspace(-3,-1,num=20)
for lambda_reg in lambda_regs:
    w_fast = fast_pegasos_algo(X_train,y_train, lambda_reg, 50)
    error_list.append(classification_error(X_test,y_test,w_fast))
    lambda_list.append(lambda_reg)
```

In [ ]:
```
min_error_lambda = lambda_list[error_list.index(min(error_list))]
```

In [ ]:
```
plt.figure(figsize=(10,8))
plt.scatter(lambda_list,error_list)
plt.xscale('log')
plt.title('Percentage Test Error by Variable Lambda')
```

Min error is achieved at $Percent\ Loss = 15.6\%$, with $\lambda = 0.007228703350949573$

Problem 13

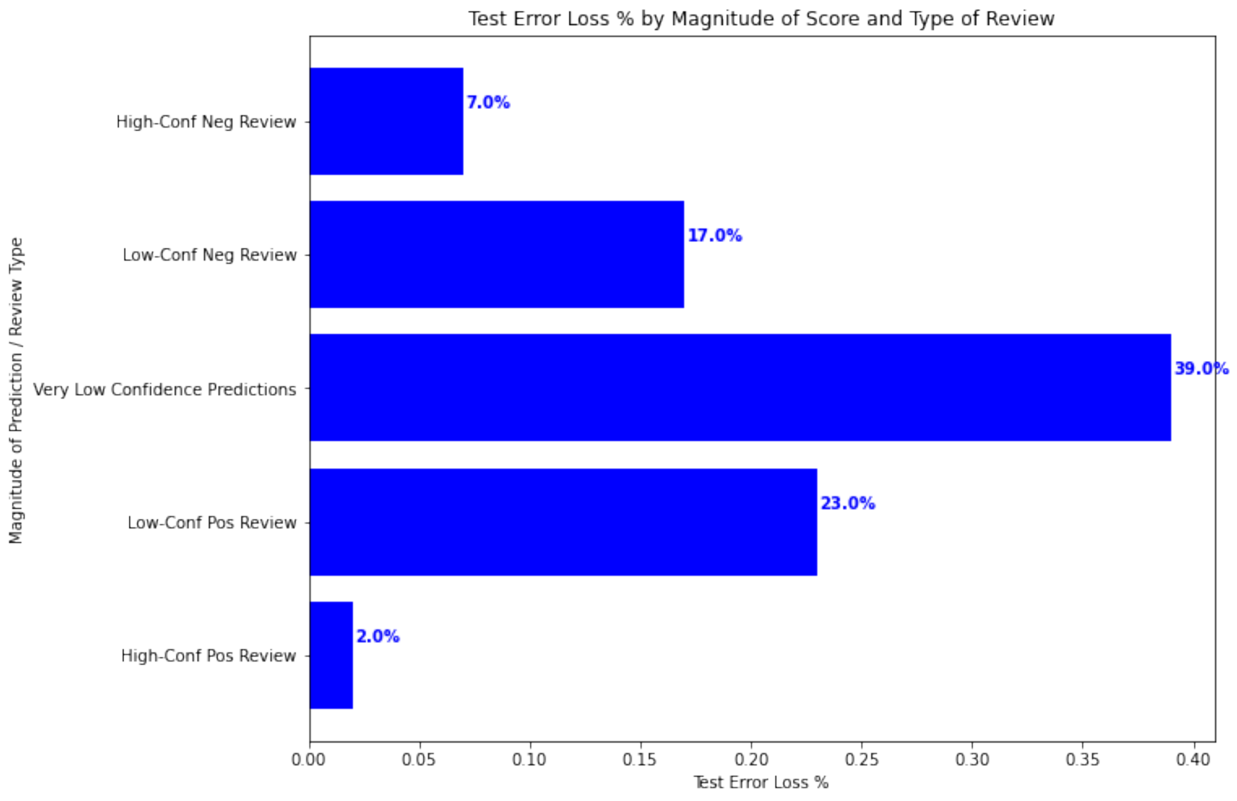
In [12]:
```
#Get w weight dictionary
lambda_reg = .007228703350949573
w_fast = fast_pegasos_algo(X_train,y_train, lambda_reg, 50)
```

In [14]:
```
score_list = []
sentences = []
true_label = y_test
predicted_label = []
prediction_result = []
index_list = []
y_hat = None
#Iterate over data
for row in range(len(X_test)):
    #Get prediction score
    score = dotProduct(X_test[row],w_fast)
    #Define Prediction
    if score <0:
        y_hat = -1
    else:
        y_hat = 1
    #Check Prediction
    if y_hat != y_test[row]:
        result = 'Wrong'
    else:
        result = 'Correct'
    #Append Score, Y_hat prediction, prediction result, and sentence
    predicted_label.append(y_hat)
    score_list.append(score)
    prediction_result.append(result)
    sentences.append(' '.join(X_test[row].keys()))
    index_list.append(row)
```

In [15]:
```
data = [sentences,score_list,predicted_label,y_test,prediction_result,
columns = dict(enumerate(['Sentence','Score','Predicted Label','True L
df = pd.DataFrame(data=data)
df = df.T
df.rename(columns=columns, inplace=True)
df['Bin'] = pd.qcut(df['Score'],5, labels=False)
```

In [16]:
```
class_list = ['High-Conf Pos Review','Low-Conf Pos Review',
              'Very Low Confidence Predictions','Low-Conf Neg Review',

high_conf_pos = df[(df['Bin']==4) & (df['Prediction Result']=='Wrong')
low_conf_pos = df[(df['Bin']==3) & (df['Prediction Result']=='Wrong')]
middle_ground = df[(df['Bin']==2) & (df['Prediction Result']=='Wrong')
low_conf_neg = df[(df['Bin']==1) & (df['Prediction Result']=='Wrong')]
high_conf_neg = df[(df['Bin']==0) & (df['Prediction Result']=='Wrong')

high_conf_pos_acc = len(high_conf_pos) / len(df[df['Bin']==4])
low_conf_pos_acc =  len(low_conf_pos) / len(df[df['Bin']==3])
middle_ground_acc =  len(middle_ground) / len(df[df['Bin']==2])
low_conf_neg_acc =  len(low_conf_neg) / len(df[df['Bin']==1])
high_conf_neg_acc =  len(high_conf_neg) / len(df[df['Bin']==0])

percentage_error_arr = [high_conf_pos_acc,low_conf_pos_acc,
                        middle_ground_acc,low_conf_neg_acc,high_conf_r
```

In [17]:
```
x = class_list
y = percentage_error_arr

fig, ax = plt.subplots(figsize=(10,8))
width = 0.75 # the width of the bars
ind = np.arange(len(y))  # the x locations for the groups
ax.barh(ind, y,color="blue")
ax.set_yticks(ind)
ax.set_yticklabels(x, minor=False)
plt.title('Test Error Loss % by Magnitude of Score and Type of Review'
plt.xlabel('Test Error Loss %')
plt.ylabel('Magnitude of Prediction / Review Type')
#plt.show()
for i, v in enumerate(y):
    ax.text(v+.001, i + .1, str(np.round(v*100,2))+'%', color='blue',
```



Commentary:

There appears to be a strong correlation between higher magnitude scores and accuracy. Both Positive and Negative predictions that were made with strong confidence had a very low test error loss rate, at 2% and 7% respectively. Interestingly, low confidence positive predictions had less error (15%) than their Low Confidence Negative predictions (17%) error. Very low confidence predictions performed the worst, with an error of 37%. The "Very Low confidence Predictions" are defined as the middle quintile, with score values centered around 0. We can infer that the model did not know how to properly classify these examples, but did have some idea, as if it guessed randomly we would expect an error of 50%, however the actual error was 37%.

Problem 14

In [18]:
```
high_conf_wrong_prediction = df[(df['Bin']==4) & (df['Prediction Resul
high_conf_wrong_prediction = high_conf_wrong_prediction.iloc[0,:]
```

In [19]:
```
word_list = []
count_list = []
weight_list = []
score_list = []
product_list = []
abs_value_list = []
for k, v in X_train[high_conf_wrong_prediction['Index']].items():
    word_list.append(k)
    if k in w_fast.keys():
        weight_list.append(w_fast[k])
    count_list.append(v)
```

In [20]:
```
#Look at greatest impact on score by Abs Value of x_i * w_i
data = [word_list,count_list,weight_list]
columns = dict(enumerate(['Word','Count','Weight']))
df1 = pd.DataFrame(data=data)
df1 = df1.T
df1.rename(columns=columns, inplace=True)
df1['Product (x_i * w_i)'] = df1['Weight']*df1['Count']
df1['Abs Val of Product'] = df1['Product (x_i * w_i)'].apply(abs)
df1.sort_values(by='Abs Val of Product',ascending=False,inplace=True)
df1.head(20)
```

Out[20]:

	Word	Count	Weight	Product (x_i * w_i)	Abs Val of Product
18	and	11	0.169692	1.866608	1.866608
0	american	5	0.265604	1.328021	1.328021
2	2	5	-0.236093	-1.180463	1.180463
123	nothing	2	-0.468496	-0.936993	0.936993
92	have	3	-0.29696	-0.890881	0.890881
53	any	2	-0.407629	-0.815258	0.815258
67	it	7	0.105135	0.735945	0.735945
192	also	2	0.354139	0.708278	0.708278
144	most	3	0.230559	0.691678	0.691678
76	well	2	0.339383	0.678766	0.678766
49	on	3	-0.21027	-0.63081	0.630810
55	from	4	0.147558	0.590232	0.590232
35	to	13	-0.044267	-0.575476	0.575476
209	some	3	-0.173381	-0.520142	0.520142
64	of	11	-0.046112	-0.50723	0.507230
121	as	7	0.068246	0.477719	0.477719
118	an	4	-0.108824	-0.435296	0.435296
244	only	1	-0.426073	-0.426073	0.426073
141	women	3	-0.125424	-0.376273	0.376273
65	original	3	-0.125424	-0.376273	0.376273

In [21]:
```
#Look at least impactful words most frequent words
df1.tail(10)
```

Out[21]:

	Word	Count	Weight	Product (x_i * w_i)	Abs Val of Product
167	overseas	1	-0.003689	-0.003689	3.688948e-03
50	discomfort	1	0.003689	0.003689	3.688948e-03
169	student	1	-0.001844	-0.001844	1.844474e-03
87	ian	1	-0.001844	-0.001844	1.844474e-03
10	mostly	1	-0.001844	-0.001844	1.844474e-03
151	raging	1	0.001844	0.001844	1.844474e-03
3	is	14	0.0	0.0	4.817226e-15
98	seem	1	-0.0	-0.0	2.573078e-16
240	jim's	1	-0.0	-0.0	5.835847e-17
241	wellmeaning	1	-0.0	-0.0	2.804238e-17

In [22]:
```
middle_ground_prediction = df[(df['Bin']==3) & (df['Prediction Result'
middle_ground_prediction = middle_ground_prediction.iloc[0,:]
```

In [23]:
```
word_list = []
count_list = []
weight_list = []
score_list = []
product_list = []
abs_value_list = []
for k, v in X_train[middle_ground_prediction['Index']].items():
    word_list.append(k)
    if k in w_fast.keys():
        weight_list.append(w_fast[k])
    count_list.append(v)
```

In [24]:
```
#Look at greatest impact on score by Abs Value of x_i * w_i
data = [word_list,count_list,weight_list]
columns = dict(enumerate(['Word','Count','Weight']))
df1 = pd.DataFrame(data=data)
df1 = df1.T
df1.rename(columns=columns, inplace=True)
df1['Product (x_i * w_i)'] = df1['Weight']*df1['Count']
df1['Abs Val of Product'] = df1['Product (x_i * w_i)'].apply(abs)
df1.sort_values(by='Abs Val of Product',ascending=False,inplace=True)
df1.head(20)
```

Out[24]:

	Word	Count	Weight	Product (x_i * w_i)	Abs Val of Product
30	and	15	0.169692	2.545374	2.545374
57	well	5	0.339383	1.696916	1.696916
77	have	3	-0.29696	-0.890881	0.890881
33	?	8	-0.095913	-0.767301	0.767301
21	he	3	0.237937	0.713811	0.713811
14	to	16	-0.044267	-0.708278	0.708278
129	job	2	0.330161	0.660322	0.660322
262	if	2	-0.328316	-0.656633	0.656633
222	great	2	0.315405	0.63081	0.630810
248	you	2	0.300649	0.601299	0.601299
41	of	13	-0.046112	-0.599454	0.599454
24	from	4	0.147558	0.590232	0.590232
91	sweet	2	0.284049	0.568098	0.568098
141	own	2	0.261915	0.523831	0.523831
114	some	3	-0.173381	-0.520142	0.520142
102	it's	2	0.260071	0.520142	0.520142
11	singer	7	-0.066401	-0.464807	0.464807
163	isn't	2	-0.219492	-0.438985	0.438985
285	only	1	-0.426073	-0.426073	0.426073
45	it	4	0.105135	0.42054	0.420540

In [25]:
```
df1.tail(10)
```

Out[25]:

	Word	Count	Weight	Product (x_i * w_i)	Abs Val of Product
274	nostalgic	1	-0.001844	-0.001844	1.844474e-03
37	giggle	1	0.001844	0.001844	1.844474e-03
137	fianc	1	-0.001844	-0.001844	1.844474e-03
215	smiles	1	0.001844	0.001844	1.844474e-03
68	not	1	-0.001844	-0.001844	1.844474e-03
44	is	5	0.0	0.0	1.720438e-15
73	mr	1	-0.0	-0.0	2.936871e-16
116	unnecessary	1	0.0	0.0	4.585308e-17
128	eponymous	1	0.0	0.0	2.614762e-17
127	hart	1	0.0	0.0	7.579021e-19

For both the incorrect predictions I looked at, each of them has the majority of their score contributed to words like "the","and","any","this",etc. which don't necessarily help understand the sentiment of any given text, as they are included in both positive and negative reviews. Conversely, rare words that appeared in these reviews had no contribution to the models output score, as the model had not seen these words in training and therefore did not have any weights associated with them. I wonder if we could do some feature engineering to account for these transitory words that help make the sentence gramatically correct while not contributing to the sentiment. We could potentially add a feature that encapsulates all of these words, or change the way we represent a text. Say, rather than store the count of each word, we use a boolean value. Or additionally, we add some sort of positional embedding making the words relate to one another in some specific sense.

Homework 3: SVMs & Kernel Methods

Due: Wednesday, March 2, 2022 at 11:59PM EST

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. If you are using LyX, then the listings package tends to work better.

In this problem set we will get up to speed with SVMs and Kernels. Long at first glance, the problem set includes a lot of helpers. You will find a review of kernalization. One section will include a revision of ridge regression which you should start to be familiar with. For the second and third problem some codes are provided to save you some time. Finally, some reminders on positive (semi)definite matrices are included in the Appendix.

1 Support Vector Machines: SVMs with Pegasos

In this first problem we will use Support Vector Machines to predict whether the sentiment of a movie review was *positive* or *negative*. We will represent each review by a vector $\boldsymbol{x} \in \mathbb{R}^d$ where d is the size of the word dictionary and x_i is equal to the number of occurrence of the i-th word in the review $\boldsymbol{x}$. The corresponding label is either $y = 1$ for a positive review or $y = -1$ for a negative review. In class we have seen how to transform the SVM training objective into a quadratic program using the dual formulation. Here we will use a gradient descent algorithm instead.

Subgradients

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

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

There may be 0, 1, or infinitely many subgradients at any point. The *subdifferential* of f at a point $\boldsymbol{x}$, denoted $\partial f(\boldsymbol{x})$, is the set of all subgradients of f at $\boldsymbol{x}$. A good reference for subgradients are the course notes on Subgradients by Boyd et al. Below we derive a property that will make our life easier for finding a subgradient of the hinge loss.

1. Suppose $f_1, \ldots, f_m : \mathbb{R}^d \to \mathbb{R}$ are convex functions, and $f(\boldsymbol{x}) = \max_{i=1,\ldots,,m} f_i(\boldsymbol{x})$. Let k be any index for which $f_k(\boldsymbol{x}) = f(\boldsymbol{x})$, and choose $g \in \partial f_k((\boldsymbol{x})$ (a convex function on $\mathbb{R}^d$ has a non-empty subdifferential at all points). Show that $g \in \partial f(\boldsymbol{x})$.

$$\begin{aligned} f(z) &\geq f(x) + g^T(z-x) \\ f_k(z) &\geq f_k(x) + g_k^T(z-x) \\ f(z) \geq f_k(z) &\geq f_k(x) + g_k^T(z-x) \\ f(z) &\geq f_k(x) + g_k^T(z-x) \end{aligned} \tag{1}$$

2. Give a subgradient of the hinge loss objective The subgradient is defined when $yw^Tx > 1$ or $yw^tx < 1$. We can summarize the subgradients by taking the derivative in each of those cases. $J(\boldsymbol{w}) = \max\left\{0, 1 - y\boldsymbol{w}^T\boldsymbol{x}\right\}$.

$$Subgradient\ of\ J(w) = \begin{cases} 0 & if y w^T x >= 1 \\ -yx & if y w^T x < 1 \end{cases}$$

3. Suppose we have function $f : \mathbb{R}^n \to \mathbb{R}$ which is sub-differentiable everywhere, i.e. $\partial f \neq \emptyset$ for all $x \in \mathbb{R}^n$. Show that f is convex. Note, in the general case, a function is convex if for all x, y in the domain of f and for all $\theta \in (0, 1)$,

$$\theta f(a) + (1-\theta)f(b) \geq f(\theta a + (1-\theta)(b))$$

Hint: Suppose f is not convex, then by definition, there exists a point in some interval: $x_0 \in (a, b)$, such that $f(x_0)$ lies above the line connection $(a, f(a)), (b, f(b))$. Is this possible if the function s sub-differentiable everywhere?

SVM with the Pegasos algorithm

You will train a Support Vector Machine using the Pegasos algorithm[1]. Recall the SVM objective using a linear predictor $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$ and the hinge loss:

$$\min_{\boldsymbol{w}\in\mathbb{R}^d} \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \frac{1}{n}\sum_{i=1}^{n} \max\left\{0, 1 - y_i \boldsymbol{w}^T \boldsymbol{x}_i\right\},$$

where n is the number of training examples and d the size of the dictionary. Note that, for simplicity, we are leaving off the bias term b. Note also that we are using ℓ_2 regularization with a parameter λ. Pegasos is stochastic subgradient descent using a step size rule $\eta_t = 1/(\lambda t)$ for iteration number t. The pseudocode is given below:

```
Input: $\lambda > 0$. Choose $w_1 = 0, t = 0$
While termination condition not met
  For $j = 1, \ldots, n$ (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
      $w_{t+1} = (1 - \eta_t \lambda) w_t$
```

4. Consider the SVM objective function for a single training point[2]: $J_i(\boldsymbol{w}) = \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \max\left\{0, 1 - y_i \boldsymbol{w}^T \boldsymbol{x}_i\right\}$. The function $J_i(\boldsymbol{w})$ is not differentiable everywhere. Specify where the gradient of $J_i(w)$ is not defined. Give an expression for the gradient where it is defined.

The gradient is defined and not defined by the following piece-wise expression:

$$\begin{cases} Defined & \text{when } y_i \boldsymbol{w}^T x_i \neq 1 \\ Not-Defined & \text{when } y_i \boldsymbol{w}^T x_i = 1 \end{cases}$$

[1] Shalev-Shwartz et al. Pegasos: Primal Estimated sub-GrAdient SOlver for SVM.

[2] Recall that if i is selected uniformly from the set $\{1, \ldots, n\}$, then this objective function has the same expected value as the full SVM objective function.

5. Show that a subgradient of $J_i(w)$ is given by

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

   You may use the following facts without proof: 1) If $f_1, \ldots, f_n : \mathbb{R}^d \to \mathbb{R}$ are convex functions and $f = f_1 + \cdots + f_n$, then $\partial f(\boldsymbol{x}) = \partial f_1(\boldsymbol{x}) + \cdots + \partial f_n(\boldsymbol{x})$. 2) For $\alpha \geq 0$, $\partial(\alpha f)(x) = \alpha \partial f(x)$. (Hint: Use the first part of this problem.)

Convince yourself that if your step size rule is $\eta_t = 1/(\lambda t)$, then doing SGD with the subgradient direction from the previous question is the same as given in the pseudocode.

Let $f = J_i(w) = g_i(w) + k_i(w)$, where $g_i(w) = \frac{1}{2}||w||^2$ and $k_i(w) = \max 0, 1 - y_i$. Since the the derivative of f is defined when $y_i w^T x_i \neq 1$ we can calculate the subgradient for two cases, when $y_i w^T x_i < 1$ and when $y_i w^T x_i \leq 1$.

Using the property that the property of derivatives, that the derivative of a sum is the sum of a derivatives, therefore $\partial f_i(w) = \partial g_i(w) + \partial k_i(w)$.

In all cases, the subgradient for $g_i(w)$ is as follows: $\partial g_i(w) = \lambda w$.

Where as for $k_i(w)$:

$$\begin{cases} \partial k_i(w) = -y_i x_i & \text{when } y_i w^T x_i < 1 \\ \partial k_i(w) = \text{ undefined} & \text{when } y_i w^T x_i = 1 \\ \partial k_i(w) = 0 & \text{otherwise} \end{cases}$$

Therefore, the sub-gradients for $J_i(w)$ are as follows:

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

Dataset and sparse representation

We will be using the Polarity Dataset v2.0, constructed by Pang and Lee, provided in the `data_reviews` folder. It has the full text from 2000 movies reviews: 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 `utils_svm_reviews.py` to assist with reading these files. The code removes some special symbols from the reviews and shuffles the data. Load all the data to have an idea of what it looks like.

A usual method to represent text documents in machine learning is with *bag-of-words*. As hinted above, here every possible word in the dictionnary is a feature, and the value of a word feature for a given text is the number of times that word appears in the text. As most words will not appear in any particular document, many of these counts will be zero. Rather than storing many zeros, we use a *sparse representation*, in which only the nonzero counts are tracked. The counts are stored in a key/value data structure, 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}`.

6. Write a function that converts an example (a list of words) into a sparse bag-of-words representation. You may find Python's Counter[3] class to be useful here. Note that a Counter is itself a dictionary.

7. Load all the data and split it into 1500 training examples and 500 validation examples. Format the training data as a list `X_train` of dictionaries and `y_train` as the list of corresponding 1 or -1 labels. Format the test set similarly.

We will be using linear classifiers of the form $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$, and we can store the $\boldsymbol{w}$ 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 $\boldsymbol{w}$ and $\boldsymbol{x}$ 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, `utils_svm_reviews.py` includes two functions for working with sparse vectors: 1) a dot product between two vectors represented as dictionaries and 2) a function that increments one sparse vector by a scaled multiple of another vector, which is a very common operation. 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.

8. Implement the Pegasos algorithm to run on a sparse data representation. The output should be a sparse weight vector $\boldsymbol{w}$ represented as a dictionary. Note that our Pegasos algorithm starts at $w = 0$, which 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. **Also**: If you normalize your data in some way, be sure not to destroy the sparsity of your data. Anything that starts as 0 should stay at 0.

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 very slow. We can make things significantly faster by representing w as $w = sW$, where $s \in \mathbb{R}$ and $W \in \mathbb{R}^d$. You can start with $s = 1$ and W all zeros (i.e. an empty dictionary). Note that both updates (i.e. whether or not we have a margin error) start with rescaling w_t, which we can do simply by setting $s_{t+1} = (1 - \eta_t\lambda)\, s_t$.

9. 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:

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

Implement the Pegasos algorithm with the (s, W) representation described above.

Using the definitions given, we can substitute in the necessary values and manipulate

[3] https://docs.python.org/2/library/collections.html

the expression to reach our equivalency:

$$
\begin{aligned}
w_{t+1} &= s_{t+1}W_{t+1} \\
w_{t+1} &= ((1-\eta_t\lambda)s_t)(W_t + \frac{1}{((1-\eta_t\lambda)s_t}\eta_t y_j x_j) \\
w_{t+1} &= ((1-\eta_t\lambda)s_t)W_t + \eta_t y_j x_j)
\end{aligned} \tag{2}
$$

[4]

10. Run both implementations of Pegasos on the training data for a couple epochs. 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.

11. Write a function `classification_error` that takes a sparse weight vector `w`, a list of sparse vectors `X` and the corresponding list of labels `y`, and returns the fraction of errors when predicting y_i using $\text{sign}(\boldsymbol{w}^T\boldsymbol{x}_i)$. In other words, the function reports the 0-1 loss of the linear predictor $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$.

12. Search for the regularization parameter that gives the minimal percent error on your test set. You should now use your faster Pegasos implementation, and run it to convergence. 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. Plot the test errors you obtained as a function of the parameters λ you tested. (Hint: the error you get with the best regularization should be closer to 15% than 20%. If not, maybe you did not train to convergence.)

Error Analysis

Recall that the *score* is the value of the prediction $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$. 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.

13. Break the predictions on the test set 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?

In natural language processing one can often interpret why a model has performed well or poorly on a specific example. The first step in this process is to look closely at the errors that the model makes.

[4]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.

14. (Optional) Choose an input example $\boldsymbol{x} = (x_1, \ldots, x_d) \in \mathbb{R}^d$ that the model got wrong. We want to investigate what features contributed to this incorrect prediction. One way to rank the importance of the features to the decision is to sort them by the size of their contributions to the score. That is, for each feature we compute $|w_i x_i|$, where w_i is the weight of the ith feature in the prediction function, and x_i is the value of the ith feature in the input x. Create a table of the most important features, sorted by $|w_i x_i|$, including the feature name, the feature value x_i, the feature weight w_i, and the product $w_i x_i$. Attempt to explain 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. Can you think of new features that might help fix a problem? (Think of making groups of words.)

2 Kernel Methods

2.1 Kernelization review

Consider the following optimization problem on a data set $(\boldsymbol{x}_1, y_1), \ldots (\boldsymbol{x}_n, y_n) \in \mathbb{R}^d \times \mathcal{Y}$:

$$\min_{w \in \mathbb{R}^d} R\left(\sqrt{\langle \boldsymbol{w}, \boldsymbol{w} \rangle}\right) + L\left(\langle \boldsymbol{w}, \boldsymbol{x}_1 \rangle, \ldots, \langle \boldsymbol{w}, \boldsymbol{x}_n \rangle\right),$$

where $\boldsymbol{w}, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathbb{R}^d$, and $\langle \cdot, \cdot \rangle$ is the standard inner product on $\mathbb{R}^d$. The function $R : [0, \infty) \to \mathbb{R}$ is nondecreasing and gives us our regularization term, while $L : \mathbb{R}^n \to \mathbb{R}$ is arbitrary[5] and gives us our loss term. We noted in lecture that this general form includes soft-margin SVM and ridge regression, though not lasso regression. Using the representer theorem, we showed if the optimization problem has a solution, there is always a solution of the form $\boldsymbol{w} = \sum_{i=1}^n \boldsymbol{\alpha}_i \boldsymbol{x}_i$, for some $\alpha \in \mathbb{R}^n$. Plugging this into the our original problem, we get the following "kernelized" optimization problem:

$$\min_{\boldsymbol{\alpha} \in \mathbb{R}^n} R\left(\sqrt{\boldsymbol{\alpha}^T K \boldsymbol{\alpha}}\right) + L\left(K\boldsymbol{\alpha}\right),$$

where $K \in \mathbb{R}^{n \times n}$ is the Gram matrix (or "kernel matrix") defined by $K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle$. Predictions are given by

$$f(x) = \sum_{i=1}^n \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x}),$$

and we can recover the original $\boldsymbol{w} \in \mathbb{R}^d$ by $\boldsymbol{w} = \sum_{i=1}^n \alpha_i \boldsymbol{x}_i$.

The *kernel trick* is to swap out occurrences of the kernel k (and the corresponding Gram matrix K) with another kernel. For example, we could replace $k(x_i, x_j) = \langle x_i, x_j \rangle$ by $k'(x_i, x_j) = \langle \psi(x_i), \psi(x_j) \rangle$ for an arbitrary feature mapping $\boldsymbol{\psi} : \mathbb{R}^d \to \mathbb{R}^d$. In this case, the recovered $\boldsymbol{w} \in \mathbb{R}^d$ would be $\boldsymbol{w} = \sum_{i=1}^n \alpha_i \boldsymbol{\psi}(\boldsymbol{x}_i)$ and predictions would be $\langle \boldsymbol{w}, \boldsymbol{\psi}(\boldsymbol{x}_i) \rangle$.

More interestingly, we can replace k by another kernel $k''(\boldsymbol{x}_i, \boldsymbol{x}_j)$ for which we do not even know or cannot explicitly write down a corresponding feature map $\boldsymbol{\psi}$. Our main example of this is the RBF kernel

$$k(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right),$$

[5] You may be wondering "Where are the y_i's?". They're built into the function L. For example, a square loss on a training set of size 3 could be represented as $L(s_1, s_2, s_3) = \frac{1}{3}\left[(s_1 - y_1)^2 + (s_2 - y_2)^2 + (s_3 - y_3)^3\right]$, where each s_i stands for the ith prediction $\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle$.

for which the corresponding feature map ψ is infinite dimensional. In this case, we cannot recover w since it would be infinite dimensional. Predictions must be done using $\alpha \in \mathbb{R}^n$, with $f(x) = \sum_{i=1}^{n} \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x})$.
Your implementation of kernelized methods below should not make any reference to $\boldsymbol{w}$ or to a feature map $\boldsymbol{\psi}$. Your learning routine should return $\boldsymbol{\alpha}$, rather than $\boldsymbol{w}$, and your prediction function should also use $\boldsymbol{\alpha}$ rather than $\boldsymbol{w}$. This will allow us to work with kernels that correspond to infinite-dimensional feature vectors.

2.2 Kernel problems

Ridge Regression: Theory

Suppose our input space is $\mathcal{X}=\mathbb{R}^d$ and our output space is $\mathcal{Y} = \mathbb{R}$. Let $\mathcal{D} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)\}$ be a training set from $\mathcal{X} \times \mathcal{Y}$. We'll use the "design matrix" $X \in \mathbb{R}^{n \times d}$, which has the input vectors as rows:

$$X = \begin{pmatrix} -\boldsymbol{x}_1- \\ \vdots \\ -\boldsymbol{x}_n- \end{pmatrix}.$$

Recall the ridge regression objective function:

$$J(\boldsymbol{w}) = ||X\boldsymbol{w} - y||^2 + \lambda ||\boldsymbol{w}||^2,$$

for $\lambda > 0$.

15. Show that for $\boldsymbol{w}$ to be a minimizer of $J(\boldsymbol{w})$, we must have $X^T X\boldsymbol{w} + \lambda I\boldsymbol{w} = X^T y$. Show that the minimizer of $J(\boldsymbol{w})$ is $\boldsymbol{w} = (X^T X + \lambda I)^{-1} X^T y$. Justify that the matrix $X^T X + \lambda I$ is invertible, for $\lambda > 0$. (You should use properties of positive (semi)definite matrices. If you need a reminder look up the Appendix.)

We can minimize $J(w)$ by taking its gradient and setting it to 0. The gradient is as follows:

$$\nabla J(\boldsymbol{w}) = \nabla(||X\boldsymbol{w} - y||^2 + \lambda ||\boldsymbol{w}||^2) = 2X^T X\boldsymbol{w} + 2\lambda I\boldsymbol{w} - 2X^T y$$

Setting it to 0 and moving over the $-X^T y$ term we have:

$$X^T X\boldsymbol{w} + \lambda I\boldsymbol{w} = X^T y$$

Lastly, we can factor out w and take the inverse of the resulting matrix to arrive at our solution:

$$X^T X\boldsymbol{w} + \lambda I\boldsymbol{w} = X^T y \rightarrow (X^T X + \lambda I)\boldsymbol{w} = X^T y \rightarrow \boldsymbol{w} = (X^T X + \lambda I)^{-1} X^T y$$

By definition, square symmetric matrices are PSD. Therefore, it will have a number positive eigenvalues equal to the matrices rank, and eigenvalues equivalent to 0 equal to the dimension of its kernel. Formalized, we can say it will have d eigenvalues greater than 0, where $d = Rank(X^T X)$ and eigenvalues equal to 0 for $n - d = dim(ker(X^T X))$ where n is the number of columns of the matrix $X^T X$. Also, due to rank nullity, we know that $Rank(X) = Rank(X^T) = Rank(X^T X)$. We also know that when we add any number, α to the diagonal of a square matrix, its eigenvalues are shifted by a degree of α. Therefore, there must exist some $\alpha > 0$ such that the eigenvalues of $X^T X$ are all positive, and therefore, the matrix will be full rank and therefore invertible.

16. Rewrite $X^TXw + \lambda Iw = X^Ty$ as $\boldsymbol{w} = \frac{1}{\lambda}(X^Ty - X^TXw)$. Based on this, show that we can write $w = X^T\boldsymbol{\alpha}$ for some $\boldsymbol{\alpha}$, and give an expression for $\boldsymbol{\alpha}$.

$$\begin{aligned} X^TXw + \lambda Iw &= X^Ty \\ \lambda Iw &= X^Ty - X^TXw \\ w &= \frac{1}{\lambda}(X^Ty - X^TXw) \\ w &= X^T\frac{1}{\lambda}(y - Xw) \\ \text{let } \alpha &= \frac{1}{\lambda}(y - Xw) \\ w &= X^T\alpha \end{aligned} \tag{3}$$

We have shown that $X^TXw + \lambda Iw = X^Ty$ can be re expressed as $w = X^T\boldsymbol{\alpha}$ when we let $\alpha = \frac{1}{\lambda}(y - Xw)$

17. Based on the fact that $\boldsymbol{w} = X^T\boldsymbol{\alpha}$, explain why we say $\boldsymbol{w}$ is "in the span of the data."

Since X is our data, and w is defined as a linear combination of our columns of X, (formalized mathematically by $w = X^T\alpha$)) by definition it w is in the span of our data.

18. Show that $\boldsymbol{\alpha} = (\lambda I + XX^T)^{-1}y$. Note that XX^T is the kernel matrix for the standard vector dot product. (Hint: Replace $\boldsymbol{w}$ by $X^T\boldsymbol{\alpha}$ in the expression for $\boldsymbol{\alpha}$, and then solve for $\boldsymbol{\alpha}$.)

$$\begin{aligned} \alpha &= \frac{1}{\lambda}(y - Xw) \\ \alpha &= \frac{1}{\lambda}(y - XX^T\alpha) \\ \lambda\alpha &= y - XX^T\alpha \\ \lambda\alpha + XX^T\alpha &= y \\ (Id_n\lambda + XX^T)\alpha &= y \\ \alpha &= (Id_n\lambda + XX^T)^{-1}y \end{aligned} \tag{4}$$

19. Give a kernelized expression for the $X\boldsymbol{w}$, the predicted values on the training points. (Hint: Replace $\boldsymbol{w}$ by $X^T\boldsymbol{\alpha}$ and $\boldsymbol{\alpha}$ by its expression in terms of the kernel matrix XX^T.)

$$\begin{aligned} Xw &= X(X^T\alpha) \\ &= X(X^T((Id_n\lambda + XX^T)^{-1}y) \\ &= XX^T(Id_n\lambda + XX^T)^{-1}y) \end{aligned} \tag{5}$$

20. Give an expression for the prediction $f(x) = x^T \boldsymbol{w}^*$ for a new point x, not in the training set. The expression should only involve x via inner products with other $\boldsymbol{x}$'s. (Hint: It is often convenient to define the column vector

$$k_{\boldsymbol{x}} = \begin{pmatrix} \boldsymbol{x}^T \boldsymbol{x}_1 \\ \vdots \\ \boldsymbol{x}^T \boldsymbol{x}_n \end{pmatrix}$$

to simplify the expression.)

$$\begin{aligned} f(x) = x^T w \\ &= x^T(X^T\alpha) \\ &= \sum_{i=1}^{n} \alpha_i \langle x^T, x_i \rangle \\ &= \sum_{i=1}^{n} \alpha_i k_{xi} \end{aligned} \tag{6}$$

Kernels and Kernel Machines

There are many different families of kernels. So far we spoken about linear kernels, RBF/Gaussian kernels, and polynomial kernels. The last two kernel types have parameters. In this section, we'll implement these kernels in a way that will be convenient for implementing our kernelized ridge regression later on. For simplicity, we will assume that our input space is $\mathcal{X} = \mathbb{R}$. This allows us to represent a collection of n inputs in a matrix $X \in \mathbb{R}^{n\times 1}$. You should now refer to the jupyter notebook `skeleton_code_kernels.ipynb`.

21. Write functions that compute the RBF kernel $k_{\mathrm{RBF}(\sigma)}(x, x') = \exp\left(-\|x - x'\|^2 / \left(2\sigma^2\right)\right)$ and the polynomial kernel $k_{\mathrm{poly}(a,d)}(x, x') = (a + \langle x, x' \rangle)^d$. The linear kernel $k_{\mathrm{linear}}(x, x') = \langle x, x' \rangle$, has been done for you in the support code. Your functions should take as input two matrices $W \in \mathbb{R}^{n_1 \times d}$ and $X \in \mathbb{R}^{n_2 \times d}$ and should return a matrix $M \in \mathbb{R}^{n_1 \times n_2}$ where $M_{ij} = k(W_{i\cdot}, X_{j\cdot})$. In words, the (i, j)'th entry of M should be kernel evaluation between w_i (the ith row of W) and x_j (the jth row of X). For the RBF kernel, you may use the scipy function `cdist(X1,X2,'sqeuclidean')` in the package `scipy.spatial.distance`.

22. Use the linear kernel function defined in the code to compute the kernel matrix on the set of points $x_0 \in \mathcal{D}_X = \{-4, -1, 0, 2\}$. Include both the code and the output.

23. Suppose we have the data set $\mathcal{D}_{X,y} = \{(-4, 2), (-1, 0), (0, 3), (2, 5)\}$ (in each set of parentheses, the first number is the value of x_i and the second number the corresponding value of the target y_i). Then by the representer theorem, the final prediction function will be in the span of the functions $x \mapsto k(x_0, x)$ for $x_0 \in \mathcal{D}_X = \{-4, -1, 0, 2\}$. This set of functions will look quite different depending on the kernel function we use. The set of functions $x \mapsto k_{\mathrm{linear}}(x_0, x)$ for $x_0 \in \mathcal{X}$ and for $x \in [-6, 6]$ has been provided for the linear kernel.

    (a) Plot the set of functions $x \mapsto k_{\mathrm{poly}(1,3)}(x_0, x)$ for $x_0 \in \mathcal{D}_X$ and for $x \in [-6, 6]$.

    (b) Plot the set of functions $x \mapsto k_{\mathrm{RBF}(1)}(x_0, x)$ for $x_0 \in \mathcal{X}$ and for $x \in [-6, 6]$.

Note that the values of the parameters of the kernels you should use are given in their definitions in (a) and (b).

24. By the representer theorem, the final prediction function will be of the form $f(x) = \sum_{i=1}^{n} \alpha_i k(x_i, x)$, where $x_1, \ldots, x_n \in \mathcal{X}$ are the inputs in the training set. We will use the class `Kernel_Machine` in the skeleton code to make prediction with different kernels. Complete the `predict` function of the class `Kernel_Machine`. Construct a `Kernel_Machine` object with the RBF kernel (sigma=1), with prototype points at $-1, 0, 1$ and corresponding weights α_i $1, -1, 1$. Plot the resulting function.

Note: For this last problem, and for other problems below, it may be helpful to use partial application on your kernel functions. For example, if your polynomial kernel function has signature `polynomial_kernel(W, X, offset, degree)`, you can write `k = functools.partial(polynomial_kernel, offset=2, degree=2)`, and then a call to `k(W,X)` is equivalent to `polynomial_kernel(W, X, offset=2, degree=2)`, the advantage being that the extra parameter settings are built into `k(W,X)`. This can be convenient so that you can have a function that just takes a kernel function `k(W,X)` and doesn't have to worry about the parameter settings for the kernel.

Kernel Ridge Regression: Practice

In the zip file for this assignment, we provide a training `krr-train.txt` and test set `krr-test.txt` for a one-dimensional regression problem, in which $\mathcal{X} = \mathcal{Y} = \mathcal{A} = \mathbb{R}$. Fitting this data using kernelized ridge regression, we will compare the results using several different kernel functions. Because the input space is one-dimensional, we can easily visualize the results.

25. Plot the training data. You should note that while there is a clear relationship between x and y, the relationship is not linear.

26. In a previous problem, we showed that in kernelized ridge regression, the final prediction function is $f(\boldsymbol{x}) = \sum_{i=1}^{n} \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x})$, where $\alpha = (\lambda I + K)^{-1} y$ and $K \in \mathbb{R}^{n \times n}$ is the kernel matrix of the training data: $K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$, for $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$. In terms of kernel machines, α_i is the weight on the kernel function evaluated at the training point $\boldsymbol{x}_i$. Complete the function `train_kernel_ridge_regression` so that it performs kernel ridge regression and returns a `Kernel_Machine` object that can be used for predicting on new points.

27. Use the code provided to plot your fits to the training data for the RBF kernel with a fixed regularization parameter of 0.0001 for 3 different values of sigma: 0.01, 0.1, and 1.0. What values of sigma do you think would be more likely to over fit, and which less?

28. Use the code provided to plot your fits to the training data for the RBF kernel with a fixed sigma of 0.02 and 4 different values of the regularization parameter λ: 0.0001, 0.01, 0.1, and 2.0. What happens to the prediction function as $\lambda \to \infty$?

29. Find the best hyperparameter settings (including kernel parameters and the regularization parameter) for each of the kernel types. Summarize your results in a table, which gives training error and test error for each setting. Include in your table the best settings for each kernel type, as well as nearby settings that show that making small change in any one of the hyperparameters in either direction will cause the performance to get worse. You should use average square loss on the test set to rank the parameter settings. To make

things easier for you, we have provided an sklearn wrapper for the kernel ridge regression function we have created so that you can use sklearn's GridSearchCV. Note: Because of the small dataset size, these models can be fit extremely fast, so there is no excuse for not doing extensive hyperparameter tuning.

30. Plot your best fitting prediction functions using the polynomial kernel and the RBF kernel. Use the domain $x \in (-0.5, 1.5)$. Comment on the results.

31. The data for this problem was generated as follows: A function $f : \mathbb{R} \to \mathbb{R}$ was chosen. Then to generate a point (x, y), we sampled x uniformly from $(0, 1)$ and we sampled $\epsilon \sim \mathcal{N}\left(0, 0.1^2\right)$ (so $var(\epsilon) = 0.1^2$). The final point is $(x, f(x) + \epsilon)$. What is the Bayes decision function and the Bayes risk for the loss function $\ell\left(\hat{y}, y\right) = \left(\hat{y} - y\right)^2$.

3 Kernel SVMs with Kernelized Pegasos (Optional)

32. Load the SVM training `svm-train.txt` and `svm-test.txt` test data from the zip file. Plot the training data using the code supplied. Are the data linearly separable? Quadratically separable? What if we used some RBF kernel?

33. Unlike for kernel ridge regression, there is no closed-form solution for SVM classification (kernelized or not). Implement kernelized Pegasos. Because we are not using a sparse representation for this data, you will probably not see much gain by implementing the "optimized" versions described in the problems above.

34. Find the best hyperparameter settings (including kernel parameters and the regularization parameter) for each of the kernel types. Summarize your results in a table, which gives training error and test error (i.e. average 0/1 loss) for each setting. Include in your table the best settings for each kernel type, as well as nearby settings that show that making small change in any one of the hyperparameters in either direction will cause the performance to get worse. You should use the 0/1 loss on the test set to rank the parameter settings.

35. Plot your best fitting prediction functions using the linear, polynomial, and the RBF kernel. The code provided may help.

Appendix

Here we are recalling important properties of positive (semi)definite matrices. The exercises below are for revisions for student who may not feel comfortable with these notions. None of the appendix is for credit.

A Positive Semidefinite Matrices (not for credit)

In statistics and machine learning, we use positive semidefinite matrices a lot. Let's recall some definitions from linear algebra that will be useful here:

Definition. A set of vectors $\{x_1, \ldots, x_n\}$ is **orthonormal** if $\langle x_i, x_i \rangle = 1$ for any $i \in \{1, \ldots, n\}$ (i.e. x_i has unit norm), and for any $i, j \in \{1, \ldots, n\}$ with $i \neq j$ we have $\langle x_i, x_j \rangle = 0$ (i.e. x_i and x_j are orthogonal).

Note that if the vectors are column vectors in a Euclidean space, we can write this as $x_i^T x_j = \mathbb{1} i \neq j$ for all $i, j \in \{1, \ldots, n\}$.

Definition. A matrix is **orthogonal** if it is a square matrix with orthonormal columns.

It follows from the definition that if a matrix $M \in \mathbb{R}^{n \times n}$ is orthogonal, then $M^T M = I$, where I is the $n \times n$ identity matrix. Thus $M^T = M^{-1}$, and so $MM^T = I$ as well.

Definition. A matrix M is **symmetric** if $M = M^T$.

Definition. For a square matrix M, if $Mv = \lambda v$ for some column vector v and scalar λ, then v is called an **eigenvector** of M and λ is the corresponding **eigenvalue**.

Theorem. [Spectral Theorem]A real, symmetric matrix $M \in \mathbb{R}^{n \times n}$ can be diagonalized as $M = Q \Sigma Q^T$, where $Q \in \mathbb{R}^{n \times n}$ is an orthogonal matrix whose columns are a set of orthonormal eigenvectors of M, and Σ is a diagonal matrix of the corresponding eigenvalues.

Definition. A real, symmetric matrix $M \in \mathbb{R}^{n \times n}$ is **positive semidefinite (psd)** if for any $x \in \mathbb{R}^n$,

$$x^T M x \geq 0.$$

Note that unless otherwise specified, when a matrix is described as positive semidefinite, we are implicitly assuming it is real and symmetric (or complex and Hermitian in certain contexts, though not here).

As an exercise in matrix multiplication, note that for any matrix A with columns $a_1, \ldots, a_d$, that is

$$A = \begin{pmatrix} | & & | \\ a_1 & \cdots & a_d \\ | & & | \end{pmatrix} \in \mathbb{R}^{n \times d},$$

we have

$$A^T M A = \begin{pmatrix} a_1^T M a_1 & a_1^T M a_2 & \cdots & a_1^T M a_d \\ a_2^T M a_1 & a_2^T M a_2 & \cdots & a_2^T M a_d \\ \vdots & \vdots & \cdots & \vdots \\ a_d^T M a_1 & a_d^T M a_2 & \cdots & a_d^T M a_d \end{pmatrix}.$$

So M is psd if and only if for any $A \in \mathbb{R}^{n \times d}$, we have $diag(A^T M A) = \left(a_1^T M a_1, \ldots, a_d^T M a_d\right)^T \succeq 0$, where $\succeq$ is elementwise inequality, and 0 is a $d \times 1$ column vector of 0's .

1. Use the definition of a psd matrix and the spectral theorem to show that all eigenvalues of a positive semidefinite matrix M are non-negative. [Hint: By Spectral theorem, $\Sigma = Q^T M Q$ for some Q. What if you take $A = Q$ in the "exercise in matrix multiplication" described above?]

2. In this problem, we show that a psd matrix is a matrix version of a non-negative scalar, in that they both have a "square root". Show that a symmetric matrix M can be expressed as $M = BB^T$ for some matrix B, if and only if M is psd. [Hint: To show $M = BB^T$ implies M is psd, use the fact that for any vector v, $v^T v \geq 0$. To show that M psd implies $M = BB^T$ for some B, use the Spectral Theorem.]

B Positive Definite Matrices (not for credit)

Definition. A real, symmetric matrix $M \in \mathbb{R}^{n \times n}$ is **positive definite** (spd) if for any $x \in \mathbb{R}^n$

with $x \neq 0$,

$$x^T M x > 0.$$

1. Show that all eigenvalues of a symmetric positive definite matrix are positive. [Hint: You can use the same method as you used for psd matrices above.]

2. Let M be a symmetric positive definite matrix. By the spectral theorem, $M = Q\Sigma Q^T$, where Σ is a diagonal matrix of the eigenvalues of M. By the previous problem, all diagonal entries of Σ are positive. If $\Sigma = diag\left(\sigma_1, \ldots, \sigma_n\right)$, then $\Sigma^{-1} = diag\left(\sigma_1^{-1}, \ldots, \sigma_n^{-1}\right)$. Show that the matrix $Q\Sigma^{-1}Q^T$ is the inverse of M.

3. Since positive semidefinite matrices may have eigenvalues that are zero, we see by the previous problem that not all psd matrices are invertible. Show that if M is a psd matrix and I is the identity matrix, then $M + \lambda I$ is symmetric positive definite for any $\lambda > 0$, and give an expression for the inverse of $M + \lambda I$.

4. Let M and N be symmetric matrices, with M positive semidefinite and N positive definite. Use the definitions of psd and spd to show that $M + N$ is symmetric positive definite. Thus $M + N$ is invertible. (Hint: For any $x \neq 0$, show that $x^T(M + N)x > 0$. Also note that $x^T(M + N)x = x^T Mx + x^T Nx$.)