2. Kernel Matrices

Problem 2.1. Consider a set of vectors $S = \{x_1, \ldots, x_m\}$. Let X denote the matrix whose rows are these vectors. Form the Gram matrix $K = XX^T$. Show that knowing K is equivalent to knowing the set of pairwise distances among the vectors in S as well as the vector lengths.

Solution From the definition of the Gram matrix, knowing K is equivalent to knowing $\langle x, x' \rangle \; \forall \; x, x' \in S$. Now let $x, x' \in S$ and observe that

$$\begin{aligned}
\langle x, x' \rangle &= \langle x' + (x - x'), x + (x' - x) \rangle \\
&= \langle x', x \rangle + \langle x', x' \rangle + \langle x', -x \rangle + \langle x, x \rangle + \langle -x', x \rangle + \langle x - x', x' - x \rangle \\
&= \|x\|^2 + \|x'\|^2 - \|x - x', x - x'\|^2 - \langle x', x \rangle \\
&= \frac{\|x\|^2 + \|x'\|^2 - \|x - x', x - x'\|^2}{2}
\end{aligned}$$

Thus it is possible to express elements of K in terms of the distance between vectors in S as well as the vector lengths. We can conclude that knowing K is equivalent to knowing the set of pairwise distances among the vectors in S as well as the vector lengths.

3. Kernel Ridge Regression

Problem 3.1. Show that for w to be a minimizer of $J(w)$, we must have $X^TXw + \lambda Iw = X^Ty$. Show that the minimizer of $J(w)$ is $w = (X^TX + \lambda I)^{-1}X^Ty$. Justify that the matrix $X^TX + \lambda I$ is invertible, for $\lambda > 0$.

Solution

1. For w to be a minimizer of $J(w)$ we must have

$$\begin{aligned} J'(w) &= 2(Xw - y)^TX + 2\lambda w^T \\ &= 2(Xw)^TX - 2y^TX + 2\lambda w^T \\ &= 2w^TX^TX - 2y^TX + 2\lambda w^T \\ &= 0 \end{aligned}$$

So then

$$y^TX = w^TX^TX + \lambda w^T$$

And by taking the transpose of both sides we get

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

2. Solving for w we get

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

3. Note that X^TX is positive semi-definite. Let $v \in \mathbf{R}^d$ and $\lambda \in \mathbf{R}$ and observe that

$$\begin{aligned} v^T(X^TX + \lambda I)v &= v^TX^TXv + \lambda v^TX^TXv \\ &= v^TX^TXv + \lambda v^Tv \end{aligned}$$

But $v^TX^TXv \geq 0$ since X^TX is positive semi-definite and if $v \neq 0$ then $\lambda v^Tv > 0$. Thus $v^T(X^TX + \lambda I)v > 0$ and so $X^TX + \lambda I$ is symmetric positive definite. As a result it is also inevitable.

Problem 3.2. Rewrite $X^TXw + \lambda Iw = X^Ty$ as $w = \frac{1}{\lambda}(X^Ty - X^TXw)$. Based on this, show that we can write $w = X^T\alpha$ for some α, and give an expression for α.

Solution Note that

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

Then setting $\alpha = \frac{1}{\lambda}(y - Xw)$ we have

$$w = X^T\alpha$$

Problem 3.3. Based on the fact that $w = X^T \alpha$, explain why we say w is "in the span of the data."

Solution Since $w = X^T \alpha$, we can also express w as $w = \sum_{i=1}^{n} x_i \alpha_i$. Thus w is a linear combination of the training data X and as such is in the span of the training data.

Problem 3.4. Show that $\alpha = (\lambda I + XX^T)^{-1}y$. Note that XX^T is the kernel matrix for the standard vector dot product.

Solution Observe that

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

Problem 3.5. Give a kernelized expression for the Xw, the predicted values on the training points.

Solution Recall that

$$\alpha = (\lambda I + XX^T)^{-1}y$$

Then

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

Problem 3.6. Give an expression for the prediction $f(x) = x^T w^*$ for a new point x, not in the training set. The expression should only involve x via inner products with other x's. [Hint: It is often convenient to define the column vector

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

to simplify the expression.]

Solution Recall that $w^* = X^T \alpha^*$ and that $\alpha^* = (\lambda I + XX^T)^{-1} y$. Also note that

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

Then

$$\begin{aligned} f(x) &= x^T w^* \\ &= x^T X^T \alpha^* \\ &= x^T X^T (\lambda I + XX^T) y \\ &= k_x (\lambda I + K) y \end{aligned}$$

4. Pegasos and SSGD for ℓ_2-regularized ERM

Problem 4.1. For each $i = 1, \dots, n$, let $g_i(w)$ be a subgradient of $J_i(w)$ at $w \in \mathbf{R}^d$. Let $v_i(w)$ be a subgradient of $\ell_i(w)$ at w. Give an expression for $g_i(w)$ in terms of w and $v_i(w)$

Solution

$$g_i(w) = \lambda w + v_i(w)$$

Problem . Show that $\mathbb{E}g_i(w) \in \partial J(w)$, where the expectation is over the randomly selected $i \in 1, \ldots, n$.

Solution Let $g(w)$ be a subgradient of $J(w)$ and $v_i(w)$ a subgradient of $l_i(w)$ Now observe that

$$\begin{aligned} g(w) &= \lambda w + \frac{1}{n}\sum_{i=1}^{n} v_i(w) \\ &= \lambda w + \mathbb{E}v_i(w) \end{aligned}$$

And that

$$\begin{aligned} \mathbb{E}g_i(w) &= \mathbb{E}\big(\lambda w + v_i(w)\big) \\ &= \lambda w + \mathbb{E}v_i(w) \end{aligned}$$

Then $\mathbb{E}g_i(w) = g(w) \in \partial J(w)$

Problem 4.3. Now suppose we are carrying out SSGD with the Pegasos step-size $\eta^{(t)} = 1/(\lambda t)$, $t = 1, 2, \ldots.$, starting from $w^{(1)} = 0$. In the t'th step, suppose we select the ith point and thus take the step $w^{(t+1)} = w^{(t)} - \eta^{(t)} g_i(w^{(t)})$. Let's write $v^{(t)} = v_i(w^{(t)})$, which is the subgradient of the loss part of $J_i(w^{(t)})$ that is used in step t. Show that

$$w^{(t+1)} = -\frac{1}{\lambda t}\sum_{\tau=1}^{t} v^{(\tau)}$$

Solution

1. Base case $t = 1$
   Observe that

$$\begin{aligned} w^{(2)} = w^{(1)} &= \frac{1}{\lambda} * \lambda w^{(1)} - \frac{1}{\lambda} v^{(1)} \\ &= -\frac{1}{\lambda} v^{(1)} \\ &= -\frac{1}{\lambda \cdot 1}\sum_{\tau=1}^{1} v^{(\tau)} \\ &= -\frac{1}{\lambda t}\sum_{\tau=1}^{t} v^{(\tau)} \end{aligned}$$

2. Induction step.
   Assume that $w^{(t)} = -\frac{1}{\lambda(t-1)}\sum_{\tau=1}^{t-1} v^{(\tau)}$. Now observe that

$$\begin{aligned} w^{(t+1)} &= w^{(t)} - \frac{1}{t} w^{(t)} - \frac{1}{\lambda t} v^{(t)} \\ &= \frac{t-1}{t} w^{(t)} - \frac{1}{\lambda t} v^{(t)} \\ &= -\frac{1}{\lambda t}\sum_{\tau=1}^{t-1} v^{(\tau)} - \frac{1}{\lambda t} v^{(t)} \\ &= -\frac{1}{\lambda t}\sum_{\tau=1}^{t} v^{(\tau)} \end{aligned}$$

Problem 4.a. Explain how Algorithm 1 can be implemented so that, if x_j has s nonzero entries, then we only need to do $O(s)$ memory accesses in every pass through the loop.

Solution If we implement the algorithm with sparse matrices or dictionaries, then when computing $y_j\langle w^{(t)}, x_j\rangle$ and $y_j x_j$ we only need to access the s nonzero elements of x_j and their corresponding elements in y_j and $w^{(t)}$. The remaining operations are $O(1)$ and so in total we only need to do $O(s)$ memory accesses.

5. Kernelized Pegasos

Problem . Kernelize the expression for the margin. That is, show that $y_j \left\langle w^{(t)}, x_j \right\rangle = y_j K_{j\cdot} \alpha^{(t)}$, where $k(x_i, x_j) = \langle x_i, x_j \rangle$ and $K_{j\cdot}$ denotes the jth row of the kernel matrix K corresponding to kernel k.

Solution Observe that

$$\begin{aligned}
\langle w^{(t)}, x_j \rangle &= \langle \sum_{i=1}^{n} \alpha_i^{(t)} x_i, x_j \rangle \\
&= \sum_{i=1}^{n} \alpha_i^{(t)} \langle x_i, x_j \rangle \\
&= \begin{bmatrix} \langle x_1, x_j \rangle & \dots & \langle x_n, x_j \rangle \end{bmatrix} \alpha^{(t)} \\
&= \begin{bmatrix} k(x_1, x_j) & \dots & k(x_n, x_j) \end{bmatrix} \alpha^{(t)} \\
&= \begin{bmatrix} k(x_j, x_1) & \dots & k(x_j, x_n) \end{bmatrix} \alpha^{(t)} \\
&= K_j \alpha^{(t)}
\end{aligned}$$

Thus $y_j \left\langle w^{(t)}, x_j \right\rangle = y_j K_{j\cdot} \alpha^{(t)}$

Problem 5.2. Suppose that $w^{(t)} = \sum_{i=1}^n \alpha_i^{(t)} x_i$ and for the next step we have selected a point (x_j, y_j) that does not have a margin violation. Give an update expression for $\alpha^{(t+1)}$ so that $w^{(t+1)} = \sum_{i=1}^n \alpha_i^{(t+1)} x_i$.

Solution When (x_j, y_j) does not result in a margin violation, $w^{(t+1)} = (1 - \eta^{(t)}\lambda)w^{(t)}$. Then in order to have $w^{(t+1)} = \sum_{i=1}^n \alpha_i^{(t+1)} x_i$, we must also have

$$\begin{aligned}(1 - \eta^{(t)}\lambda)w^{(t)} &= \sum_{i=1}^n \alpha_i^{(t+1)} x_i \\ (1 - \eta^{(t)}\lambda)\sum_{i=1}^n \alpha_i^{(t)} x_i &= \sum_{i=1}^n \alpha_i^{(t+1)} x_i\end{aligned}$$

Now setting $\alpha^{(t+1)} = (1 - \eta^{(t)}\lambda)\alpha^{(t)}$ preserves the equality above. Thus this is a valid update rule.

Problem 5.3. Repeat the previous problem, but for the case that (x_j, y_j) has a margin violation. Then give the full pseudocode for kernelized Pegasos. You may assume that you receive the kernel matrix K as input, along with the labels $y_1, \dots, y_n \in \{-1, 1\}$.

Solution Note that when we have a margin error,

$$\begin{aligned} w^{(t+1)} &= (1 - \eta^{(t)}\lambda)w^{(t)} + \eta^{(t)} y_j x_j \\ &= (1 - \eta^{(t)}\lambda)\sum_{i=1}^{n} \alpha_i^{(t)} x_i + \eta^{(t)} y_j x_j \\ &= \sum_{i=1}^{n} \big((1 - \eta^{(t)}\lambda)\alpha_i^{(t)} + \mathbb{1}[i = j]\eta^{(t)} y_j\big) x_i \end{aligned}$$

Thus an appropriate update rule for α is

$$\alpha_i^{(t+1)} = (1 - \eta^{(t)}\lambda)\alpha_i^{(t)} + \mathbb{1}[i = j]\eta^{(t)} y_j$$

Problem 5.4. While the direct implementation of the original Pegasos required updating all entries of w in every step, a direct kernelization of Algorithm 2, as we have done above, leads to updating all entries of α in every step. Give a version of the kernelized Pegasos algorithm that does not suffer from this inefficiency. You may try splitting the scale and direction similar to the approach of the previous problem set, or you may use a decomposition based on Algorithm 1 from the optional problem 4 above.

Solution

7. Representer Theorem

Problem 7.1. Let M be a closed subspace of a Hilbert space $\mathcal{H}$. For any $x \in \mathcal{H}$, let $m_0 = \text{Proj}_M x$ be the projection of x onto M. By the Projection Theorem, we know that $(x - m_0) \perp M$. Then by the Pythagorean Theorem, we know $\|x\|^2 = \|m_0\|^2 + \|x - m_0\|^2$. From this we concluded in lecture that $\|m_0\| \leq \|x\|$. Show that we have $\|m_0\| = \|x\|$ only when $m_0 = x$.

Solution First, assume that $m_0 = x$. Then clearly $\|m_0\| = \|x\|$.
Now it remains to show that $\|m_0\| \neq \|x\|$ when $m_0 \neq x$.

Assume that $m_0 \neq x$. Also note that since the inner product is positive-definite,
$\|m_0\|^2 \neq \|x\|^2$ iff $\|m_0\| \neq \|x\|$. Then

$$\begin{aligned}
\|m_0\|^2 &= \|x - (x - m_0)\|^2 \\
&= \|x\|^2 - \|x - m_0\|^2 && \text{Pythagorean Theorem} \\
&= \|x\|^2 - \langle x - m_0, x - m_0 \rangle
\end{aligned}$$

Now since $x \neq m_0$, $x - m_0 \neq 0$. Then since the inner product is positive-definite,
$\langle x - m_0, x - m_0 \rangle > 0$. Thus $\|m_0\|^2 \neq \|x\|^2$ and so $\|m_0\| \neq \|x\|$.

Problem 7.2. Give the proof of the Representer Theorem in the case that R is strictly increasing. That is, show that if R is strictly increasing, then all minimizers have this form claimed.

Solution Proof by contradiction:

Assume R is strictly increasing and w^* is a minimizer of $J(w)$. Let $w^*_M = Proj_M w$ where $M = span(x_1, \ldots, x_n)$. Note that for any $m \in M$, $\exists\ \alpha$ such that $m = \sum_{i=1}^n \alpha_i \psi(x_i)$. By way of contradiction also assume that $w^* \notin M$.

Since $w^* \notin M$, $w^* \neq w^*_M$. Furthermore, $\|w^*_M\| < \|w^*\|$ since projections reduce norms. Since R is strictly increasing, $R\big(\|w^*_M\|\big) < R\big(\|w^*\|\big)$. Now let $x \in M$ and observe that

$$\begin{aligned}
\langle w^*, x\rangle &= \langle w^*_M + w^* - w^*_M, x\rangle \\
&= \langle w^*_M, x\rangle + \langle w^* - w^*_M, x\rangle \\
&= \langle w^*_M, x\rangle && \text{since } w^* - w^*_M \perp M
\end{aligned}$$

Thus

$$L\big(\langle w^*, \psi(x_1)\rangle, \ldots, \langle w^*, \psi(x_n)\rangle\big) = L\big(\langle w^*_M, \psi(x_1)\rangle, \ldots, \langle w^*_M, \psi(x_n)\rangle\big)$$

Therefore $J(w^*_M) < J(w^*)$ and so w^* is not a minimizer of $J(w)$.

We have shown that when R is strictly increasing, we cannot have any minimizers outside of M. Thus all minimizers are elements of M and can be written as $w^* = \sum_{i=1}^n \alpha_i \psi(x_i)$.

Problem 7.3. Suppose that $R : \mathbf{R}^{\geq 0} \to \mathbf{R}$ and $L : \mathbf{R}^n \to \mathbf{R}$ are both convex functions. Use properties of convex functions to **show that** $w \mapsto L\left(\langle w, \psi(x_1)\rangle, \ldots, \langle w, \psi(x_n)\rangle\right)$ is a convex function of w, and then that $J(w)$ is also a convex function of w. For simplicity, you may assume that our feature space is $\mathbf{R}^d$, rather than a generic Hilbert space. You may also use the fact that the composition of a convex function and an affine function is convex. That is, suppose $f : \mathbf{R}^n \to \mathbf{R}$, $A \in \mathbf{R}^{n\times m}$ and $b \in \mathbf{R}^n$. Define $g : \mathbf{R}^m \to \mathbf{R}$ by $g(x) = f\left(Ax + b\right)$. Then if f is convex, then so is g. From this exercise, **we can conclude** that if L and R are convex, then J does have a minimizer of the form $w^* = \sum_{i=1}^n \alpha_i \psi(x_i)$, and if R is also strictly increasing, then all minimizers of J have this form.

Solution Note that in $\mathbf{R}^n$, $\langle x, y\rangle = x^T y = y^T x$. Also note that since the sum of convex functions is also convex, if $L\left(\langle w, \psi(x_1)\rangle, \ldots, \langle w, \psi(x_n)\rangle\right)$ is convex then $J(w)$ is convex. Now observe that

$$\begin{bmatrix} \langle w, \psi(x_1)\rangle \\ \vdots \\ \langle w, \psi(x_n)\rangle \end{bmatrix} = \begin{bmatrix} w^T\psi(x_1) \\ \vdots \\ w^T\psi(x_n) \end{bmatrix} = \begin{bmatrix} — & \psi(x_1) & — \\ & \vdots & \\ — & \psi(x_n) & — \end{bmatrix} w$$

Thus $\begin{bmatrix} \langle w, \psi(x_1)\rangle \\ \vdots \\ \langle w, \psi(x_n)\rangle \end{bmatrix}$ can be represented as an affine function of w.

Since the composition of a convex function and an affine function is also convex, $L\left(\langle w, \psi(x_1)\rangle, \ldots, \langle w, \psi(x_n)\rangle\right)$ is a convex function of w. Therefore $J(w)$ is convex.

8. Ivanov and Tikhonov Regularization

Problem 8.1. Suppose that for some $\lambda \geq 0$ we have the Tikhonov regularization solution

$$f^* \in \underset{f \in \mathcal{F}}{\arg\min} \left[\phi(f) + \lambda \Omega(f)\right]. \tag{1}$$

Show that f^* is also an Ivanov solution. That is, $\exists r \geq 0$ such that

$$f^* \in \underset{f \in \mathcal{F}}{\arg\min}\, \phi(f) \text{ subject to } \Omega(f) \leq r. \tag{2}$$

Solution Suppose $f^* \in \arg\min_{f \in \mathcal{F}} \left[\phi(f) + \lambda \Omega(f)\right]$ and let $r = \Omega(f^*)$. By way of contradiction, suppose that the solution to the Ivanov form is $f' \neq f^*$. Then we have

$$\Omega(f') \leq r = \Omega(f^*)$$

and

$$\phi(f') < \phi(f^*).$$

Thus

$$\phi(f') + \lambda \Omega(f') < \phi(f^*) + \lambda \Omega(f^*).$$

This contradicts our original assumption that f^* is a minimizer of the Tikhonov problem. Thus $f' = f^*$ And so f^* is a solution to both the Tikhonov and the Ivanov form of the problem.

Problem 8.2.1. Write the Lagrangian $L(w, \lambda)$ for the Ivanov optimization problem.

Solution

$$L(w, \lambda) = \phi(w) + \lambda(\Omega(w) - r)$$

Problem 8.2.2. Write the dual optimization problem in terms of the dual objective function $g(\lambda)$, and give an expression for $g(\lambda)$.

Solution The dual optimization problem is to find

$$d^* = \sup_{\lambda \succ 0} \inf_w L(w, \lambda) = \sup_{\lambda \succ 0} g(\lambda)$$

Where $g(\lambda) = \inf_w \big(\phi(w) + \lambda(\Omega(w) - r)\big)$

Problem 8.2.3. We assumed that the dual solution is attained, so let $\lambda^* \in \arg\max_{\lambda \geq 0} g(\lambda)$. We also assumed strong duality, which implies $\phi(w^*) = g(\lambda^*)$. Show that the minimum in the expression for $g(\lambda^*)$ is attained at w^*. **Conclude the proof** by showing that for the choice of $\lambda = \lambda^*$, we have $w^* \in \arg\min_{w \in \mathbf{R}^d} \left[\phi(w) + \lambda^* \Omega(w)\right]$.

Solution Observe that

$$\begin{aligned}
\phi(w^*) &= g(\lambda^*) \\
&= \inf_w L(w, \lambda^*) \\
&\leq L(w^*, \lambda^*) \\
&= \phi(w^*) + \lambda^*(\Omega(w^*) - r) \\
&\leq \phi(w^*) && \text{since } \Omega(w^*) - r \leq 0
\end{aligned}$$

Thus we have equality throughout the expression above and so $\inf_w L(w, \lambda^*) = L(w^*, \lambda^*)$. We can now conclude that, since $\phi(w^*) = \inf_w \left[\phi(w^*) + \lambda^*(\Omega(w^*) - r)\right]$,

$$\begin{aligned}
w^* &\in arg\, min_w \ \phi(w) + \lambda^*(\Omega(w) - r) \\
&= arg\, min_w \ \phi(w) + \lambda^* \Omega(w).
\end{aligned}$$

Problem 8.3. Show that the Ivanov form of ridge regression

$$\begin{array}{ll} \text{minimize} & \sum_{i=1}^{n} \left(y_i - w^T x_i\right)^2 \\ \text{subject to} & w^T w \leq r. \end{array}$$

is a convex optimization problem with a strictly feasible point, so long as $r > 0$. (Thus implying the Ivanov and Tikhonov forms of ridge regression are equivalent when $r > 0$.)

Solution Assume that $r > 0$.
To show that Slater's condition holds, we need to find a w such that

$$w^T w - r < 0$$

Let $w = 0$, then since $r > 0$, we have $w^T w - r < 0$. Thus a solution has been found and Slater's condition holds.

hwk4-skeleton-code

March 7, 2019

```
In [3]: import numpy as np
        import matplotlib.pyplot as plt
        import sklearn
        import scipy.spatial
        import functools
        from scipy.spatial.distance import cdist
        import warnings
        warnings.filterwarnings('ignore')

        %matplotlib inline
```

6.2 Kernels and Kernel Machines

6.2.1

Write functions to compute the RBF and polynomial kernels.

```
In [2]: ### Kernel function generators
        def linear_kernel(X1, X2):
            """
            Computes the linear kernel between two sets of vectors.
            Args:
                X1 - an n1xd matrix with vectors x1_1,...,x1_n1 in the rows
                X2 - an n2xd matrix with vectors x2_1,...,x2_n2 in the rows
            Returns:
                matrix of size n1xn2, with x1_i^T x2_j in position i,j
            """
            return np.dot(X1,np.transpose(X2))

        def RBF_kernel(X1,X2,sigma):
            """
            Computes the RBF kernel between two sets of vectors
            Args:
                X1 - an n1xd matrix with vectors x1_1,...,x1_n1 in the rows
                X2 - an n2xd matrix with vectors x2_1,...,x2_n2 in the rows
                sigma - the bandwidth (i.e. standard deviation) for the RBF/Gaussian kernel
            Returns:
                matrix of size n1xn2, with exp(-||x1_i-x2_j||^2/(2 sigma^2)) in position i,j
```

```
    """
    return np.exp(-cdist(X1,X2,'sqeuclidean')/(2*sigma**2))

def polynomial_kernel(X1, X2, offset, degree):
    """
    Computes the inhomogeneous polynomial kernel between two sets of vectors
    Args:
        X1 - an n1xd matrix with vectors x1_1,...,x1_n1 in the rows
        X2 - an n2xd matrix with vectors x2_1,...,x2_n2 in the rows
        offset, degree - two parameters for the kernel
    Returns:
        matrix of size n1xn2, with (offset + <x1_i,x2_j>)^degree in position i,j
    """
    return (offset + X1@X2.T)**degree
```

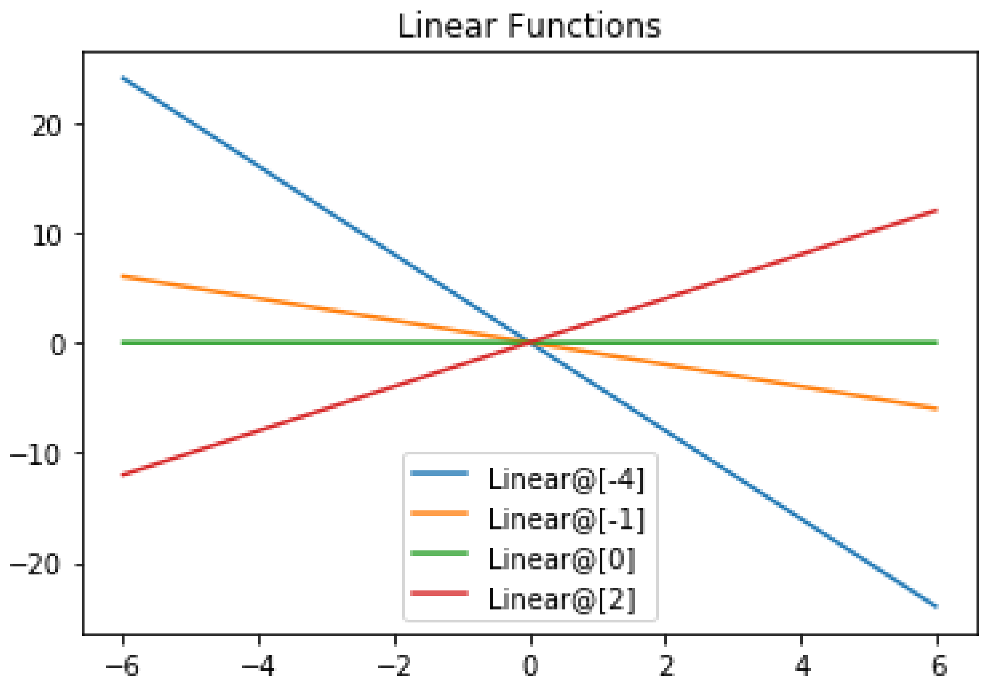
6.2.2

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.

```
In [3]: x0 = np.array([[-4],[-1],[0],[2]])
        linear_kernel(x0,x0)

Out[3]: array([[16,  4,  0, -8],
               [ 4,  1,  0, -2],
               [ 0,  0,  0,  0],
               [-8, -2,  0,  4]])
```

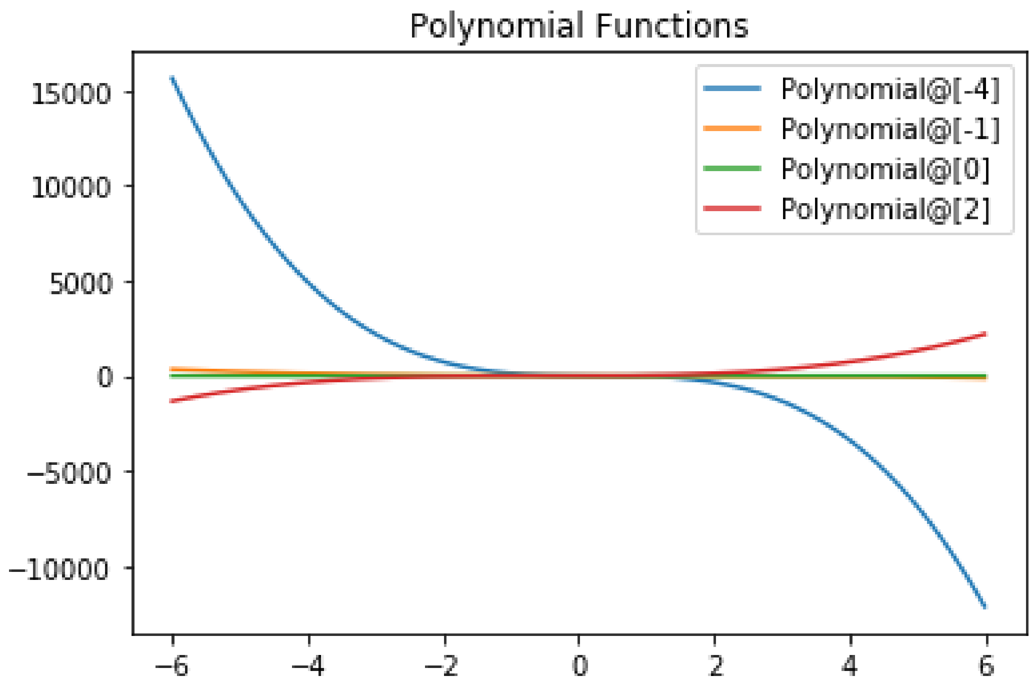
6.2.3.a

```
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 = 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.title('Linear Functions')
        plt.show()
```



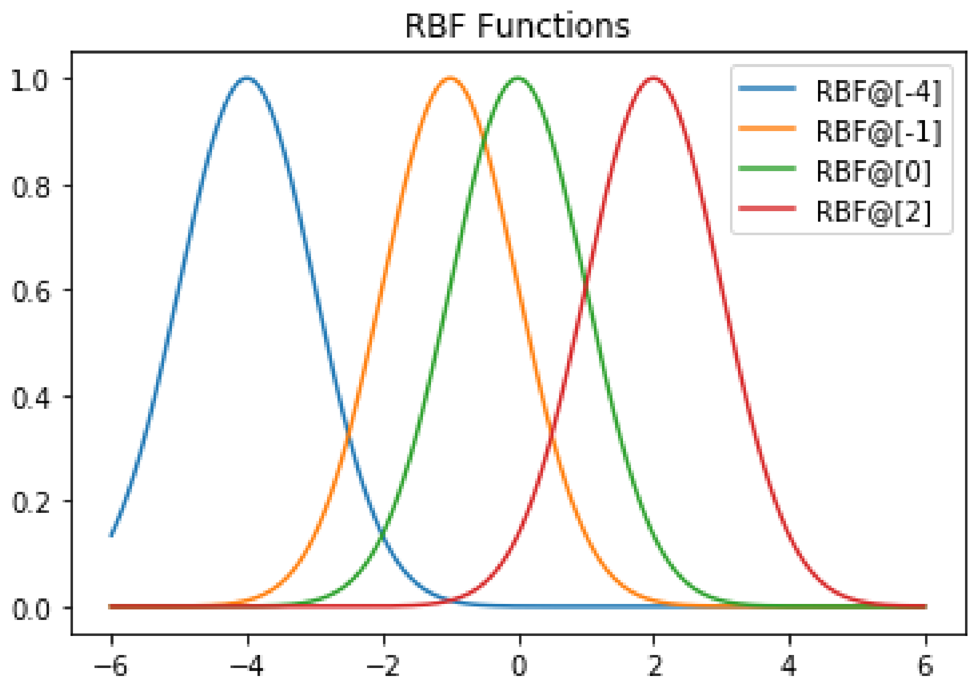
6.2.3.b

```
In [5]: # 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.title('Polynomial Functions')
        plt.show()
```



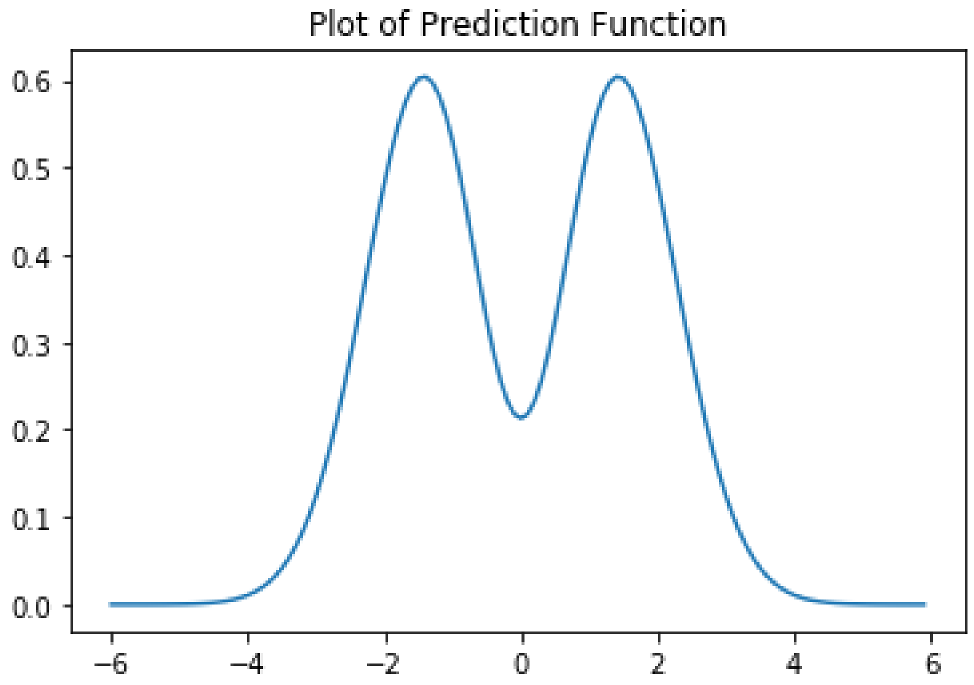
6.2.3.c

```
In [6]: # 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.title('RBF Functions')
        plt.show()
```



6.2.3.d

```
In [7]: class Kernel_Machine(object):
            def __init__(self, kernel, prototype_points, weights):
                """
                Args:
                    kernel(X1,X2) - a function return the cross-kernel matrix between rows of X1
                    prototype_points - an Rxd matrix with rows mu_1,...,mu_R
                    weights - a vector of length R with entries w_1,...,w_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.  Specifically, jt
                        Sum_{i=1}^R w_i k(x_j, mu_i)
                """
                return self.kernel(X, self.prototype_points)@self.weights

In [8]: from functools import partial
        kernel = partial(RBF_kernel, sigma=1)
        x = np.array([[-1],[0],[1]])
        w = np.array([1,-1,1])
        model = Kernel_Machine(kernel=kernel, prototype_points=x, weights=w)
        x_pred = np.arange(-6.0, 6, 0.1).reshape(-1,1)
        y_pred = model.predict(x_pred)
        plt.plot(x_pred, y_pred)
        plt.title("Plot of Prediction Function")
        plt.show()
```


Plot of Prediction Function
0.6
0.5
0.4
0.3
0.2
0.1
0.0
−6
−4
−2
0
2
4
6

6.3 Kernel Ridge Regression

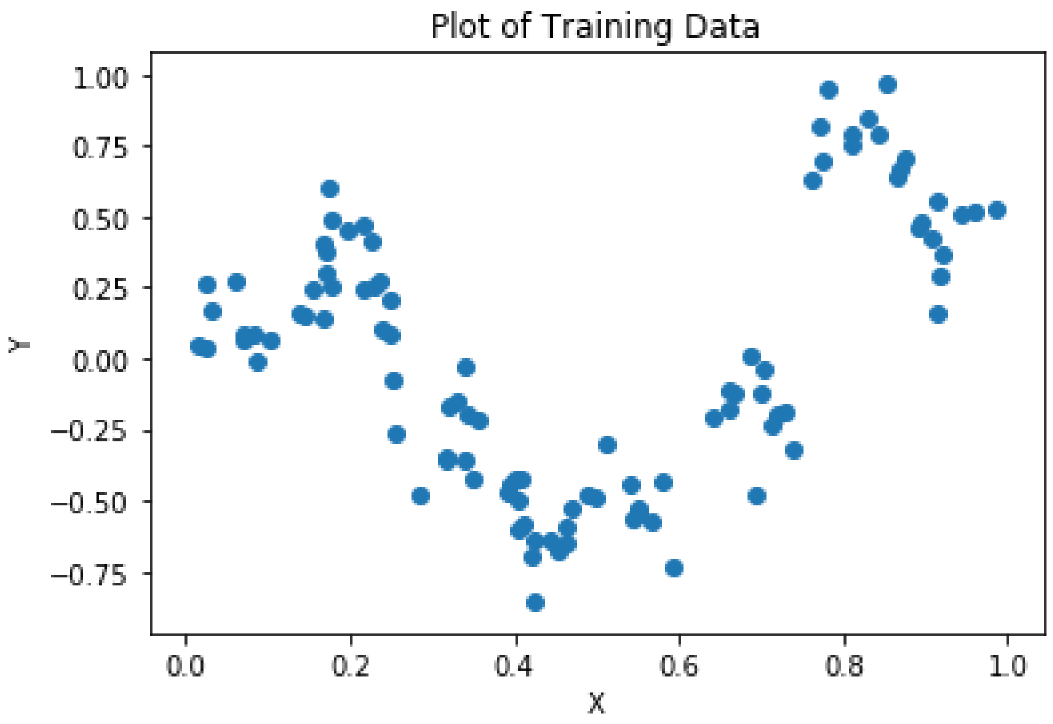
Load train & test data; Convert to column vectors so it generalizes well to data in higher dimensions.

```
In [5]: data_train,data_test = np.loadtxt("krr-train.txt"),np.loadtxt("krr-test.txt")
        x_train, y_train = data_train[:,0].reshape(-1,1),data_train[:,1].reshape(-1,1)
        x_test, y_test = data_test[:,0].reshape(-1,1),data_test[:,1].reshape(-1,1)
```

6.3.1

Plot the training data

```
In [10]: plt.scatter(x_train, y_train)
         plt.xlabel('X')
         plt.ylabel('Y')
         plt.title('Plot of Training Data')
         plt.show()
```



6.3.2

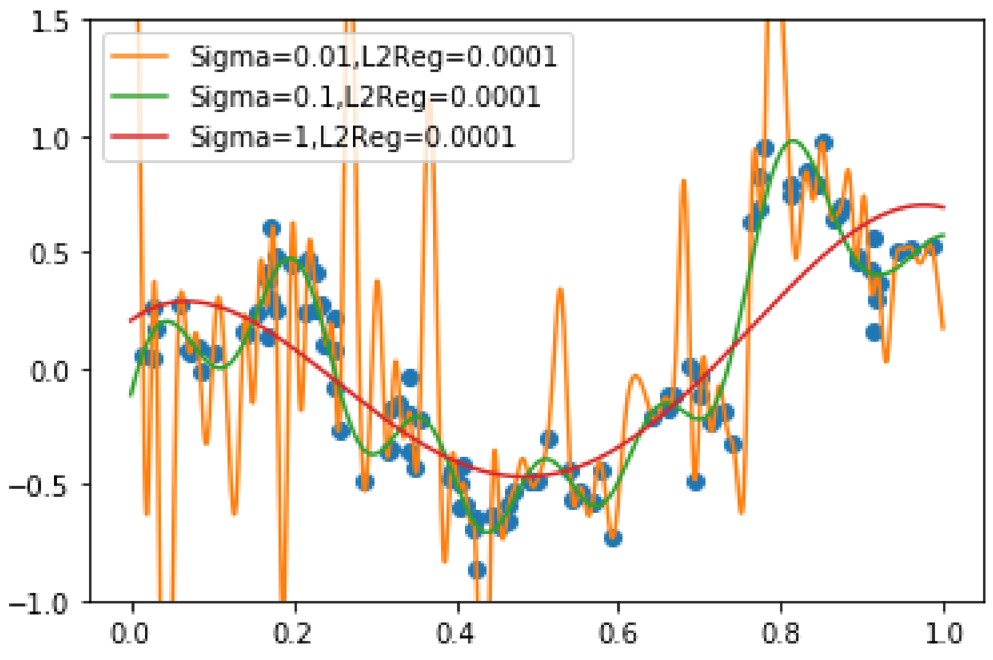
Complete the function train_kernel_ridge_regression

```
In [11]: def train_kernel_ridge_regression(X, y, kernel, l2reg):
             K = kernel(X,X)
             alpha = np.linalg.inv(l2reg*np.eye(len(X)) + K)@y
             return Kernel_Machine(kernel, X, alpha)
```

6.3.3

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?

```
In [12]: 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()
```

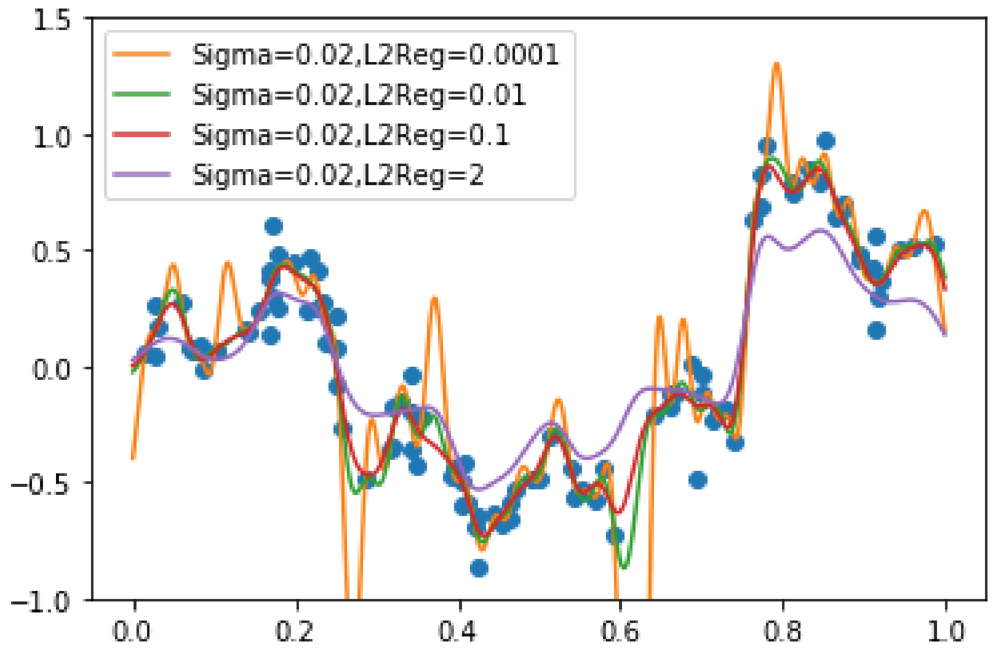


Lower values of σ are more prone to overfitting. With lower values of σ, the final prediction function is highly dependednt on local values of the training data.

6.3.4

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 $\lambda \to \infty$?

```
In [13]: plot_step = .001
         xpts = np.arange(0 , 1, plot_step).reshape(-1,1)
         plt.plot(x_train,y_train,'o')
         sigma= .02
         for l2reg in [.0001,.01,.1,2]:
             k = functools.partial(RBF_kernel, sigma=sigma)
             f = train_kernel_ridge_regression(x_train, y_train, k, l2reg=l2reg)
             label = "Sigma="+str(sigma)+",L2Reg="+str(l2reg)
             plt.plot(xpts, f.predict(xpts), label=label)
         plt.legend(loc = 'best')
         plt.ylim(-1,1.5)
         plt.show()
```



As $\lambda \to \infty$, the weights become smaller, eventually reaching 0. At this point the prediction function is just 0 for all x.

```
In [ ]: def sigmoid_kernel(X1, X2, eta, nu):
            return np.tanh(eta*X1@X2.T+nu)

In [30]: 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, eta=1, nu=1)
                 self.kernel = kernel
                 self.sigma = sigma
                 self.degree = degree
                 self.offset = offset
                 self.l2reg = l2reg
                 self.eta = eta
                 self.nu = nu

             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=se
                 elif self.kernel == 'sigmoid':
                     self.k = functools.partial(sigmoid_kernel, eta=self.eta, nu=self.nu)
                 else:
                     raise ValueError('Unrecognized kernel type requested.')

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

                 return self

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

                 return(self.kernel_machine_.predict(X))

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

In [6]: from sklearn.model_selection import GridSearchCV,PredefinedSplit
```

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

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

In [55]: param_grid = [{'kernel': ['RBF'],'sigma':np.linspace(0.01,1,25), 'l2reg': np.logspace(1
                       {'kernel':['polynomial'],'offset':np.linspace(-5,5,25), 'degree':[2,3,4,5
                       {'kernel':['linear'],'l2reg': np.linspace(0.1,10,50)}]
         kernel_ridge_regression_estimator = KernelRidgeRegression()
         grid = GridSearchCV(kernel_ridge_regression_estimator,
                             param_grid,
                             cv = predefined_split,
                             scoring = make_scorer(mean_squared_error,greater_is_better = False)
                             ,n_jobs = -1  #should allow parallelism, but crashes Python on my m
                             ,verbose=0
                            )
         _ = grid.fit(np.vstack((x_train,x_test)),np.vstack((y_train,y_test)))

In [ ]: 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_sigm
                "mean_test_score","mean_train_score"]
        df_toshow = df[cols_to_keep].fillna('-')
        df_toshow = df_toshow.sort_values(by=["mean_test_score"])
```

6.3.5

Perform hyperparameter search and provide a summary of the results for each kernel

Best params for RBF

```
In [57]: df_toshow[df_toshow.param_kernel=='RBF'].head()
```

```
Out[57]:    param_degree param_kernel  param_l2reg param_offset param_sigma  \
        1             -          RBF     1.000023            -     0.05125
        26            -          RBF     3.359892            -     0.05125
        2             -          RBF     1.000023            -      0.0925
        4             -          RBF     1.000023            -       0.175
        3             -          RBF     1.000023            -     0.13375

            mean_test_score  mean_train_score
        1          0.015609          0.018215
        26         0.025470          0.031895
        2          0.026406          0.029071
        4          0.028863          0.040367
        3          0.028890          0.034944
```

Best params for Polynomial Kernel

```
In [58]: df_toshow[df_toshow.param_kernel=='polynomial'].head()
```

```
Out[58]:      param_degree param_kernel  param_l2reg param_offset param_sigma  \
        3042             6   polynomial      0.05125      2.08333           -
        3069             6   polynomial      0.09250      2.91667           -
        3095             6   polynomial      0.13375      3.33333           -
        3121             6   polynomial      0.17500         3.75           -
        3096             6   polynomial      0.13375         3.75           -

              mean_test_score  mean_train_score
        3042         0.032453          0.049113
        3069         0.032517          0.048018
        3095         0.032600          0.048699
        3121         0.032672          0.048877
        3096         0.032692          0.046910
```

Best params for Linear

```
In [59]: df_toshow[df_toshow.param_kernel=='linear'].head()
```

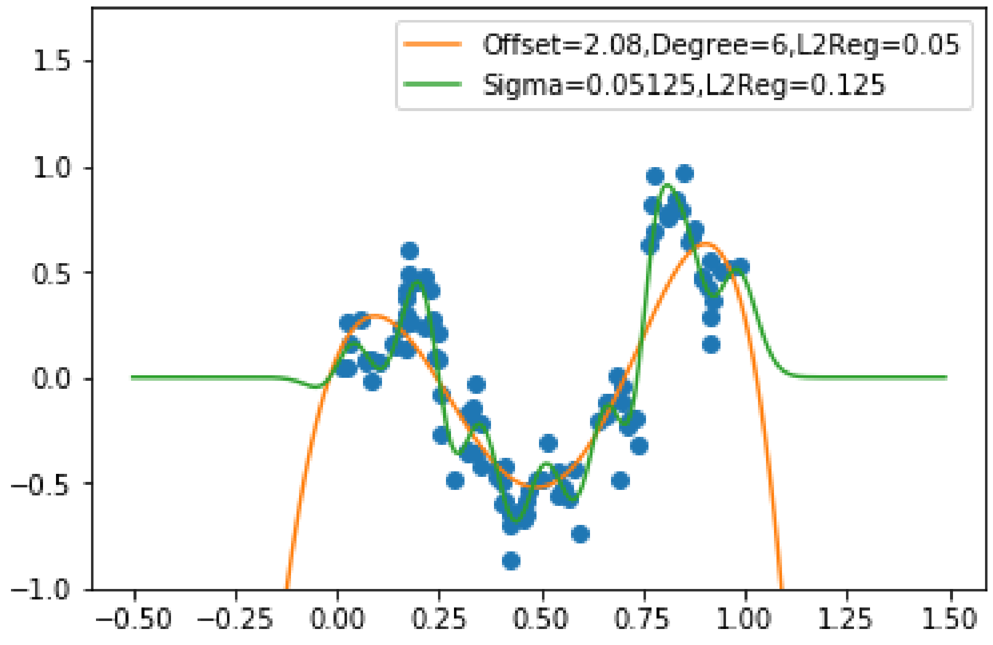
```
Out[59]:      param_degree param_kernel  param_l2reg param_offset param_sigma  \
        3644             -       linear     3.938776            -           -
        3643             -       linear     3.736735            -           -
        3645             -       linear     4.140816            -           -
        3642             -       linear     3.534694            -           -
```

```
3646            -       linear     4.342857             -            -

      mean_test_score  mean_train_score
3644          0.16451          0.206561
3643          0.16451          0.206556
3645          0.16451          0.206567
3642          0.16451          0.206551
3646          0.16451          0.206572
```

6.3.6

Plot the best fitting prediction functions using the polynomial and RBF kernels.

```
In [60]: ## 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= 2.08
         degree = 6
         l2reg = 0.05
         k = functools.partial(polynomial_kernel, offset=offset, degree=degree)
         f = train_kernel_ridge_regression(x_train, y_train, k, l2reg=l2reg)
         label = "Offset="+str(offset)+",Degree="+str(degree)+",L2Reg="+str(l2reg)
         plt.plot(xpts, f.predict(xpts), label=label)
         #Plot best RBF fit
         sigma = 0.05125
         l2reg= 0.125000
         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 RBF kernel fits the data much better. Another thing to note is that outside the domain of x values that the model saw, it predicts approximately 0 while the polynomial kernel quickly goes to $-\infty$ on both ends. This means that most likely, the RBF would perform better with predicting new data that fell outside the range of observed values.

6.3.7

The Bayes prediction function is

$$f^* = E(Y|X) = f(x)$$

The risk of this function is

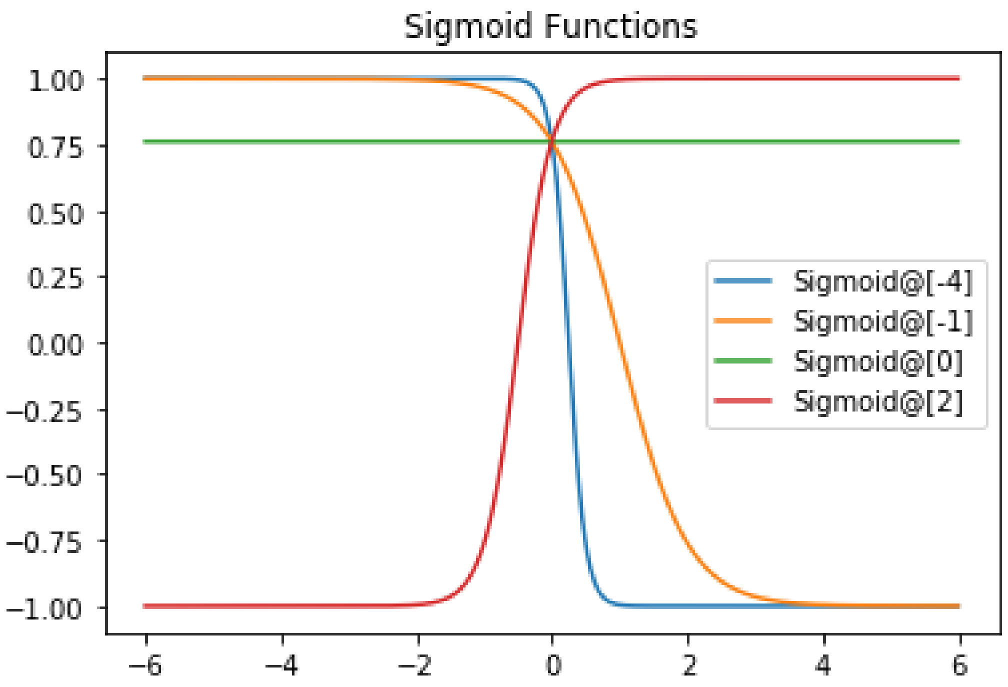
$$\begin{aligned} R(f^*) &= E\big[l(f^*(x), y)\big] \\ &= E\big[(f^*(x) - f(x) - \epsilon)^2\big] \\ &= E\big[(f(x) - f(x) - \epsilon)^2\big] \\ &= E\big[\epsilon^2\big] \\ &= Var(\epsilon) \\ &= 0.1^2 \end{aligned}$$

6.3.8

Attempt to improve performance by using different kernel functions.

```
In [61]: def sigmoid_kernel(X1, X2, eta, nu):
             return np.tanh(eta*X1@X2.T+nu)
```

```
In [62]: # 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)

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



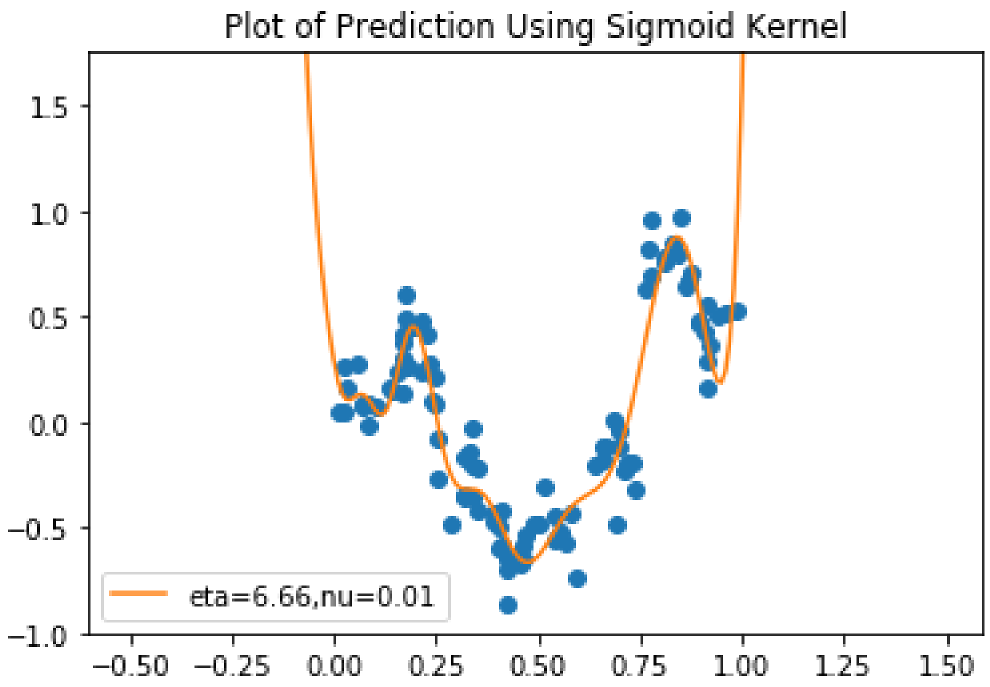
```
In [63]: param_grid = {'eta': np.linspace(0, 10, 10),
                       'nu': np.logspace(-2, 0, 10),
                      'l2reg': np.logspace(-7,-5, 15),
                       'kernel': ['sigmoid']}
```

```
        kernel_ridge_regression_estimator = KernelRidgeRegression()
        grid = GridSearchCV(kernel_ridge_regression_estimator,
                            param_grid,
                            cv = predefined_split,
                            scoring = make_scorer(mean_squared_error,
                                                  greater_is_better = False)
                            ,n_jobs = -1
                            ,verbose=0
                           )
        _ = grid.fit(np.vstack((x_train,x_test)),np.vstack((y_train,y_test)))
```

```
In [64]: df2 = 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"
         df2['mean_test_score'] = -df2['mean_test_score']
         df2['mean_train_score'] = -df2['mean_train_score']
         cols_to_keep = ["param_eta", "param_nu","mean_test_score",
                         "mean_train_score", "param_l2reg"]
         df2 = df2[cols_to_keep]
         df2 = df2.sort_values(by=["mean_test_score"])
         df2.head()
```

```
Out[64]:      param_eta   param_nu  mean_test_score  mean_train_score param_l2reg
         900    6.66667       0.01         0.020141          0.025288       1e-07
         901    6.66667   0.016681         0.020214          0.025650       1e-07
         902    6.66667  0.0278256         0.020612          0.026430       1e-07
         1354        10  0.0774264         0.021767          0.025143       1e-07
         1362        10  0.0278256         0.021950          0.024905  1.3895e-07
```

```
In [65]: plot_step = .01
         xpts = np.arange(-.5 , 1.5, plot_step).reshape(-1,1)
         plt.plot(x_train,y_train,'o')
         #Plot best sigmoid fit
         eta = 6.66
         nu= 0.01
         l2reg = 1e-7
         k = functools.partial(sigmoid_kernel, eta=eta, nu=nu)
         f = train_kernel_ridge_regression(x_train, y_train, k, l2reg=l2reg)
         label = "eta="+str(eta)+",nu="+str(nu)
         plt.plot(xpts, f.predict(xpts), label=label)
         plt.legend(loc = 'best')
         plt.ylim(-1,1.75)
         plt.title("Plot of Prediction Using Sigmoid Kernel")
         plt.show()
```



This didn't improve the test score over using RBF (RBF=0.015, Sigmoid=0.02). However it still fits the data very well.

6.3.9

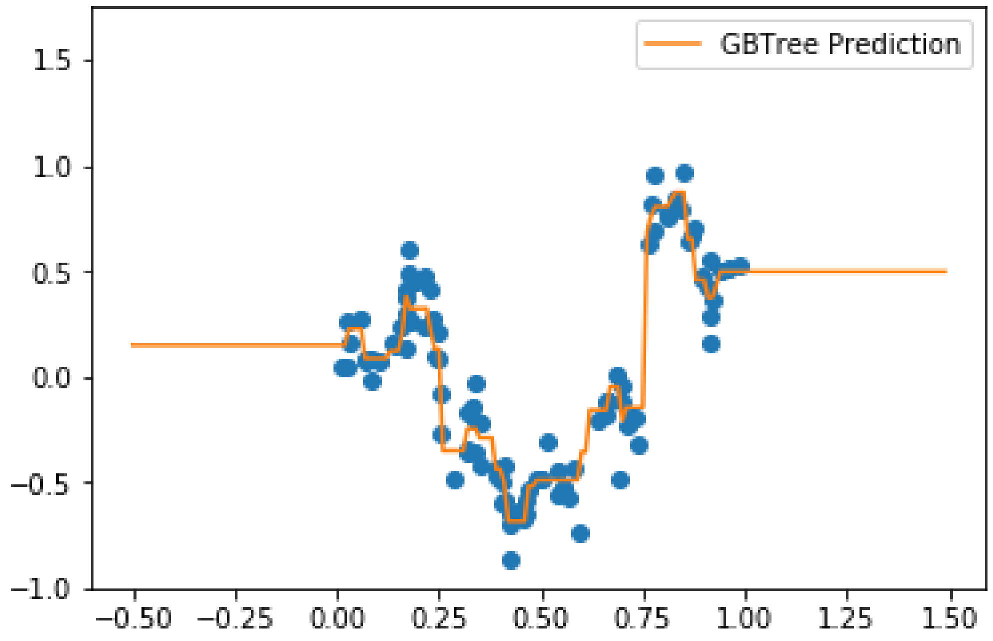
Use any machine learning model to get the best performance you can.

```
In [7]: from sklearn.ensemble import GradientBoostingRegressor
        from sklearn.model_selection import RandomizedSearchCV
        from scipy.stats import randint
        from scipy.stats.distributions import uniform
        param_dist = {'learning_rate': uniform(0.01,1),
                      'n_estimators': randint(1,50),
                      'min_samples_leaf': randint(1,5),
                      'max_depth': randint(2,5)}
        grid = RandomizedSearchCV(GradientBoostingRegressor(),
                                  param_dist,
                                  cv = predefined_split,
                                  scoring = make_scorer(mean_squared_error,
                                                        greater_is_better = False)
                                  ,n_jobs = -1
                                  ,verbose=0
                                  , n_iter=500
                                 )
        _ = grid.fit(np.vstack((x_train,x_test)),np.vstack((y_train,y_test)))
```

```
In [73]: pd.set_option('display.max_rows', 20)
         df2 = 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"
         df2['mean_test_score'] = -df2['mean_test_score']
         df2['mean_train_score'] = -df2['mean_train_score']
         cols_to_keep = ["param_learning_rate", "param_n_estimators",
                         "param_min_samples_leaf", 'param_max_depth',
                         "mean_test_score","mean_train_score"]
         df_toshow2 = df2[cols_to_keep].fillna('-')
         df_toshow2 = df_toshow2.sort_values(by=["mean_test_score"])
         df_toshow2.head()
```

```
Out[73]:      param_learning_rate  param_n_estimators  param_min_samples_leaf  \
         99              0.270287                  25                       3
         308             0.086171                  43                       2
         355             0.211756                  43                       1
         64              0.218163                  24                       2
         9               0.082174                  30                       2

              param_max_depth  mean_test_score  mean_train_score
         99                 4         0.394587          0.032204
         308                4         0.395288          0.054924
         355                4         0.395995          0.008076
         64                 4         0.407881          0.039017
         9                  4         0.408592          0.088934
```

```
In [8]: plot_step = .01
        xpts = np.arange(-.5 , 1.5, plot_step).reshape(-1,1)
        plt.plot(x_train,y_train,'o')
        f = grid.best_estimator_
        plt.plot(xpts, f.predict(xpts), label='GBTree Prediction')
        plt.legend()
        plt.ylim(-1,1.75)
        plt.show()
```



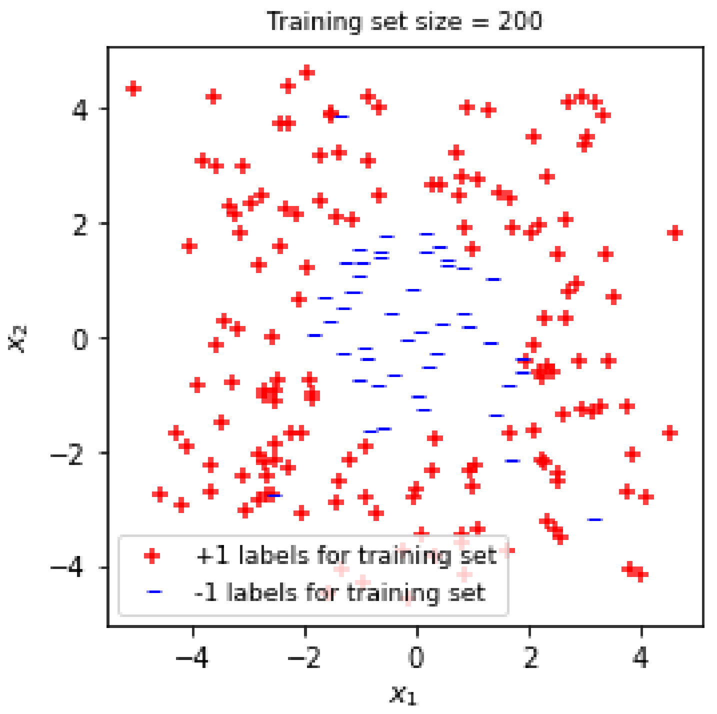
Usign gradient boosted trees resulted in an improved test score over RBF SVF. (RBF=0.15, GBT=0.013)

6.4 Kernalized Support Vector Machines with Kernalized Pegasos

6.4.1

Load the SVM training and 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?

```
In [69]: # Load and plot the SVM data
         #load the training and test sets
         data_train,data_test = np.loadtxt("svm-train.txt"),np.loadtxt("svm-test.txt")
         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 is not linearly or quadratically seperable. However, an RBF kernel could separate it,

```
In [70]: # Code to help plot the decision regions
         # (Note: This ode isn't necessarily entirely appropriate for the questions asked. So th

         sigma=1
         k = functools.partial(RBF_kernel, sigma=sigma)
         f = train_soft_svm(x_train, y_train, k, ...)

         #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)]

         #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_train), fontsize=9
         ax.axis('tight')
         ax.legend(handles=[pluses, minuses], fontsize=9)
         plt.show()


        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-70-2d02ee1ae610> in <module>
          4 sigma=1
```

```
      5 k = functools.partial(RBF_kernel, sigma=sigma)
----> 6 f = train_soft_svm(x_train, y_train, k, ...)
      7
      8 #determine the decision regions for the predictions


    NameError: name 'train_soft_svm' is not defined
```