P2

March 27, 2017

1 2 Positive Semidefinite Matrix

1.1 2-1

Example:

$$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

1.2 2-2

Proof:

Let M be a psd matrix and v be one of the eigenvector of M with eigenvalue λ.

By definition of psd we must have $v^T Mv \geq 0$. Let $v^T Mv = c$. By definition of eigenvector, we have:

$$\begin{aligned} v^T(Mv) &= c \\ v^T \lambda v &= c \\ \lambda v^T v &= c \\ \lambda &= \frac{c}{||v||} \end{aligned}$$

Since $c \geq 0$ and $||v|| = 0$ (v is eigenvector), we must have $\lambda \geq 0$.

1.3 2-3

Proof:

If $M = BB^T$ where $B \in \mathbf{R}^{n\times d}$ .Let $x \in \mathbf{R}^n$, then we must have:

$$\begin{aligned} x^T M x &= x^T (BB^T) x \\ &= (x^T B)(B^T x) \\ &= (B^T x)^T (B^T x) \\ &= ||B^T x|| \geq 0. \end{aligned}$$

Then M is psd.

If M is psd, then it is real and symmetric. By Spectual Theorem, $M = Q\Sigma Q^T$, where Σ is a diagonal matrix of the corresponding eigenvalues of M. By part 2-2, we proved that all eigenvalues are positive. Thus we can always write $\Sigma = RR^T$ where R is a diagonal matrix whose values is the square root of eigenvalues. Let $B = QS$, $M = BB^T$

```
In [ ]:
```

P3

March 27, 2017

1 3 Positive Definite Matrix

1.1 3-1

Let M be a positive definite matrix and v be one of the eigenvector of M with eigenvalue λ.

By definition of psd we must have $v^T M v > 0$. Let $v^T M v = c$. By definition of eigenvector, we have:

$$\begin{aligned} v^T(Mv) &= c \\ v^T \lambda v &= c \\ \lambda v^T v &= c \\ \lambda &= \frac{c}{||v||} \end{aligned}$$

Since $c > 0$ and $||v|| = 1$ (v is eigenvector), we must have $\lambda > 0$.

1.2 3-2

Proof:

Since Q is orthogonal matrix, we have $Q^TQ = 1$, therefore:

$$M(Q\Sigma^{-1}Q^T) = (Q\Sigma Q^T)(Q\Sigma^{-1}Q^T) = Q\Sigma\Sigma^{-1}Q^T = QQ^T = I$$

1.3 3-3

Proof: Let $x \in \mathbf{R}^n$ be any vector. Since M is symmetric positive definite matrix we must have $x^T Mx > 0$

$$
\begin{aligned}
x^T(M+\lambda I)x &= x^T(Mx + \lambda Ix) \\
&= x^T Mx + \lambda x^T Ix \\
&= x^T Mx + \lambda ||x|| \geq x^T Mx > 0.
\end{aligned}
$$

Hence $x^T(M+\lambda I)x$ is symmetric positive definite matrix.

Let v be an eigen vector of M and λ_v be its corresponding eigenvalue. Observe that $(M+\lambda I)v = (\lambda_v + \lambda)v$. Hence v is also an eigenvector of $M + \lambda I$ with eigenvalue $\lambda_v + \lambda$.

Since $M + \lambda I$ is a positive definite matrix, by our result of 3-2, its inverse has form $Q(\Sigma + \lambda I)^{-1}Q^T$ where $\Sigma^{-1} = diag((\sigma_1 + \lambda)^{-1}, (\sigma_2 + \lambda)^{-1}, ..., (\sigma_n + \lambda)^{-1})$.

1.4 3-4

Proof: Let $x \in \mathbf{R}$ be any vector. We have:

$$x^T(M+N)x = x^T M x + x^T N x > 0$$

```
In [ ]:
```

P4

March 27, 2017

1 4 Kernel Matrix

1.1 4-1

By the definition of K, $K_{ij} = x_i^T x_j = x_j^T x_i$. We also have $K_{ii} = x_i^T x_i = ||x_i||^2$. Using these informations we can calculate $x_1^T x_1 - x_2^T x_1 - x_1^T x_2 + x_2^T x_2 = (x_1 - x_2)^T (x_1 - x_2) = ||x_1 - x_2||^2$. There fore K contains pairwise distances information of X.

```
In [ ]:
```

p5

March 27, 2017

1 5 Kernel Ridge Regression

1.1 5-1

If w is a minimizer of $J(w)$, we must have $\frac{dJ(w)}{dw} = 0$. Therefore:

$$\begin{aligned}
\frac{dJ(w)}{dw} &= 2X^T(Xw - y) + 2\lambda Iw = 0 \\
2X^TXw - 2X^Ty + 2\lambda Iw &= 0 \\
X^TXw + \lambda Iw &= X^Ty \\
(X^TX + \lambda I)w &= X^Ty.
\end{aligned}$$

By our conclusion from 2-3, X^TX is a psd Matrix and from 3-3 $X^TX + \lambda I$ is positive definite for any $\lambda > 0$ and is invertible. Then we have $w = (X^TX + \lambda I)^{-1}X^Ty$.

1.2 5-2

Since $w = \frac{1}{\lambda}(X^T y - X^T X w) = \frac{1}{\lambda} X^T (y - Xw)$. We have $w = X\alpha$ where $\alpha = \frac{1}{\lambda}(y - Xw)$

1.3 5-3

$w = \sum_{i=1}^{n} \alpha_i xi$. Hence w must lie in the span of x_i, this is what we described as "in the span of data".

1.4 5-4

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

1.5 5-5

Let $K = XX^T$

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

1.6 5-6

$$
\begin{aligned}
x^T w &= x^T X^T \alpha \\
&= k_x^T (\lambda I - X X^T)^{-1} y
\end{aligned}
$$

```
In [ ]:
```

P7

March 27, 2017

1 7 Kernelized Pegasos

1.1 7-1

The margin of SVM is $y_i w^T x_i$. By Representer's theorem, $w = \sum_{i=1}^{n}$. Therefore:

$$
\begin{aligned}
y_j w^T x_j &= y_j \sum_{i=1}^{n} \alpha_i^{(t)} x_i x j \\
&= y_j \sum_{i=1}^{n} \alpha_i^{(t)} < x_i, x_j > \\
&= y_j K_j^T \alpha^{(t)}
\end{aligned}
$$

1.2 7-2

By Pegaso Algorithm,If t+1 step does not violate margin:

$$w^{(t+1)} = (1 - \eta^{(t)}\lambda)w^{(t)}$$
$$X^T\alpha^{(t+1)} = (1 - \eta^{(t)}\lambda)X^T\alpha^{(t)}$$
$$\alpha^{(t+1)} = (1 - \eta^{(t)}\lambda)\alpha^{(t)}$$

Algorithm step: If t+1 step does not violate margin, then $\alpha_i^{(t+1)} = (1 - \eta_t\lambda)\alpha_i^{(t)}$

1.3 7-3

Define (0,0,...,1,...,0) as a vector of dimension n where j^{th} element is 1 and other elements are zeros. If t+1 step, the prediction on point x_j does violate margin:

$$
\begin{aligned}
w^{(t+1)} &= (1-\eta^{(t)}\lambda)w^{(t)} + \eta_t y_j x_j \\
X^T\alpha^{(t+1)} &= (1-\eta^{(t)}\lambda)X^T\alpha^{(t)} + \eta_t y_j X^T(0,0,...,1,...,0) \\
\alpha^{(t+1)} &= (1-\eta^{(t)}\lambda)\alpha^{(t)} + \eta_t y_j(0,0,...,1,...,0)
\end{aligned}
$$

Algorithm step: If t+1 step, the prediction on point x_j does violate margin, then first update $\alpha_i^{(t+1)} = (1-\eta_t\lambda)\alpha_i^{(t)}$ Then we add the j^{th} element of α_t with $y_j x_j$

Psuedo Code:

Algorithm 2: Kernelized Pegasos Algorithm

```
input: Kernel Matrix K labels y_1,...,y_n ∈ {-1,1} and λ > 0.
α^(1) = (0,...,0) ∈ R^d
t = 0 # step number
repeat
  t = t + 1
  η^(t) = 1/(tλ) # step multiplier
  randomly choose j in 1,...,n
  if y_j K_j^T α^(t) < 1
    α^(t+1) = (1 - η^(t) λ) α^(t)
    α^(t+1)[j] = α^(t+1)[j] + η^(t) y_j
  else
    α^(t+1) = (1 - η^(t) λ) α^(t)
until bored
return α^(t)
```

7.4 (Optional) Efficient Kernelized Pegasos

Algorithm 2: Kernelized Pegasos Algorithm

```
input: Kernel Matrix K labels y_1,...,y_n ∈ {-1,1} and λ > 0.
α^(1) = (0,...,0) ∈ R^d
s^(0) = 1
t = 0 # step number
repeat
  t = t + 1
  η^(t) = 1/(tλ) # step multiplier
  s^(t+1) = (1 - η^(t) λ) s^(t)
  randomly choose j in 1,...,n
  if y_j K_j^T α^(t) < 1
    α^(t+1) = α^(t)
    α^(t+1)[j] = α^(t+1)[j] + (1/s^(t+1)) η^(t) y_j
until bored
α^(t) = α^(t) s^(t)
return α^(t)
```

P8

March 27, 2017

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

1 8-2 Kernel and Kernel Machine

1.1 8-2-1

In [173]:
```
#Kernel test W,X,sigma
import numpy as np
W_test = np.array([[1,2,3],[4,5,6]])
X_test = np.array([[2,3,4],[5,6,7]])
sigma = 1
```

In [174]:
```
from scipy.spatial.distance import cdist
def linear_kernel(W, X):
    """
    Computes the linear kernel between two sets of vectors.
    Args:
        W, X - two matrices of dimensions n1xd and n2xd
    Returns:
        matrix of size n1xn2, with w_i^T x_j in position i,j
    """
    return np.dot(W,np.transpose(X))

def RBF_kernel(W,X,sigma):
    """
    Computes the RBF kernel between two sets of vectors
    Args:
        W, X - two matrices of dimensions n1xd and n2xd
        sigma - the bandwidth (i.e. standard deviation) for the RBF/Gauss
    Returns:
        matrix of size n1xn2, with exp(-||w_i-x_j||^2/(2 sigma^2)) in pos
    """
    #TODO
    diff_mat = cdist(W,X,'sqeuclidean')
    return np.exp(-(diff_mat)/(2*sigma**2))

def polynomial_kernel(W, X, offset, degree):
    """
    Computes the inhomogeneous polynomial kernel between two sets of vect
    Args:
        W, X - two matrices of dimensions n1xd and n2xd
        offset, degree - two parameters for the kernel
    Returns:
        matrix of size n1xn2, with (offset + <w_i,x_j>)^degree in positio
    """
    #TODO
    return (offset+linear_kernel(W,X))**degree
```

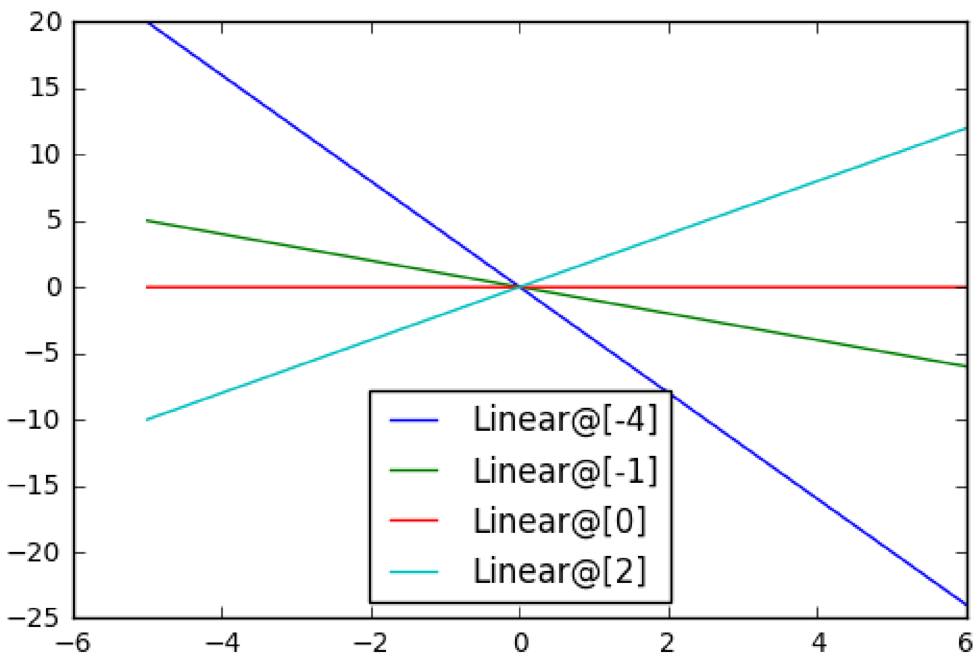
1.2 8-2-2

```
In [175]: def linear_kernel(W, X):
              """
              Computes the linear kernel between two sets of vectors.
              Args:
                  W, X - two matrices of dimensions n1xd and n2xd
              Returns:
                  matrix of size n1xn2, with w_i^T x_j in position i,j
              """
              return np.dot(W,np.transpose(X))
          X0=[[-4],[-1],[0],[2]]
          linear_kernel(X0,X0)
```

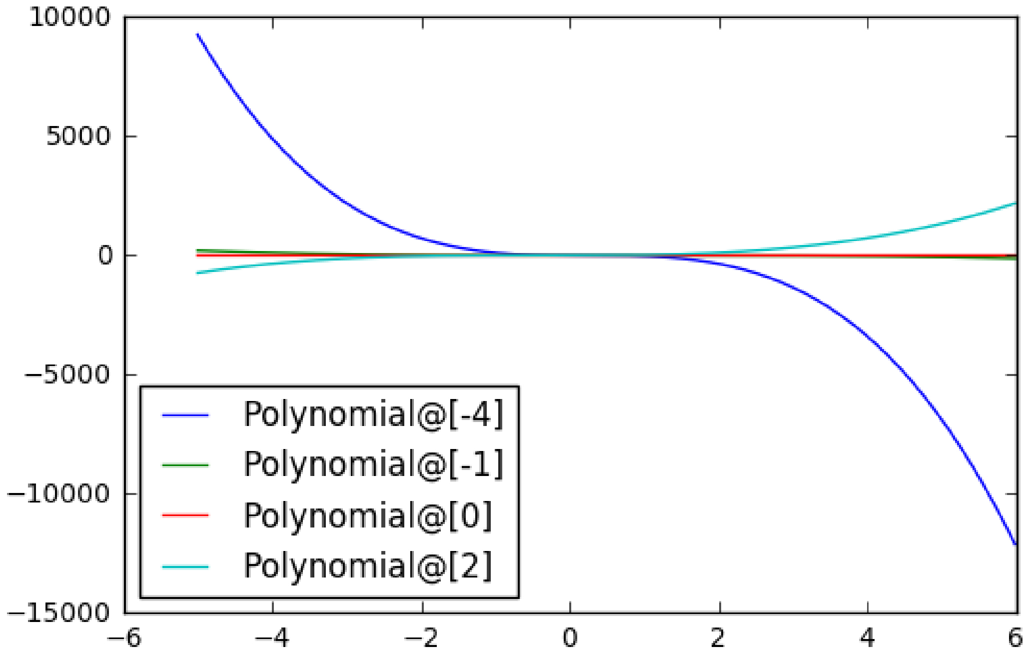
```
Out[175]: array([[16,  4,  0, -8],
                 [ 4,  1,  0, -2],
                 [ 0,  0,  0,  0],
                 [-8, -2,  0,  4]])
```

1.3 8-2-3

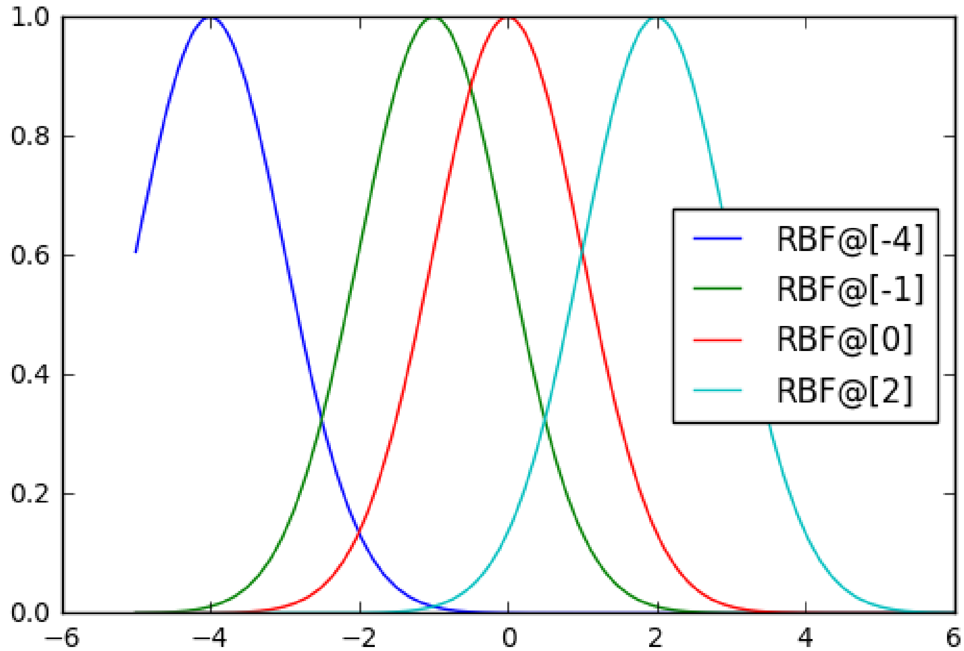
In [176]:
```
import matplotlib.pyplot as plt
plot_step = .01
xpts = np.arange(-5.0, 6, plot_step).reshape(-1,1)
prototypes = np.array([-4,-1,0,2]).reshape(-1,1)

# 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()
```



In [177]:
```
# Poly 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()
```




```
In [178]: # RBF 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()
```



RBF@[-4]
RBF@[-1]
RBF@[0]
RBF@[2]

1.4 8-2-4

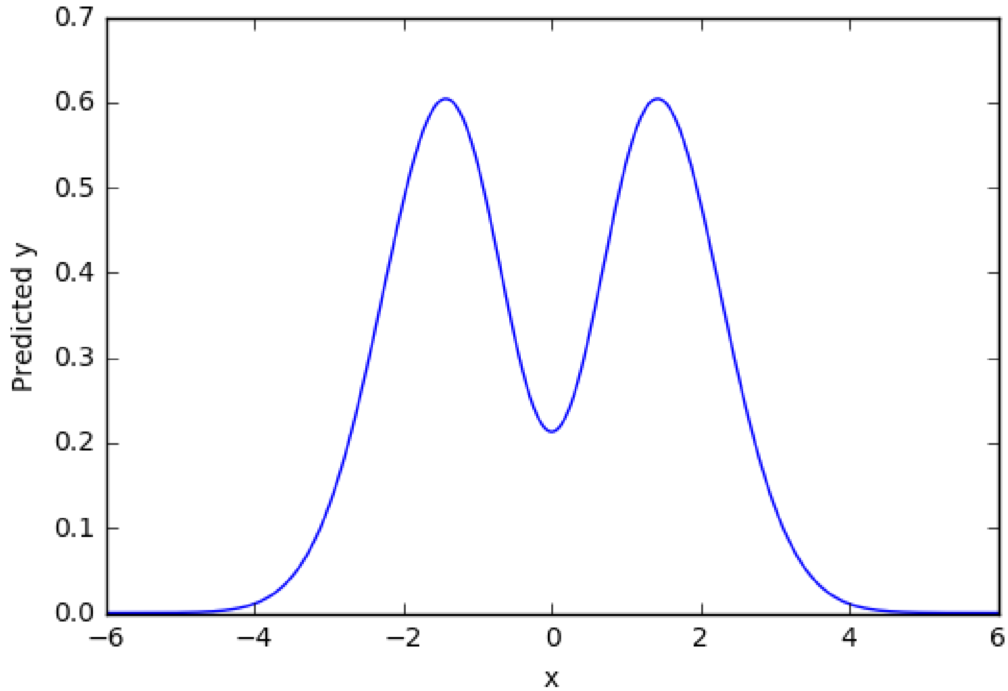
```
In [179]: class Kernel_Machine(object):
              def __init__(self, kernel, prototype_points, weights):
                  """
                  Args:
                      kernel(W,X) - a function return the cross-kernel matrix betwe
                      prototype_points - an Rxd matrix with rows mu_1,...,mu_R
                      weights - a vector of length R
                  """

                  self.kernel = kernel
                  self.prototype_points = prototype_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.  S
                          Sum_{i=1}^R w_i k(x_j, mu_i)
                  """
                  # TODO
                  xX = self.kernel(X,self.prototype_points)
                  return np.dot(xX,self.weights)
```

```
In [180]: from functools import partial
          prototype_points = np.array([-1,0,1]).reshape(-1,1)
          weights = np.array([1,-1,1]).reshape(-1,1)
          x=np.array([-4,-1,0,2]).reshape(-1,1)
          k = partial(RBF_kernel,sigma=1)
          machine = Kernel_Machine(k,prototype_points,weights)
```

```
In [181]: machine.predict(x)
```

```
Out[181]: array([[ 0.01077726],
                 [ 0.52880462],
                 [ 0.21306132],
                 [ 0.48230437]])
```

```
In [182]: #Plot the result function
          plot_step = .001
          xpts = np.arange(-6 , 6, plot_step).reshape(-1,1)
          plt.plot(xpts, machine.predict(xpts), label=label)
          plt.xlabel('x')
          plt.ylabel('Predicted y')
          plt.show()
```

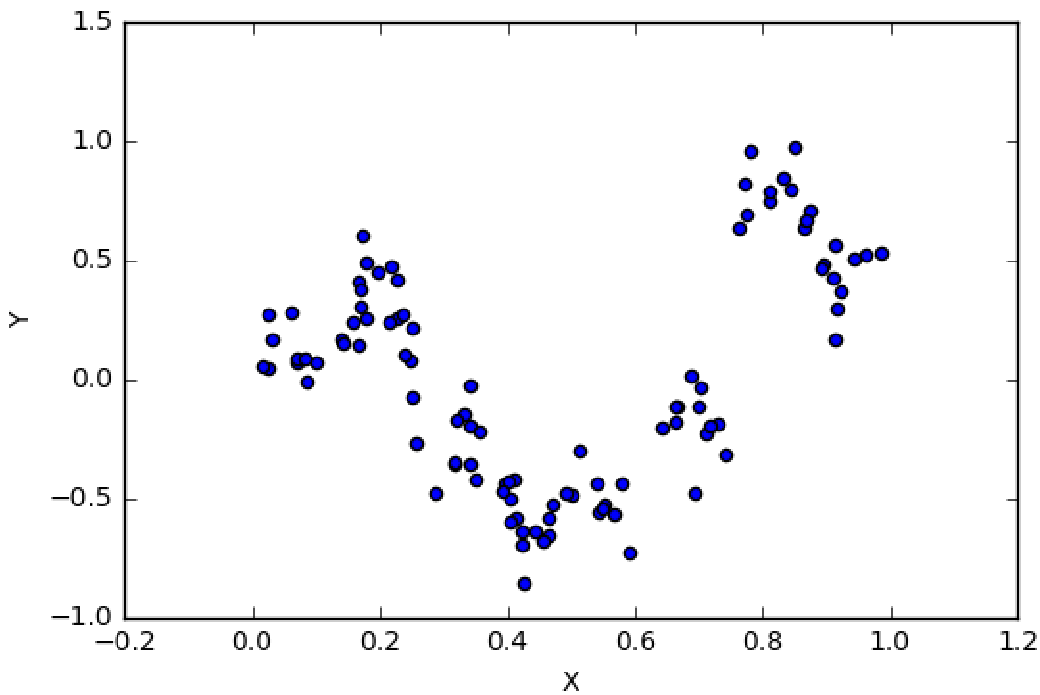

0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
Predicted y
−6
−4
−2
0
2
4
6
x

2 8-3 Kernel Ridge Regression

2.1 8-3-1 Show data

```
In [183]: data_train,data_test = np.loadtxt("../krr-train.txt"),np.loadtxt("../krr-
          x_train, y_train = data_train[:,0].reshape(-1,1),data_train[:,1].reshape(
          x_test, y_test = data_test[:,0].reshape(-1,1),data_test[:,1].reshape(-1,1
```

```
In [184]: plt.scatter(x_train,y_train)
          plt.xlabel('X')
          plt.ylabel('Y')
          plt.show()
```



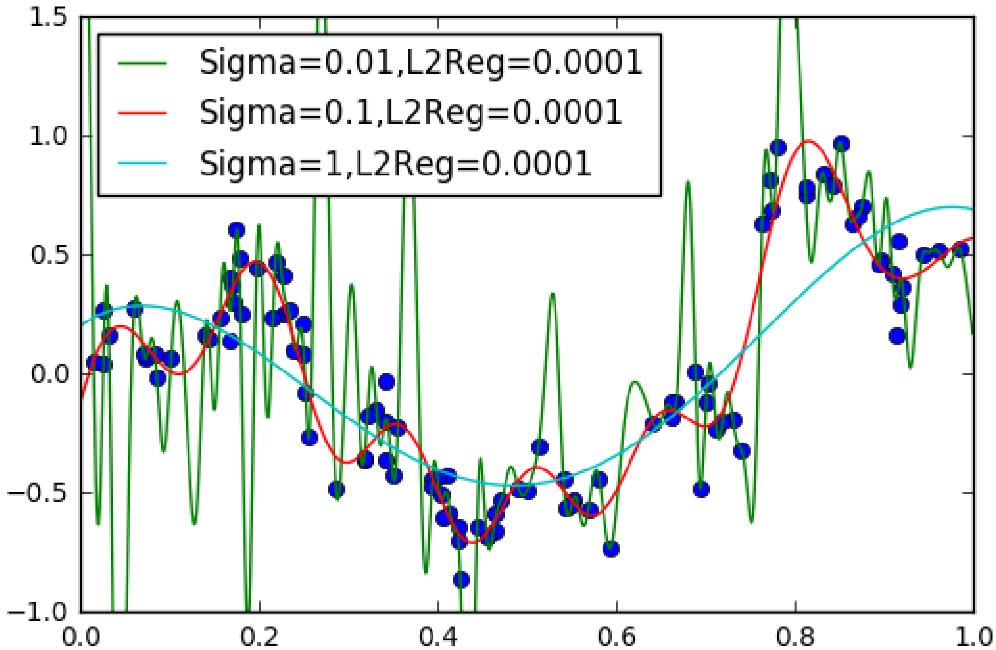
As we seen X and Y are non lienar and there is a clear twist at x=0.5

2.2 8-3-2

```
In [185]: def train_kernel_ridge_regression(X, y, kernel, l2reg):
              # TODO
              K = kernel(X,X)
              dim_K = K.shape[0]
              alpha = np.linalg.inv((np.identity(dim_K)*l2reg+K)).dot(y)
              return Kernel_Machine(kernel, X, alpha)
```

2.3 8-3-3

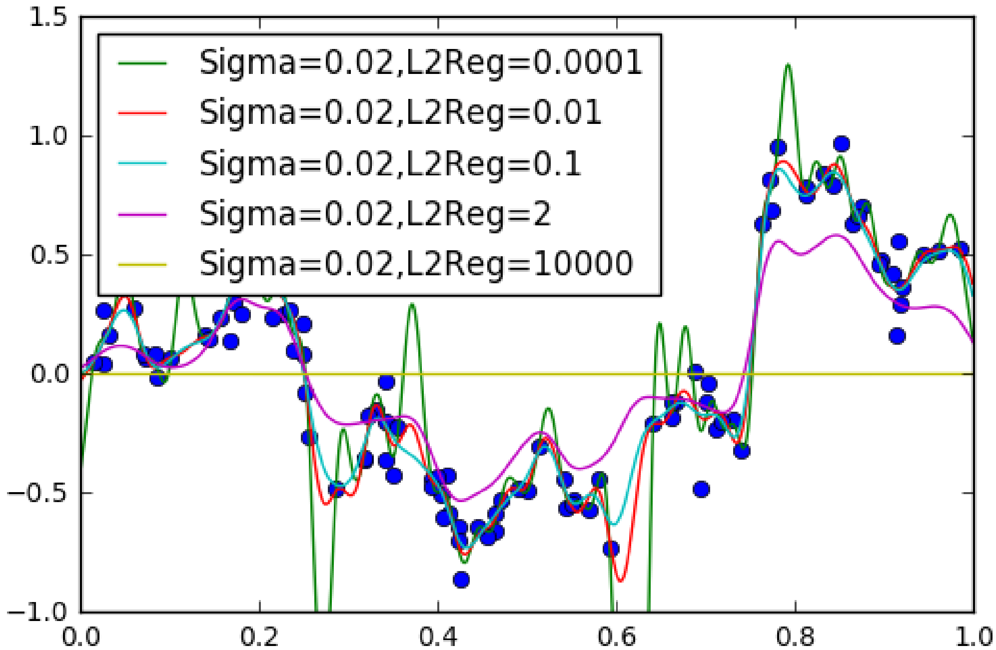
```
In [186]: plot_step = .001
          xpts = np.arange(0 , 1, plot_step).reshape(-1,1)
          plt.plot(x_train,y_train,'o')
          l2reg = 0.0001
          for sigma in [.01,.1,1]:
              k = functools.partial(RBF_kernel, sigma=sigma)
              f = train_kernel_ridge_regression(x_train, 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()
```



When $\sigma = 0.01$ RBF kernel tends to overfit, because it assigns huge inner product to the neighbours of training data. When $\sigma = 1$ RBF kernel does not overfit, but it could be under fit.

2.4 8-3-4

```
In [187]: 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()
```



The prediction function will become a constant prediction of 0 as $\lambda \to \infty$.

2.5 8-3-5

In [17]:
```
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.offs
        else:
            raise ValueError('Unrecognized kernel type requested.')

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

        return self

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

        return(self.kernel_machine_.predict(X))

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

In [18]:
```
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,
predefined_split = PredefinedSplit(test_fold=test_fold)
```

In [135]:
```
param_grid = [{'kernel': ['RBF'],'sigma':np.exp2(-np.arange(-5,5,0.1)), '
              {'kernel':['polynomial'],'offset':np.arange(-10,10,1), 'deg
              {'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_b
                   # n_jobs = -1  #should allow parallelism, but crashes P
                   )
grid.fit(np.vstack((x_train,x_test)),np.vstack((y_train,y_test)))
```

Out[135]:
```
GridSearchCV(cv=PredefinedSplit(test_fold=array([-1, -1, ...,  0,  0])),
       error_score='raise',
       estimator=KernelRidgeRegression(degree=2, kernel='RBF', l2reg=1, o
       fit_params={}, iid=True, n_jobs=1,
       param_grid=[{'kernel': ['RBF'], 'sigma': array([ 32.      ,  29.857
         3,   4,   5,   6,   7,   8,   9])}, {'kernel': ['linear'], 'l2re
       pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
       scoring=make_scorer(mean_squared_error, greater_is_better=False),
       verbose=0)
```

In [136]:
```
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_offs
        "mean_test_score","mean_train_score"]
df_toshow = df[cols_to_keep].fillna('-')
#df_toshow.sort_values(by=["mean_test_score"])
df_RBF = df_toshow[df_toshow['param_kernel']=='RBF']
df_RBF = df_RBF.sort_values(by=["mean_test_score"])
```

In [137]:
```
df_poly = df_toshow[df_toshow['param_kernel']=='polynomial']
df_poly = df_poly.sort_values(by=["mean_test_score"])
```

2.5.1 Print out best settings

In [138]:
```
#Store best parameter settings from the sorted parameters grid generated
best_pm_poly = df_poly.iloc[0]
best_pm_RBF = df_RBF.iloc[0]
print('Polynomial Kernel best parameter settings')
print(best_pm_poly)
print('')
```

```
        print('RBF Kernel best parameter settings')
        print(best_pm_RBF)
```

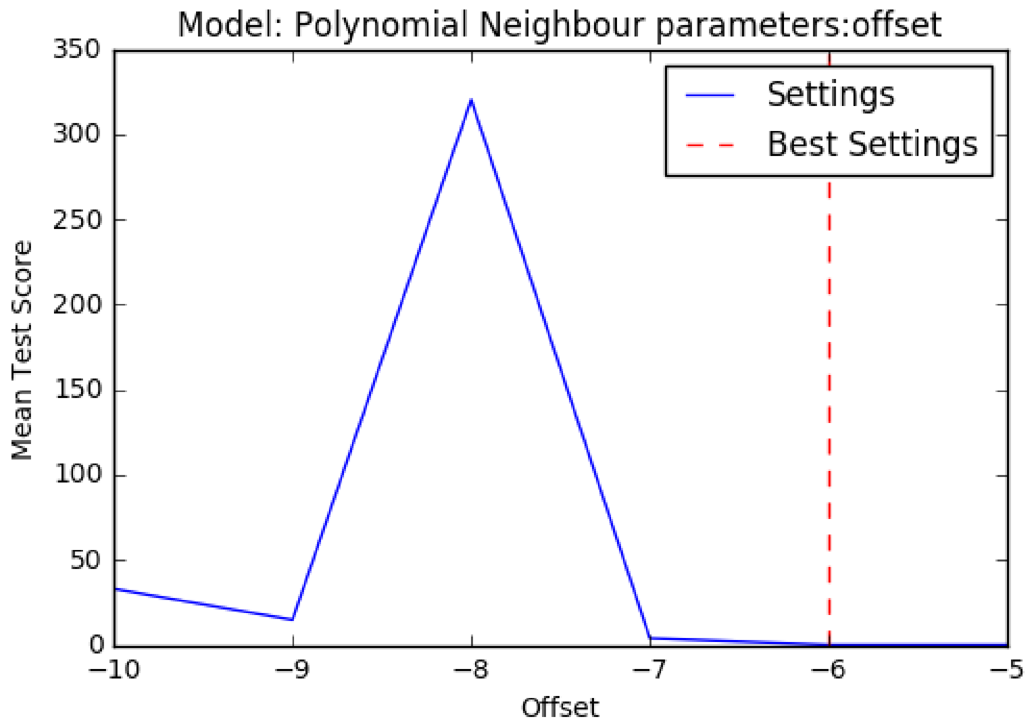
```
Polynomial Kernel best parameter settings
param_degree                 16
param_kernel         polynomial
param_l2reg           0.0473661
param_offset                 -6
param_sigma                   -
mean_test_score       0.0240397
mean_train_score      0.0339528
Name: 39884, dtype: object

RBF Kernel best parameter settings
param_degree                 -
param_kernel               RBF
param_l2reg           0.133972
param_offset                 -
param_sigma             0.0625
mean_test_score      0.0138071
mean_train_score      0.014427
Name: 7990, dtype: object
```

2.5.2 Print out neighbours of best settings

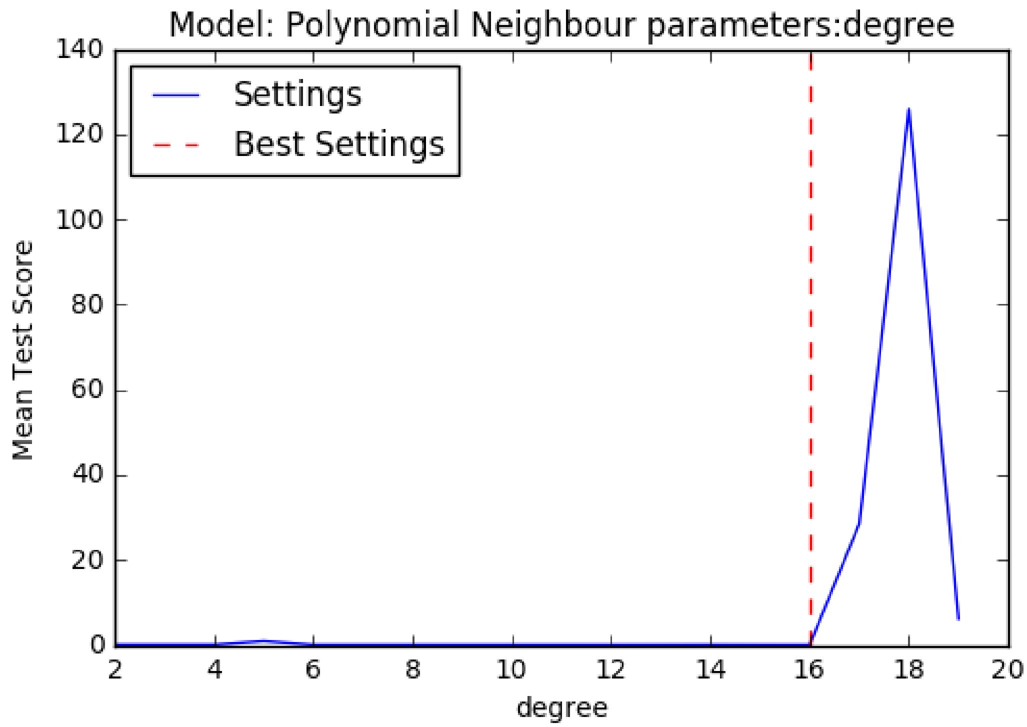
As we can see these settings are at local minimuns. Changing in any parameters to both direction can lead to a increase in test score.

```
In [172]: # Hold param_l2reg and degree compare changes in parm_offset
          df_poly_neighbours_offset = df_poly[(df_poly['param_l2reg']==best_pm_poly
                                              &(df_poly['param_degree']==best_pm_po
          .sort_values(by=['param_offset']).head(6)
```

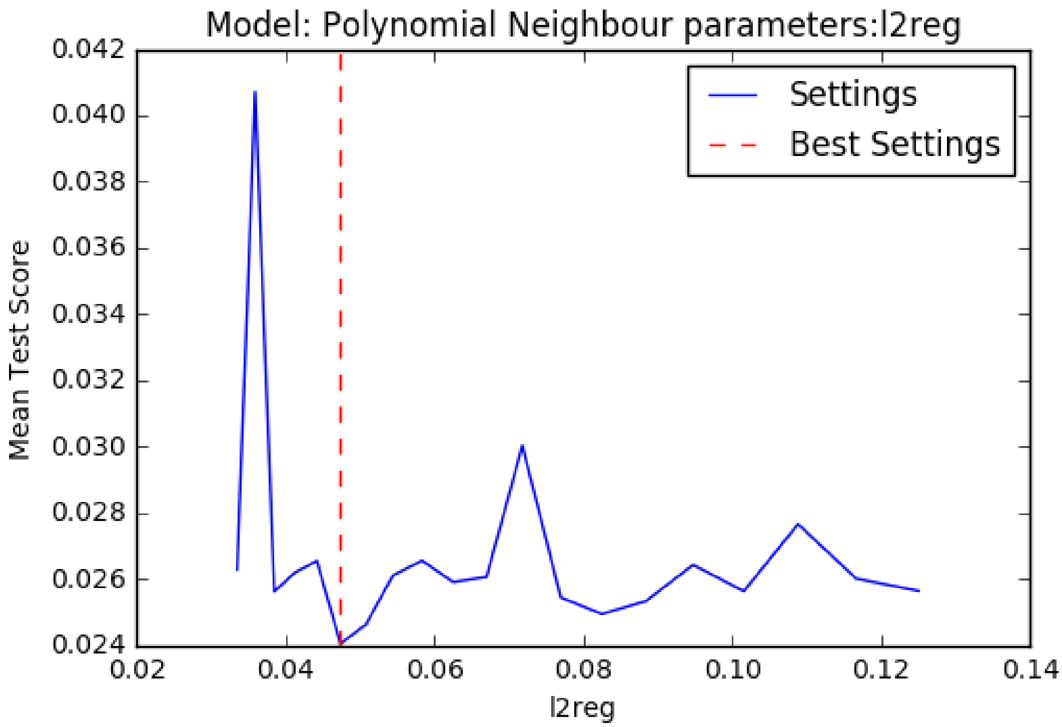
```
In [146]: plt.plot(df_poly_neighbours_offset['param_offset'],df_poly_neighbours_off
          plt.axvline(x=best_pm_poly['param_offset'],label='Best Settings',ls='dash
          plt.title('Model: Polynomial Neighbour parameters:offset')
          plt.xlabel('Offset')
          plt.ylabel('Mean Test Score')
          plt.legend(loc='best')
          plt.show()
```




```
In [153]: # Hold param_l2reg and offset compare changes in param_degree
          df_poly_neighbours_degree = df_poly[(df_poly['param_l2reg']==best_pm_poly
                                             & (df_poly['param_offset']==best_pm_po
          .sort_values(by=['param_degree'])

In [159]: plt.plot(df_poly_neighbours_degree['param_degree'],df_poly_neighbours_deg
          plt.axvline(x=best_pm_poly['param_degree'],label='Best Settings',ls='dash
          plt.title('Model: Polynomial Neighbour parameters:degree')
          plt.xlabel('degree')
          plt.ylabel('Mean Test Score')
          plt.legend(loc='best')
          plt.show()
```

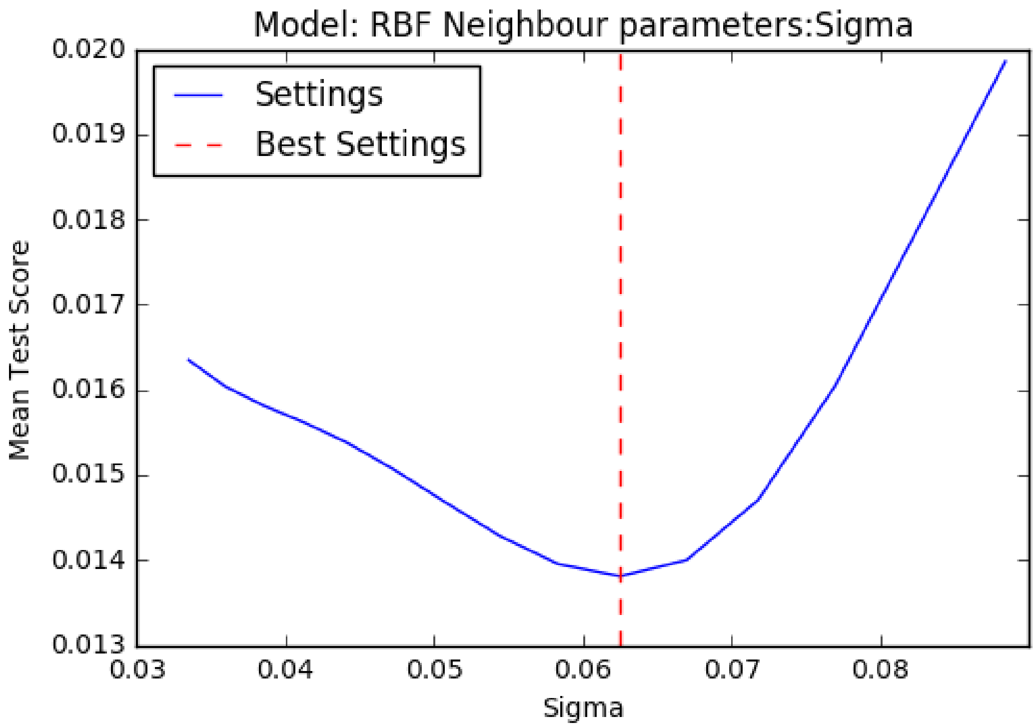



```
In [169]: # Hold param_degree and offset compare changes in param_degree
          df_poly_neighbours_l2reg = df_poly[(df_poly['param_degree']==best_pm_poly
                                                    &(df_poly['param_offset']==best_pm_po
          .sort_values(by=['param_l2reg']).head(20)

In [171]: plt.plot(df_poly_neighbours_l2reg['param_l2reg'],df_poly_neighbours_l2reg
          plt.axvline(x=best_pm_poly['param_l2reg'],label='Best Settings',ls='dashe
          plt.title('Model: Polynomial Neighbour parameters:l2reg')
          plt.xlabel('l2reg')
          plt.ylabel('Mean Test Score')
          plt.legend(loc='best')
          plt.show()
          plt.savefig('test.png')
```

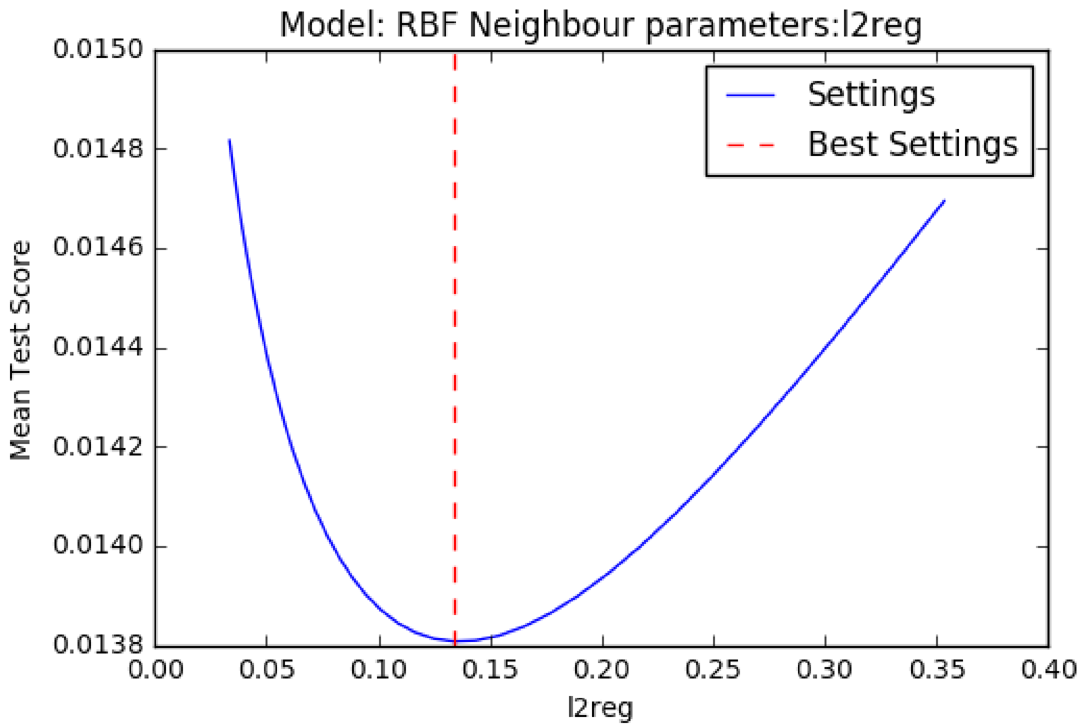



```
In [77]: df_RBF_neighbours_sigma = df_RBF[df_RBF['param_l2reg']==best_pm_RBF['param
```

```
In [88]: plt.plot(df_RBF_neighbours_sigma['param_sigma'],df_RBF_neighbours_sigma['m
         plt.axvline(x=best_pm_RBF['param_sigma'],label='Best Settings',ls='dashed'
         plt.title('Model: RBF Neighbour parameters:Sigma')
         plt.xlabel('Sigma')
         plt.ylabel('Mean Test Score')
         plt.legend(loc='best')
         plt.show()
```

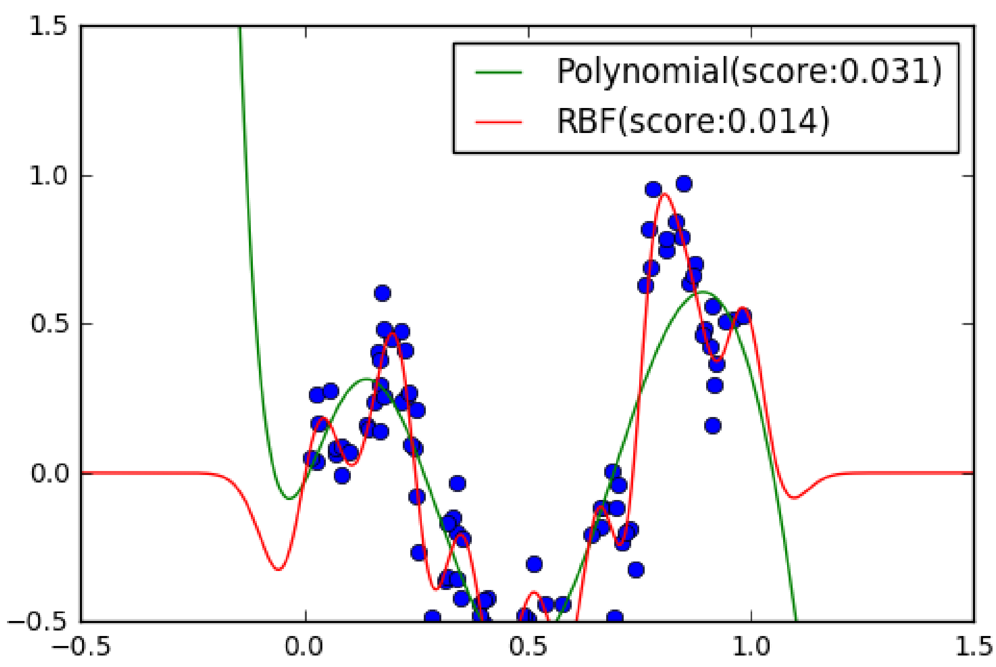



```
In [102]: df_RBF_neighbours_l2reg = df_RBF[df_RBF['param_sigma']==best_pm_RBF['para

In [103]: plt.plot(df_RBF_neighbours_l2reg['param_l2reg'],df_RBF_neighbours_l2reg['
          plt.axvline(x=best_pm_RBF['param_l2reg'],label='Best Settings',ls='dashed
          plt.title('Model: RBF Neighbour parameters:l2reg')
          plt.xlabel('l2reg')
          plt.ylabel('Mean Test Score')
          plt.legend(loc='best')
          plt.show()
```


Model: RBF Neighbour parameters:l2reg
Settings
Best Settings
Mean Test Score
0.0150
0.0148
0.0146
0.0144
0.0142
0.0140
0.0138
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
l2reg

2.6 8-3-6

```
In [23]: #Best setting polynomial kernel
         plot_step = .001
         xpts = np.arange(-0.5 , 1.5, plot_step).reshape(-1,1)
         plt.plot(x_train,y_train,'o')
         degree=best_pm_poly['param_degree']
         offset=best_pm_poly['param_offset']
         sigma= best_pm_RBF['param_sigma']
         score_RBF = best_pm_RBF['mean_test_score']
         score_poly = best_pm_poly['mean_test_score']
         for l2reg in [best_pm_poly['param_l2reg']]:
             k = functools.partial(polynomial_kernel,degree=degree,offset=offset)
             f = train_kernel_ridge_regression(x_train, y_train, k, l2reg=l2reg)
             label = "Polynomial(score:%.3f)"%(score_poly)
             plt.plot(xpts, f.predict(xpts), label=label)
             k = functools.partial(RBF_kernel, sigma=sigma)
             f = train_kernel_ridge_regression(x_train, y_train, k, l2reg=l2reg)
             label = 'RBF(score:%.3f)'%(score_RBF)
             plt.plot(xpts, f.predict(xpts), label=label)
         plt.legend(loc = 'best')
         plt.ylim(-0.5,1.5)
         plt.show()
```



2.6.1 Comment

We observed that RBF kernel provides stronger fitting to training data than Polynomial kernel since RBF can model infinite dimensional features mapping.

2.7 8-3-7

As we have derived in Assignment 1, the bayes decision function for square loss for given X Y is $E[y|x]$. In this case $E[y|x] = E[f(x) + \epsilon|x] = f(x)$. Also we derived that the associated bayes risk is $var(y|x)$. In this case, since $f(x)$ is uncorrelated with ϵ, $var(y|x) = var(f(x) + \epsilon|x) + var(f(x)|x) + var(\epsilon|x) = var(\epsilon) = 0.1$