Machine Learning and Computational Statistics
Homework 4: Kernel Methods and Lagrangian Duality

Yidi Zhang (yz3464)

1 Introduction

2 Positive Semidefinite Matrices

1. Give an example of an orthogonal matrix that is not symmetric. (Hint: You can use a 2×2 matrix with only the entries -1, 0, and 1).
Answer 2.1:

$$\begin{pmatrix} 0.96 & -0.28 \\ 0.28 & 0.96 \end{pmatrix}$$

2. 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?]
Answer 2.2: Since M is positive semidefinite matrix, so it is square matrix. So we can get $Mv = \lambda v$.

$$v^T M v = \lambda v v^T$$

For any $x \in \mathbf{R}^n$,

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

So if v is an eigenvector of M. Since vv^T is a non-negative number, so in order for $v^T M v \geq 0$, λ must be non-negative, that is all eigenvalues of a positive semidefinite matrix M are non-negative.

3. 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.]

Answer 2.3:

If $M = BB^T$, since $BB^T \geq 0$, we can conclude,

For any $x \in \mathbf{R}^n$,

$$x^T(BB^T)x = (x^T B)(xB^T)^T \geq 0.$$

which means $M = BB^T$ is psd.

If M is psd, we can get,

For any $x \in \mathbf{R}^n$,

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

and M is symmetric. According to the Spectral Theorem, M can be written as:

$$M = Q\Sigma Q^T = Q\Sigma^{\frac{1}{2}}(Q\Sigma^{\frac{1}{2}})^T$$

Thus M is psd implies $M = BB^T$ for some B.

3 Positive Definite Matices

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

   Answer 3.1: Since M is symmetric positive definite matrix, so it is square matrix. So we can get $Mv = \lambda v$.

   $$v^T M v = \lambda v v^T$$

   For any $x \in \mathbf{R}^n$ with $x \neq 0$,

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

   So if v is an eigenvector of M. Since vv^T is a positive number, so in order for $v^T M v > 0$, λ must be positive, that is all eigenvalues of a symmetric positive definite matrix M are positive.

2. Let M be a symmetric positve 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 $\Sigma = \text{diag}\,(\sigma_1, \ldots, \sigma_n)$, then $\Sigma^{-1} = \text{diag}\left(\sigma_1^{-1}, \ldots, \sigma_n^{-1}\right)$. Show that the matrix $Q\Sigma^{-1}Q^T$ is the inverse of M.

   Answer 3.2:

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

   So all diagonal entries of Σ are positive.

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 $M + \lambda I$.

Answer 3.3:

Since M is psd, so for any $x \in \mathbf{R}^n$,

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

Thus

$$x^T (M + \lambda I) x = x^T M x + x^T \lambda I x$$

Since $x^T M x \geq 0$ and $x^T \lambda I x > 0$, we can conclude that $x^T (M + \lambda I) x > 0$. So $M + \lambda I$ is symmetric positive definite for any $\lambda > 0$.

Since M is spd, so it can be written as $M = Q \Sigma Q^T$.

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

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 $x^T(M + N)x = x^T Mx + x^T Nx$.)

   Answer 3.4:

$$x^T(M + N)x = x^T Mx + x^T Nx$$

   Since M is psd and N is spd, so we can get for any $x \in \mathbf{R}^n$ with $x \neq 0$, $x^T Mx \geq 0$ and $x^T Nx > 0$.

   Thus $x^T(M + N)x > 0$, and we can conclude that $M + N$ is symmetric positive definite.

4 [Optional] Kernel Matrices

The following problem will gives us some additional insight into what information is encoded in the kernel matrix.

1. [Optional] 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. [Hint: The distance between x and y is given by $d(x, y) = \|x - y\|$, and the norm of a vector x is defined as $\|x\| = \sqrt{\langle x, x \rangle} = \sqrt{x^T x}$.]

5 Kernel Ridge Regression

In lecture, we discussed how to kernelize ridge regression using the representer theorem. Here we pursue a bare-hands approach.

Suppose our input space is $\mathcal{X}$=$\mathbf{R}^d$ and our output space is $\mathcal{Y} = \mathbf{R}$. Let $\mathcal{D} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ be a training set from $\mathcal{X} \times \mathcal{Y}$. We'll use the "design matrix" $X \in \mathbf{R}^{n \times d}$, which has the input vectors as rows:

$$X = \begin{pmatrix} -x_1- \\ \vdots \\ -x_n- \end{pmatrix}.$$

Recall the ridge regression objective function:

$$J(w) = ||Xw - y||^2 + \lambda ||w||^2,$$

for $\lambda > 0$.

1. Show that for w to be a minimizer of $J(w)$, we must have $X^T X w + \lambda I w = X^T y$. Show that the minimizer of $J(w)$ is $w = (X^T X + \lambda I)^{-1} X^T y$. Justify that the matrix $X^T X + \lambda I$ is invertible, for $\lambda > 0$. (The last part should follow easily from the earlier exercises on psd and spd matrices.)

   Answer 5.1: We can get

   $$\nabla J(w) = 2X^T X w - 2X^T y + 2\lambda w$$

   For w to be a minimizer of $J(w)$, we must have $X^T X w + \lambda I w = X^T y$

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

   for any $v \in \mathbf{R}^n$, we can get,

   $$v^T (X^T X + \lambda I) v = v^T X^T X v + v^T \lambda I v$$

   Since we can see $X^T X$ as psd, so $v^T X^T X v \geq 0$. And $v^T \lambda I v > 0$, for $\lambda > 0$.

   Thus

   $$v^T X^T X v + v^T \lambda I v > 0$$

   So $v^T (X^T X + \lambda I) v > 0$, and the matrix $X^T X + \lambda I$ is invertible.

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

Answer 5.2:

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

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

Thus $\alpha = \frac{1}{\lambda}(y - X w)$

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

Answer 5.3:

Since w is in form of $w = X^T\alpha$, so w can be linearly transforms. Thus according to the definition of the 'the span of data', w is 'in the span if the data'

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

Answer 5.4:

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

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

Answer 5.5:

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

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.]

Answer 5.6:

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

Using column vector

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

It can be written as,

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

6 [Optional] Pegasos and SSGD for ℓ_2-regularized ERM[1]

Consider the objective function

$$J(w) = \frac{\lambda}{2}\|w\|_2^2 + \frac{1}{n}\sum_{i=1}^{n} \ell_i(w),$$

where $\ell_i(w)$ represents the loss on the ith training point (x_i, y_i). Suppose $\ell_i(w) : \mathbf{R}^d \to \mathbf{R}$ is a convex function. Let's write

$$J_i(w) = \frac{\lambda}{2}\|w\|_2^2 + \ell_i(w),$$

for the one-point approximation to $J(w)$ using the ith training point. $J_i(w)$ is probably a very poor approximation of $J(w)$. However, if we choose i uniformly at random from $1, \ldots, n$, then we do have $\mathbb{E}J_i(w) = J(w)$. We'll now show that subgradients of $J_i(w)$ are unbiased estimators of some subgradient of $J(w)$, which is our justification for using SSGD methods.

In the problems below, you may use the following facts about subdifferentials without proof (as in Homework #3): 1) If $f_1, \ldots, f_m : \mathbf{R}^d \to \mathbf{R}$ are convex functions and $f = f_1 + \cdots + f_m$, then $\partial f(x) = \partial f_1(x) + \cdots + \partial f_m(x)$ **[additivity]**. 2) For $\alpha \geq 0$, $\partial(\alpha f)(x) = \alpha \partial f(x)$ **[positive homogeneity]**.

1. [Optional] For each $i = 1, \ldots, n$, 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 $v_i(w)$

2. [Optional] Show that $\mathbb{E}g_i(w) \in \partial J(w)$, where the expectation is over the randomly selected $i \in 1, \ldots, n$. (In words, the expectation of our subgradient of a randomly chosen $J_i(w)$ is in the subdifferential of J.)

3. [Optional] Now suppose we are carrying out SSGD with the Pegasos step-size $\eta^{(t)} = 1/(\lambda t)$, $t = 1, 2, \ldots$. 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

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

[Hint: One approach is proof by induction. First show it's true for $w^{(2)}$. Then assume it's true for $w^{(t)}$ and prove it's true for $w^{(t+1)}$. This will prove that it's true for all $t = 2, 3, \ldots$ by induction.

[1] This problem is based on Shalev-Shwartz and Ben-David's book Understanding Machine Learning: From Theory to Algorithms, Sections 14.5.3, 15.5, and 16.3).

4. [Optional] We can use the previous result to get a nice equivent formulation of Pegasos. Let $\theta^{(t)} = \sum_{\tau=1}^{t-1} v^{(t)}$. Then $w^{(t+1)} = -\frac{1}{\lambda t}\theta^{(t+1)}$ Then Pegasos from Homework #3 is equivalent to Algorithm 1. Similar to the $w = sW$ decomposition from homework #3, this decomposition

Algorithm 1: Pegasos Algorithm Reformulation

input: Training set $(x_1, y_1), \ldots, (x_n, y_n) \in \mathbf{R}^d \times \{-1, 1\}$ and $\lambda > 0$.
$\theta^{(1)} = (0, \ldots, 0) \in \mathbf{R}^d$
$w^{(1)} = (0, \ldots, 0) \in \mathbf{R}^d$
$t = 1$ # step number
repeat
  randomly choose j in $1, \ldots, n$
  if $y_j \left\langle w^{(t)}, x_j \right\rangle < 1$
    $\theta^{(t+1)} = \theta^{(t)} + y_j x_j$
  else
    $\theta^{(t+1)} = \theta^{(t)}$
  endif
  $w^{(t+1)} = -\frac{1}{\lambda t}\theta^{(t+1)}$ # need not be explicitly computed
  $t = t + 1$
until bored
return $w^{(t)} = -\frac{1}{\lambda(t-1)}\theta^{(t)}$

gives the opportunity for significant speedup. Explain how Algorithm 1 can be implemented so that, if x_j has s nonzero entries, then we only need to do $O(s)$ accesses in every pass through the loop.

7 Kernelized Pegasos

Recall the SVM objective function

$$\min_{w \in \mathbf{R}^n} \frac{\lambda}{2}\|w\|^2 + \frac{1}{m}\sum_{i=1}^{m} \max\left\{0, 1 - y_i w^T x_i\right\}$$

and the Pegasos algorithm on the training set $(x_1, y_1), \ldots, (x_n, y_n) \in \mathbf{R}^d \times \{-1, 1\}$ (Algorithm 3).

Algorithm 2: Pegasos Algorithm

input: Training set $(x_1, y_1), \ldots, (x_n, y_n) \in \mathbf{R}^d \times \{-1, 1\}$ and $\lambda > 0$.
$w^{(1)} = (0, \ldots, 0) \in \mathbf{R}^d$
$t = 0$ # step number
repeat
  $t = t + 1$
  $\eta^{(t)} = 1/(t\lambda)$ # step multiplier
  randomly choose j in $1, \ldots, n$
  if $y_j \left\langle w^{(t)}, x_j \right\rangle < 1$
    $w^{(t+1)} = (1 - \eta^{(t)}\lambda) w^{(t)} + \eta_t y_j x_j$
  else
    $w^{(t+1)} = (1 - \eta^{(t)}\lambda) w^{(t)}$
until bored
return $w^{(t)}$

Note that in every step of Pegasos, we rescale $w^{(t)}$ by $\left(1 - \eta^{(t)}\lambda\right) = \left(1 - \frac{1}{t}\right) \in (0, 1)$. This "shrinks" the entries of $w^{(t)}$ towards 0, and it's due to the regularization term $\frac{\lambda}{2}\|w\|_2^2$ in the SVM objective function. Also note that if the example in a particular step, say (x_j, y_j), is not classified with the required margin (i.e. if we don't have margin $y_j w_t^T x_j \geq 1$), then we also add a multiple of x_j to $w^{(t)}$ to end up with $w^{(t+1)}$. This part of the adjustment comes from the empirical risk. Since we initialize with $w^{(1)} = 0$, we are guaranteed that we can always write[2]

$$w^{(t)} = \sum_{i=1}^{n} \alpha_i^{(t)} x_i$$

after any number of steps t. When we kernelize Pegasos, we'll be tracking $\alpha^{(t)} = (\alpha_1^{(t)}, \ldots, \alpha_n^{(t)})^T$ directly, rather than w.

[2]Note: This resembles the conclusion of the representer theorem, but it's saying something different. Here, we are saying that the $w^{(t)}$ after every step of the Pegasos algorithm lives in the span of the data. The representer theorem says that a mathematical minimimizer of the SVM objective function (i.e. what the Pegasos algorithm would converge to after infinitely many steps) lies in the span of the data. If, for example, we had chosen an initial $w^{(1)}$ that is NOT in the span of the data, then none of the $w^{(t)}$'s from Pegasos would be in the span of the data. However, we know Pegasos converges to a minimum of the SVM objective. Thus after a very large number of steps, $w^{(t)}$ would be very close to being in the span of the data. It's the gradient of the regularization term that pulls us back towards the span of the data. This is basically because the regularization is driving all components towards 0, while the empirical risk updates are only pushing things away from 0 in directions in the span of the data.

1. 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.

   Answer 7.1:

$$\begin{aligned} y_j \left\langle w^{(t)}, x_j \right\rangle &= y_j \left\langle \sum_{i=1}^{n} \alpha_i^{(t)} x_i, x_j \right\rangle \\ &= y_j \alpha^{(t)} \left\langle \sum_{i=1}^{n} x_i, x_j \right\rangle \\ &= y_j \alpha^{(t)} \sum_{i=1}^{n} \langle x_i, x_j \rangle \\ &= y_j K_{j\cdot} \alpha^{(t)} \end{aligned}$$

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 $w^{(t+1)} = \sum_{i=1}^{n} \alpha_i^{(t+1)} x_i$.

   Answer 7.2: Since points do not have a margin violation, thus,

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

   Since $\eta^{(t)} = 1/(t\lambda)$,

$$\alpha^{(t+1)} = (1 - \frac{1}{t}) \alpha_i^{(t)}$$

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 $y_1, \ldots, y_n \in \{-1, 1\}$

Answer 7.3:

If (x_j, y_j) has a margin violation, then

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

Thus, for $\eta^{(t)} = 1/(t\lambda)$

$$
\alpha^{(t+1)_i} = \begin{cases} (1 - \eta^{(t)}\lambda)\alpha_i^{(t)} & i \neq j \\ (1 - \eta^{(t)}\lambda)\alpha_i^{(t)} + \eta_t y_i & i = j \end{cases}
$$

Algorithm 3: Pegasos Algorithm

input: Training set $(x_1, y_1), \ldots, (x_n, y_n) \in \mathbf{R}^d \times \{-1, 1\}$ and $\lambda > 0$.
$w^{(1)} = (0, \ldots, 0) \in \mathbf{R}^d$
$t = 0$ # step number
repeat
  $t = t + 1$
  $\eta^{(t)} = 1/(t\lambda)$ # step multiplier
  randomly choose j in $1, \ldots, n$
  if $y_j K_j^T \alpha^{(t)} < 1$
    $\alpha^{(t+1)} = (1 - \eta^{(t)}\lambda)\alpha^{(t)}$
    $\alpha_j^{(t+1)} += \eta^{(t)} y_j$
  else
    $\alpha_j^{(t+1)} = (1 - \eta^{(t)}\lambda)\alpha^{(t)} + \eta^{(t)} y_j$
until bored
return $\alpha^{(t+1)}$

4. [Optional] Above we've directly kernelized . While the direct implementation of the original Pegasos required updating all entries of w in every step, a direct kernelization of Algorithm 3, 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 6 above.

8 Kernel Methods: Let's Implement

8.1 One more review of kernelization can't hurt (no problems)

8.2 Kernels and Kernel Machines

There are many different families of kernels. So far we've 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 ML methods later on. For simplicity, and because it is by far the most common situation[3], we will assume that our input space is $\mathcal{X} = \mathbf{R}^d$. This allows us to represent a collection of n inputs in a matrix $X \in \mathbf{R}^{n \times d}$, as usual.

1. Write functions that compute the RBF kernel $k_{\text{RBF}(\sigma)}(x, x') = \exp\left(-\|x - x'\|^2 / \left(2\sigma^2\right)\right)$ and the polynomial kernel $k_{\text{poly}(a,d)}(x, x') = (a + \langle x, x' \rangle)^d$. The linear kernel $k_{\text{linear}}(x, x') = \langle x, x' \rangle$, has been done for you in the support code. Your functions should take as input two matrices $W \in \mathbf{R}^{n_1 \times d}$ and $X \in \mathbf{R}^{n_2 \times d}$ and should return a matrix $M \in \mathbf{R}^{n_1 \times n_2}$ 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). The matrix M could be called the "cross-kernel" matrix, by analogy to the cross-covariance matrix. For the RBF kernel, you may use the scipy function `cdist(X1,X2,'sqeuclidean')` in the package `scipy.spatial.distance` or (with some more work) write it in terms the linear kernel (Bauckhage's article on calculating Euclidean distance matrices may be helpful).

   Answer 8.2.1:

```
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/Gaussian kernel
    Returns:
        matrix of size n1xn2, with exp(-||w_i-x_j||^2/(2 sigma^2)) in position i,j
    """
    k = scipy.spatial.distance.cdist(W, X, 'sqeuclidean')
    k *= -1.0/(2*np.square(sigma))

    return np.exp(k)

def polynomial_kernel(W, X, offset, degree):
    """
    Computes the inhomogeneous polynomial kernel between two sets of vectors
    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 position i,j
    """
    k = linear_kernel(W, X)
```

[3] We are noting this because one interesting aspect of kernel methods is that they can act directly on an arbitrary input space $\mathcal{X}$ (e.g. text files, music files, etc.), so long as you can define a kernel function $k : \mathcal{X} \times \mathcal{X} \to \mathbf{R}$. But we'll not consider that case here.

```
    k = offset + k
    k = np.power(k,degree)
    return k
```

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.

Answer 8.2.2:

```
# compute the kernel matrix
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)
print(y)
```

OUTPUT

```
[[ 24.    23.96  23.92 ..., -23.88 -23.92 -23.96]
 [  6.     5.99   5.98 ...,  -5.97  -5.98  -5.99]
 [ -0.    -0.    -0.   ...,   0.     0.     0.  ]
 [-12.   -11.98 -11.96 ...,  11.94  11.96  11.98]]
```

3. Suppose we have the data set $\mathcal{D} = \{(-4, 2), (-1, 0), (0, 3), (2, 5)\}$. 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.

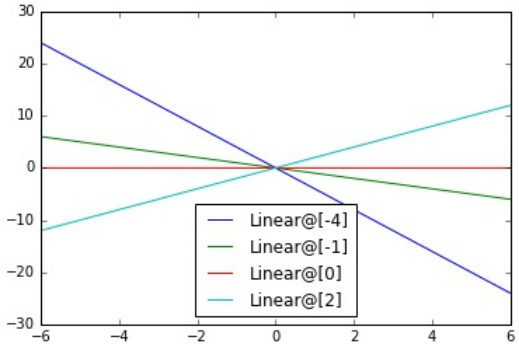
Answer 8.2.3:

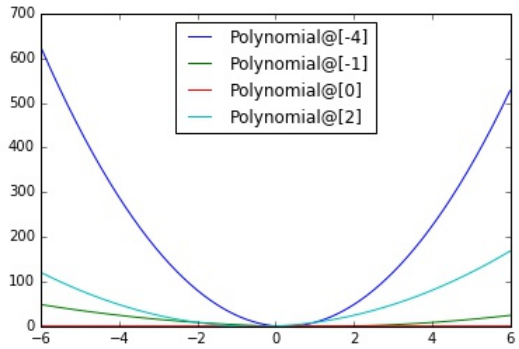
```
# PLot kernel machine functions

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

# RBF kernel
y = RBF_kernel(prototypes, xpts,sigma=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()

# Plot kernel machine functions
plot_step = .01
xpts = np.arange(-6, 6, plot_step).reshape(-1,1)
prototypes = np.array([-4,-1,0,2]).reshape(-1,1)
offset = 1
degree = 2

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

# 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.show()
```

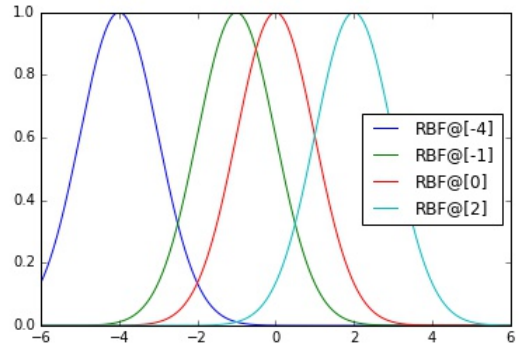
(a) Plot the set of functions $x \mapsto k_{\text{linear}}(x_0, x)$ for $x_0 \in \mathcal{D}_X$ and for $x \in [-6, 6]$.

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

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







4. 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. This is a special case of what is sometimes called a **kernel machine**, which is a function of the form $f(x) = \sum_{i=1}^{r} \alpha_i k(\mu_i, x)$, where $\mu_1, \ldots, \mu_r \in \mathcal{X}$ are called **prototypes** or **centroids** (Murphy's book Section 14.3.1.).

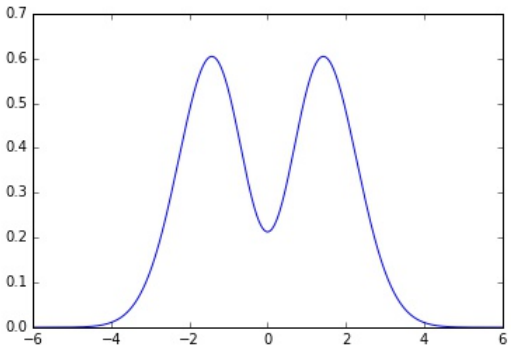
In the special case that the kernel is an RBF kernel, we get what's called an **RBF Network** (proposed by Broomhead and Lowe in 1988). We can see that the prediction functions we get from our kernel methods will be kernel machines in which every input point $x_1, \ldots, x_n$ serves as a prototype point. Complete the `predict` function of the class `Kernel_Machine` in the skeleton code. Construct a `Kernel_Machine` object with the RBF kernel (sigma=1), with prototype points at $-1, 0, 1$ and corresponding weights $1, -1, 1$. Plot the resulting function.

Note: For this 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.

Answer 8.2.4:

```
class Kernel_Machine(object):
    def __init__(self, kernel, prototype_points, weights):
        """
        Args:
            kernel(W,X) - a function return the cross-kernel matrix between rows of W and
     rows of X for kernel k
            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):#xpts
        """
        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, jth
     entry of return vector is
                Sum_{i=1}^R w_i k(x_j, mu_i)
        """
        y = self.kernel(self.prototype_points, X)
        y = np.dot(np.transpose(y), self.weights)
        return y

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

k = functools.partial(RBF_kernel, sigma=sigma)
f = Kernel_Machine(k, prototypes, W)
y = f.predict(xpts)
```

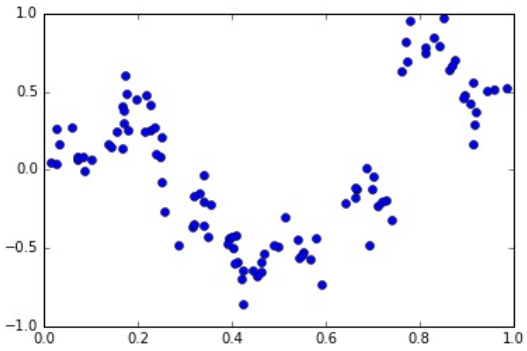
```
plt.plot(xpts, y)
#plt.legend(loc = 'best')
plt.show()
```



8.3 Kernel Ridge Regression

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

   Answer 8.3.1:



2. In a previous problem, we showed that in kernelized ridge regression, the final prediction function is $f(x) = \sum_{i=1}^{n} \alpha_i k(x_i, x)$, where $\alpha = (\lambda I + K)^{-1} y$ and $K \in \mathbf{R}^{n \times n}$ is the Gram matrix of the training data: $K_{ij} = k(x_i, x_j)$, for $x_1, \ldots, x_n$. In terms of kernel machines, α_i is the weight on the kernel function evaluated at the prototype point 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.

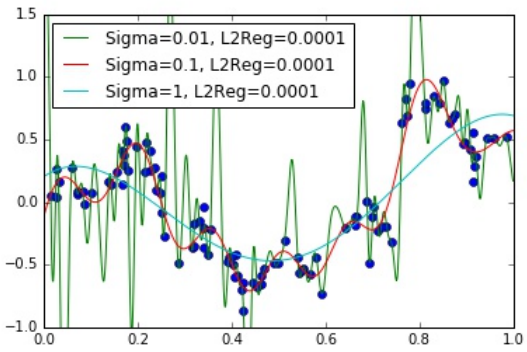
Answer 8.3.2:

```
def train_kernel_ridge_regression(X, y, kernel, l2reg):
    '''
    X is prototype_points, which is the training set.
    y actual value of y
    kernel is kernel machine types
    ïŒ ïŒ ïŒ ïŒ ïŒ ïŒ ïŒ ïŒ ïŒ ïŒ ïŒ ïŒ ïŒ ïŒ ïŒ ïŒ ïŒ ïŒ ïŒ ïŒ ïŒ ïŒ ïŒ ïŒ ïŒ ïŒ ïŒ
    alpha is the weight matrix
    '''
    N = X.shape[0]
    reg = np.identity(N) * l2reg
    k = kernel(X,X)
    alpha = reg + k
    alpha = np.linalg.inv(alpha)
    alpha = np.dot(alpha, y).reshape(-1,1)
    return Kernel_Machine(kernel, X, alpha)
```

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. With values of sigma do you think would be more likely to over fit, and which less?

Answer 8.3.3:

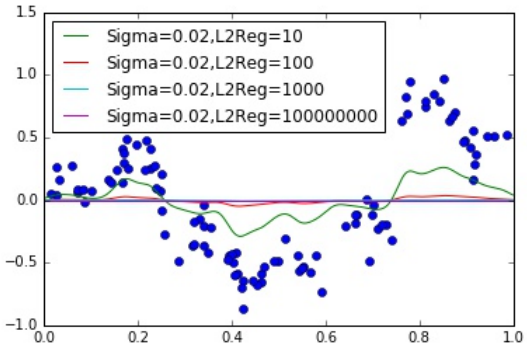
```
plot_step = .001
xpts = np.arange(0 , 1, plot_step).reshape(-1,1)
plt.plot(x_train,y_train,'o')
l2reg = 0.0001
for sigma in [.01,.1,1]:
    k = functools.partial(RBF_kernel, sigma=sigma)
    f = train_kernel_ridge_regression(x_train, y_train, k, l2reg=l2reg)
    label = "Sigma="+str(sigma)+", L2Reg="+str(l2reg)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



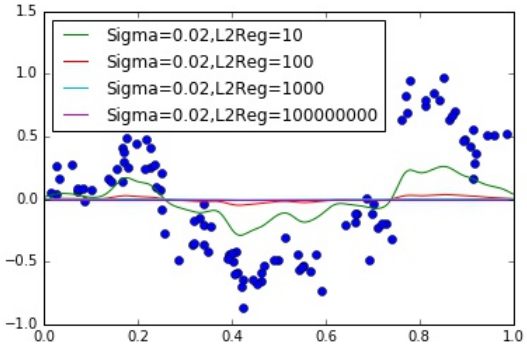
With value of σ 0.01, the model are more likely to be overfitting. With value of σ 1, the model are more likely to be underfitting.

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 $\lambda \to \infty$? **Answer 8.3.4:**

```
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()

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 [10,100,1000,100000000]:
    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 lambda approaching infinity, the line approch horizontal stright line
```



As $\lambda \to \infty$, the plot of prediction function tend to be a straight horizontal line with y value 0.



5. 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 us 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.

Answer 8.3.5:

	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_sqloss	mean_train_sqloss
5998	-	RBF	0.000983	-	0.1	1.489041e-02	0.014938
5995	-	RBF	0.000990	-	0.1	1.489184e-02	0.014945
5992	-	RBF	0.000997	-	0.1	1.489327e-02	0.014951
5989	-	RBF	0.001004	-	0.1	1.489471e-02	0.014957
5986	-	RBF	0.001011	-	0.1	1.489617e-02	0.014964
5983	-	RBF	0.001018	-	0.1	1.489763e-02	0.014970
5980	-	RBF	0.001025	-	0.1	1.489911e-02	0.014977
5977	-	RBF	0.001032	-	0.1	1.490060e-02	0.014983
5974	-	RBF	0.001039	-	0.1	1.490209e-02	0.014990
5971	-	RBF	0.001047	-	0.1	1.490361e-02	0.014997
...	...	...	...	...	...	...	...
6958	15	polynomial	0.010000	8	-	1.030903e+03	109.877197
7199	19	polynomial	0.010000	9	-	2.084279e+03	316.984336
7179	19	polynomial	0.100000	9	-	2.084279e+03	316.984336
7159	19	polynomial	10.000000	9	-	2.084279e+03	316.984336
6688	11	polynomial	0.100000	-2	-	2.210860e+03	1218.082351
6881	14	polynomial	0.010000	-9	-	5.362450e+03	1767.032426
7042	17	polynomial	0.100000	-8	-	9.521349e+03	8578.850060
7184	19	polynomial	0.010000	-6	-	9.905335e+03	5980.056517
7062	17	polynomial	0.010000	-8	-	1.087366e+04	10468.480197
6982	16	polynomial	0.100000	-8	-	1.295554e+06	622848.591099

7206 rows × 7 columns

To get the best hyperparameter, the parameter range is

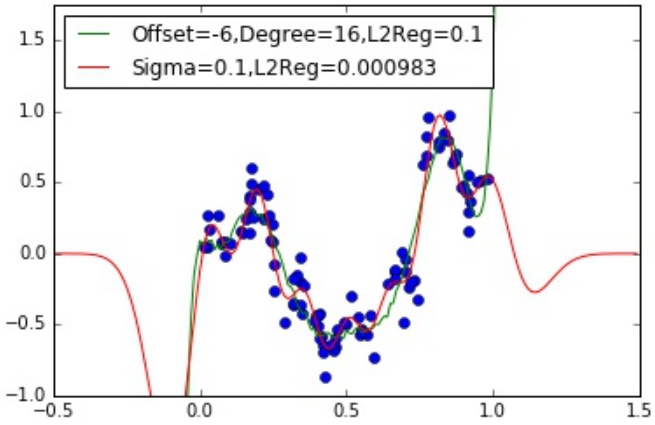
```
{'kernel': ['RBF'],'sigma':[.001,0.1,100], 'l2reg': np.exp2(-np.arange(-10,10,0.01))
    },
              {'kernel':['polynomial'],'offset':range(-10,10), 'degree':range(20),'
    l2reg':[10, .1, .01] },
              {'kernel':['linear'],'l2reg': [100,10,1,.01,0.001,0.0001]}]
```

6. 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.

Answer 8.3.6:

After ranking, we can get the best prediction function. For polynomial function, when $offset = -6$, $degree = 16$, and $l2reg = 0.1$, it gets the best polynmial function. For RBF kernel, $sigma = 0.1$ and $l2reg = 0.000983$ can fit the best function.

```
## 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= -6
degree = 16
l2reg = 0.1
k = functools.partial(polynomial_kernel, offset=offset, degree=degree)
f = train_kernel_ridge_regression(x_train, y_train, k, l2reg=l2reg)
label = "Offset="+str(offset)+",Degree="+str(degree)+",L2Reg="+str(l2reg)
plt.plot(xpts, f.predict(xpts), label=label)
#Plot best RBF fit
sigma = 0.1
l2reg= 0.000983
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()
```



In my point of view, they both can fit the model perfectly. But due to the ack of data when x bigger than 1, the RBF kernel and Polynomial kernel show a obvious difference.

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

Answer 8.3.7:

Risk of $f : \mathbf{R} \to \mathbf{R}$ is:

$$R(f) = E\ell(f(x), y)$$

So the Bayes decision function $f^* : \mathbf{R} \to A$ is:

$$f^* = \arg\min_f R(f) = \arg\min_f E\ell(f(x), y) = \arg\min_f E\left((f(x) + \varepsilon - y)^2\right)$$

And the corresponding Bayes risk is:

$$R(f^*) = E\left((f(x) + \varepsilon - y)^2\right)$$

According to 3.1 in assignment1, we can conclude that,

$$f^* = E(\hat{y}) = E(f(x) + \varepsilon | x) = E(f(x)|x)$$

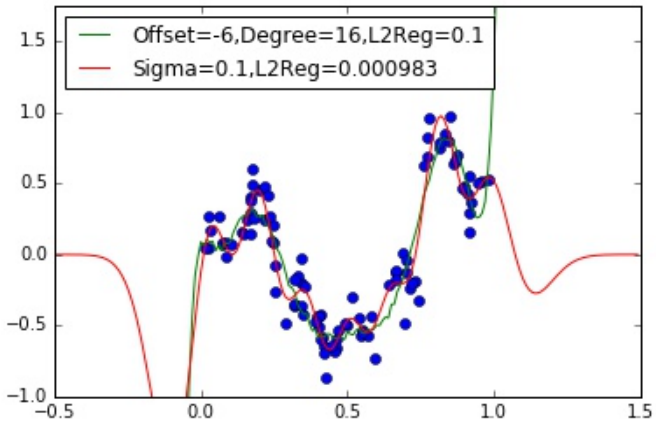
And Bayes Risk is

$$R(f^*) = E(\varepsilon^2) = \mathrm{Var}(\varepsilon) = 0.1^2$$

8. [Optional] Attempt to improve performance by using different kernel functions. Chapter 4 from Rasmussen and Williams' book *Gaussian Processes for Machine Learning* describes many kernel functions, though they are called **covariance functions** in that book (but they have exactly the same definition). Note that you may also create a kernel function by first explicitly creating feature vectors, if you are so inspire.

9. [Optional] Use any machine learning model you like to get the best performance you can.

```
## 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= -6
degree = 16
l2reg = 0.1
k = functools.partial(polynomial_kernel, offset=offset, degree=degree)
f = train_kernel_ridge_regression(x_train, y_train, k, l2reg=l2reg)
label = "Offset="+str(offset)+",Degree="+str(degree)+",L2Reg="+str(l2reg)
plt.plot(xpts, f.predict(xpts), label=label)
#Plot best RBF fit
sigma = 0.1
l2reg= 0.000983
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()
```

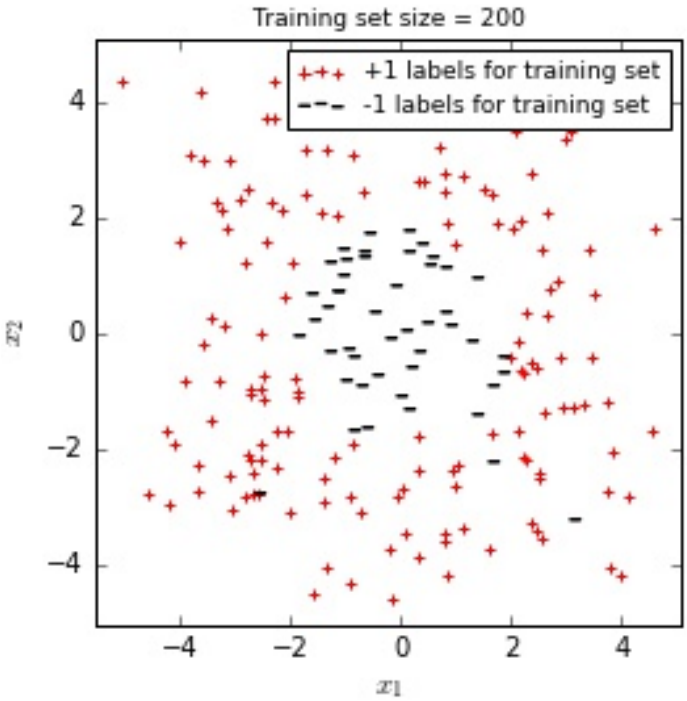


8.4 [Optional] Kernelized Support Vector Machines with Kernelized Pegasos

1. Load the SVM training and test data from the zip file. Plot the training data using the code supplied. Is the data linearly separable? Quadratically seperable? What if we used some RBF kernel?

```
# 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.savefig('/Users/twff/Downloads/machine_learning/hw/hw4-kernels/svm.jpg')
```



It is not linearly separable. It is not quadratically seperable. RBF working really bad on this data set.

2. 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.

```
# 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.)

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()
```

3. 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.

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

9 Representer Theorem [Optional]

Recall the following theorem from lecture:

Theorem (Representer Theorem)**.** *Let*

$$J(w) = R\left(\|w\|\right) + L\left(\langle w, \psi(x_1)\rangle, \ldots, \langle w, \psi(x_n)\rangle\right),$$

where $R : \mathbf{R}^{\geq 0} \to \mathbf{R}$ *is nondecreasing (the* ***regularization*** *term) and* $L : \mathbf{R}^n \to \mathbf{R}$ *is arbitrary (the* ***loss*** *term). If* $J(w)$ *has a minimizer, then it has a minimizer of the form*

$$w^* = \sum_{i=1}^{n} \alpha_i \psi(x_i).$$

Furthermore, if R *is strictly increasing, then all minimizers have this form.*

Note: There is nothing in this theorem that guarantees $J(w)$ has a minimizer at all. If there is no minimizer, then this theorem does not tell us anything.

In this problem, we will prove the part of the Representer theorem for the case that R is strictly increasing.

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 $m_0 = x$. (Hint: Use the postive-definiteness of the inner product: $\langle x, x\rangle \geq 0$ and $\langle x, x\rangle = 0 \iff x = 0$, and the fact that we're using the norm derived from such an inner product.)

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. (Hint: Consider separately the cases that $\|w\| < \|w^*\|$ and the case $\|w\| = \|w^*\|$.)

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 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.

10 Ivanov and Tikhonov Regularization [Optional]

In lecture there was a claim that the Ivanov and Tikhonov forms of ridge and lasso regression are equivalent. We will now prove a more general result.

10.1 Tikhonov optimal implies Ivanov optimal

Let $\phi : \mathcal{F} \to \mathbf{R}$ be any performance measure of $f \in \mathcal{F}$, and let $\Omega : \mathcal{F} \to \mathbf{R}$ be any complexity measure. For example, for ridge regression over the linear hypothesis space $\mathcal{F} = \left\{ f_w(x) = w^T x \mid w \in \mathbf{R}^d \right\}$, we would have $\phi(f_w) = \frac{1}{n} \sum_{i=1}^{n} \left(w^T x_i - y_i \right)^2$ and $\Omega(f_w) = w^T w$.

1. Suppose that for some $\lambda > 0$ we have the Tikhonov regularization solution

$$f_* = \arg\min_{f \in \mathcal{F}} \left[\phi(f) + \lambda \Omega(f) \right]. \tag{1}$$

   Show that f_* is also an Ivanov solution. That is, $\exists r > 0$ such that

$$f_* = \arg\min_{f \in \mathcal{F}} \phi(f) \text{ subject to } \Omega(f) \leq r. \tag{2}$$

   (Hint: Start by figuring out what r should be. If you're stuck on this, ask for help. Then one approach is proof by contradiction: suppose f^* is not the optimum in (2) and show that contradicts the fact that f^* solves (1).)

10.2 Ivanov optimal implies Tikhonov optimal (when we have Strong Duality)

For the converse, we will restrict our hypothesis space to a parametric set. That is,

$$\mathcal{F} = \left\{ f_w(x) : \mathcal{X} \to \mathbf{R} \mid w \in \mathbf{R}^d \right\}.$$

So we will now write ϕ and Ω as functions of $w \in \mathbf{R}^d$.

Let w^* be a solution to the following Ivanov optimization problem:

$$\begin{array}{ll} \text{minimize} & \phi(w) \\ \text{subject to} & \Omega(w) \leq r. \end{array}$$

Assume that strong duality holds for this optimization problem and that the dual solution is attained. Then we will show that there exists a $\lambda \geq 0$ such that $w_* = \arg\min_{w \in \mathbf{R}^d} \left[\phi(w) + \lambda \Omega(w) \right]$.

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

2. Write the dual optimization problem in terms of the dual objective function $g(\lambda)$, and give an expression for $g(\lambda)$. [Writing $g(\lambda)$ as an optimization problem is expected - don't try to solve it.]

3. We assumed that the dual solution is attained, so let $\lambda^* = \arg\max_{\lambda \geq 0} g(\lambda)$. 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^*. [Hint: You can use the same approach we used when we derived that strong duality implies complementary slackness[4].] **Conclude the proof** by showing that for the choice of $\lambda = \lambda^*$, we have $w_* = \arg\min_{w \in \mathbf{R}^d} \left[\phi(w) + \lambda\Omega(w)\right]$.

4. The conclusion of the previous problem allows $\lambda = 0$, which means we're not actually regularizing at all. To ensure we get a proper Ivanov regularization problem, we need an additional assumption. The one below is taken from [**?**]:

$$\inf_{w \in \mathbf{R}^d} \phi(w) < \inf_{\substack{w \in \mathbf{R}^d \\ \Omega(w) \leq r}} \phi(w)$$

Note that this is a rather intuitive condition: it is simply saying that we can fit the training data better [strictly better] if we don't use any regularization. With this additional condition, show that $w_* = \arg\min_{w \in \mathbf{R}^d} \left[\phi(w) + \lambda\Omega(w)\right]$ for some $\lambda > 0$.

10.3 Ivanov implies Tikhonov for Ridge Regression.

To show that Ivanov implies Tikhonov for the ridge regression problem (square loss with ℓ_2 regularization), we need to demonstrate strong duality and that the dual optimum is attained. Both of these things are implied by Slater's constraint qualifications.

1. Show that the Ivanov form of ridge regression is a convex optimization problem with a strictly feasible point.

11 Novelty Detection [Optional]

(Problem derived from Michael Jordan's Stat 241b Problem Set #2, Spring 2004)

A novelty detection algorithm can be based on an algorithm that finds the smallest possible sphere containing the data in feature space.

1. Let $\phi : \mathcal{X} \to \mathcal{H}$ be our feature map, mapping elements of the input space into our "feature space" $\mathcal{H}$, which is a Hilbert space (and thus has an inner product). Formulate the novelty detection algorithm described above as an optimization problem.

2. Give the Lagrangian for this problem, and write an equivalent, unconstrained "inf sup" version of the optimization problem.

3. Show that we have strong duality and thus we will have an equivalent optimization problem if we swap the inf and the sup. [Hint: Use Slater's qualification conditions.]

[4] See https://davidrosenberg.github.io/mlcourse/Archive/2016/Lectures/3b.convex-optimization.pdf#page=30 slide 30.

4. Solve the inner minimization problem and give the dual optimization problem. [Note: You may find it convenient to define the kernel function $k(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle$ and to write your final problem in terms of the corresponding kernel matrix K to simplify notation.]

5. Write an expression for the optimal sphere in terms of the solution to the dual problem.

6. Write down the complementary slackness conditions for this problem, and characterize the points that are the "support vectors".

7. Briefly explain how you would apply this algorithm in practice to detect "novel" instances.

8. [More Optional] Redo this problem allowing some of the data to lie outside of the sphere, where the number of points outside the sphere can be increased or decreased by adjusting a parameter. (Hint: Use slack variables).

6.1

$$g_i(w) = J_i'(w)$$
$$= \lambda w + \nu_i(w)$$

6.2

$$E(g_i(w)) = \frac{1}{n}\sum_{i=1}^{n}(\lambda w + \nu_i(w)) = \lambda w + \frac{1}{n}\sum_{i=1}^{n}\nu_i(w)$$

$\because E(g_i(w)) \in \partial J(w)$

6.3.

$$w^{(t+1)} = w^{(t)} - \frac{1}{\lambda t}(\lambda w^{(t)} + \nu^{(t)})$$
$$= w^{(t)} - \frac{1}{t}w^{(t)} - \frac{1}{\lambda t}\nu^{(t)}$$
$$= \frac{t-1}{t}w^{(t)} - \frac{1}{\lambda t}\nu^{(t-1)}$$

$$\therefore w^{(t)} = \left(\frac{t-2}{t-1}\right)w^{(t-1)} - \frac{1}{\lambda(t-1)}\nu^{(t-1)}$$

Thus we can get.

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

6.4

Since. $\theta^{(t)}$ can be unchanged if. it doesn't have a. margin violation. Thus. we can conclude that. for. each x_j, it only updates. $O(s)$ times. through the loop.

9.1

$$\|x\|^2 = \|m_0\|^2 + \|x - m_0\|^2$$

$$\|x - m_0\|^2 = \|x\|^2 - \|m_0\|^2$$

if $\|x\| = \|m_0\|$

$$\|x - m_0\|^2 = 0.$$

Since.

$$\|x - m_0\|^2 = \langle x - m_0, x - m_0 \rangle$$

So according to the definition of inner product.

we can get.

$$\|x - m_0\|^2 = \langle x - m_0, x - m_0 \rangle = 0 \quad \text{if and only if.}$$

$$x - m_0 = 0, \quad x = m_0$$

9.2. Assume w^* is another minimizer.

if. $\|w^*\| > \|w\|$

$R(\|w^*\|) > R(\|w\|) \Rightarrow J(w^*) > J(w)$

which. contradicts to the. assumption of w^*

if $\|w^*\| = \|w\|$, we can get.

$\|r\|^2 = 0$, $w^* = \sum_{i=1}^{n} \alpha_i \varphi(x_i)$

Thus R is strictly increasing.