Homework 4: Kernel Methods

Rui Jiang NetID: rj1407

1 Introduction

2 [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 $\|x\| = \sqrt{\langle x, x\rangle} = \sqrt{x^T x}$.]

   Solution:

$$K = XX^T = \begin{bmatrix} \text{—} & x_1 & \text{—} \\ \text{—} & x_2 & \text{—} \\ & \vdots & \\ \text{—} & x_m & \text{—} \end{bmatrix} \begin{bmatrix} | & & | \\ x_1 & \ldots & x_m \\ | & & | \end{bmatrix}$$

$$\therefore K_{ij} = x_i x_j$$

When $i = j$, $K_{ij} = (x_i)^2 = \|x_i\|^2$
$\because d(x, y)^2 = \|x - y\|^2 = x^2 + y^2 - 2xy$
When $i \neq j$, $K_{ij} = x_i x_j = \frac{\|x_i\|^2 + \|x_j\|^2 - d(x_i, x_j)^2}{2}$
$\therefore$ 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

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 $X^TXw + \lambda Iw = X^Ty$. 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 $\lambda > 0$. (The last part should follow easily from the exercises on psd and spd matrices in the Appendix.)

   Solution

   $$\begin{aligned} J(w) &= (Xw - y)^T(Xw - y) + \lambda w^T w \\ &= 2(X^TXw - X^Ty) + 2\lambda Iw \end{aligned}$$

   when w to be a minimizer of J(w), J(w) = 0, then:

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

   Then,

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

   Next, to justify that the matrix $X^TX + \lambda I$ is invertible, for $\lambda > 0$:
   For any $v \in \mathbf{R}^n$,

   $$v^T(X^TX)v = v^TX^TXv = (Xv)^T(Xv) \geq 0$$

   $\therefore X^TX$ is positive semidefinite(psd)
   For any $v \in \mathbf{R}^n, \lambda \geq 0$,

   $$v^T(\lambda I)v = \lambda v^TIv \geq 0$$

   $\therefore \lambda I$ is positive definite(pd)
   $\because X^TX, \lambda I$ are both symmetric matrices, and the former is psd, the latter is pd, according to the definitions, we have: For any $v \in \mathbf{R}^n$,

   $$v^T(X^TX + \lambda I)v = (Xv)^T(Xv) + \lambda v^TIv > 0$$

   $\therefore X^TX + \lambda I$ is pd
   If vector v is an eigenvector of matrix $X^TX + \lambda I$, then we have $(X^TX + \lambda I)v = \lambda_1 v, \lambda_1$ is the corresponding eigenvalue.
   $\therefore v^T(X^TX + \lambda I)v = \lambda_1 v^Tv > 0$
   $\therefore \lambda_1 > 0$, which means $X^TX + \lambda I$ has strictly positive eigenvalues.
   so 0 is not an eigenvalue of the matrix. Therefore, the system of equations $(X^TX + \lambda I)x = 0$ has no non-trivial solution, and so $X^TX + \lambda I$ is invertible.

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

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

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

Solution

$$\because \alpha = \tfrac{y - Xw}{\lambda} \text{ is a scalar.}$$
$$\text{Then } w = X^T\alpha = \begin{bmatrix} | & & | \\ x_1 & \dots & x_n \\ | & & | \end{bmatrix} \begin{bmatrix} -\alpha_1- \\ \vdots \\ -\alpha_n- \end{bmatrix}$$
$$= \alpha_1 \begin{bmatrix} | \\ x_1 \\ | \end{bmatrix} + \alpha_2 \begin{bmatrix} | \\ x_2 \\ | \end{bmatrix} + \cdots + \alpha_n \begin{bmatrix} | \\ x_n \\ | \end{bmatrix}$$

Therefore, w is a linear combination of $\{x_1, x_2, ..., x_n\}$, which means w is in the span of 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.)

Solution

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

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.)

Solution
Let $K = XX^T$

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

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.]

Solution

$$\begin{aligned} f(x) = x^T w^* &= x^T X^T \alpha \\ &= \begin{pmatrix} x^T x_1 \\ \vdots \\ x^T x_n \end{pmatrix}^T (\lambda I + XX^T)^{-1} y \\ &= k_x^T (\lambda I + XX^T)^{-1} y \end{aligned}$$

4 [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 $v_i(w)$

Solution:

[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).

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

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.)
Solution:

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

$$\because \frac{1}{n}\sum_{i=1}^{n} v_i(w) \in \partial(\frac{1}{n}\sum_{i=1}^{n} \ell_i(w))$$
$$\therefore \mathbb{E}g_i(w) \in \partial J(w)$$

3. [Optional] 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

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

[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

Solution:
When t=1,
$w^{(2)} = w^{(1)} - \frac{1}{\lambda} g_i(w^{(1)}) = w^{(1)} - \frac{1}{\lambda}(v_i(w^{(1)}) + \lambda w^{(1)}) = -\frac{1}{\lambda} v^{(1)} = w^{(t+1)} = -\frac{1}{\lambda t}\sum_{\tau=1}^{t} v^{(\tau)}$
We assume it's true for $w^{(t)}$, then
$w^{(t+1)} = w^{(t)} - \frac{1}{\lambda t}(v_i w^{(t)} + \lambda w^{(t)}) = w^{(t)}(1 - \frac{1}{t}) - \frac{1}{\lambda t} v^{(t)} = -\frac{1}{\lambda(t-1)}\sum_{\tau=1}^{t-1} v^{(\tau)} \cdot \frac{t-1}{t} - \frac{1}{\lambda t} v^{(t)}$
$= -\frac{1}{\lambda t}(\sum_{\tau=1}^{t-1} v^{(\tau)} + v^{(t)}) = -\frac{1}{\lambda t}\sum_{\tau=1}^{t} v^{(\tau)}$
Q.E.D.

(a) [Optional] We can use the previous result to get a nice equivalent 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 the previous homework is equivalent to Algorithm 1. Similar to the $w = sW$ decomposition from homework #3, this decomposition gives the opportunity for significant speedup. Explain how Algorithm 1 can be implemented so that, if x_j has s nonzero entries, then we

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 $y_j \langle w^{(t)}, x_j \rangle < 1$
    $\theta^{(t+1)} = \theta^{(t)} + y_j x_j$
  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 $w^{(t)} = -\frac{1}{\lambda(t-1)}\theta^{(t)}$

only need to do $O(s)$ memory accesses in every pass through the loop.

Solution:
Instead of updating w , we updateθ
when not satisfying $y_j \langle w^{(t)}, x_j \rangle < 1$, no need to update θ
else, when $x_j[i] = 0, \theta[i]^{(t+1)} = \theta[i]^{(t)}$; when $x_j[i] \neq 0, \theta[i]^{(t+1)} = \theta[i]^{(t)} + y_j x_j[i]$
So only need to touch all the nonzero entries of x_j, then we only need to do $O(s)$ memory accesses in every pass through the loop.

5 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 2).

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

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

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 $w^{(t)}$

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.

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.

   Solution:

$$y_j \left\langle w^{(t)}, x_j \right\rangle = y_j (\sum_{i=1}^{n} x_j^T x_i \alpha_i^{(t)})$$

$$\begin{aligned} K_{j\cdot}\alpha^{(t)} &= \left[k(x_j, x_1) \cdots k(x_j, x_n)\right] (\alpha_1^{(t)}, \ldots, \alpha_n^{(t)})^T \\ &= \left[< x_j, x_1 > \cdots < x_j, x_n >\right] (\alpha_1^{(t)}, \ldots, \alpha_n^{(t)})^T \\ &= \sum_{i=1}^{n} x_j^T x_i \alpha_i^{(t)} \end{aligned}$$

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

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:
   For a point (x_j, y_j) that does not have a margin violation, then

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

$$\therefore \alpha_i^{(t+1)} = (1-\eta^{(t)}\lambda)\alpha_i^{(t)}, \alpha^{(t+1)} = (1-\eta^{(t)}\lambda)\alpha^{(t)}$$

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 $y_1, \ldots, y_n \in \{-1, 1\}$
   Solution:
   For a point (x_j, y_j) that does have a margin violation, then

$$w^{(t+1)} = (1-\eta^{(t)}\lambda)w^{(t)} + \eta^{(t)} y_j x_j$$
$$X^T\alpha^{(t+1)} = (1-\eta^{(t)}\lambda)X^T\alpha^{(t)} + \eta^{(t)} y_j X^T v_j,$$
$$\text{where } v_j = (0, 0, ..., 1, 0, ...0)^T \in \mathbf{R}^n, \text{ with its jth entry} = 1$$
$$\alpha^{(t+1)} = (1-\eta^{(t)}\lambda)\alpha^{(t)} + sign(y_j)\eta^{(t)} v_j$$

   So if point j violates the margin, we first update $\alpha^{(t+1)} = (1-\eta^{(t)}\lambda)\alpha^{(t)}$, then we update the jth entry of $\alpha^{(t+1)} : \alpha^{(t+1)}[j] = \alpha^{(t+1)}[j] + \eta^{(t)} y_j$

Algorithm 3: Kernelized Pegasos Algorithm

input: Kernel matrix K, label $\{y_1, y_2, ..., y_n\} \in \{-1, 1\}$ *and* $\lambda > 0$.
$\alpha^{(1)} = (0, \ldots, 0) \in \mathbf{R}^d$
$t = 0$ # step number
repeat
  $t = t + 1$
  $\eta^{(t)} = 1/(t\lambda)$ # step multiplier
  randomly choose j in $1, \ldots, n$
  if $y_j K_{j.}\alpha^{(t)} < 1$
    $\alpha^{(t+1)} = (1-\eta^{(t)}\lambda)\alpha^{(t)}$
    $\alpha^{(t+1)}[j] = \alpha^{(t+1)}[j] + \eta^{(t)} y_j$
  else
    $\alpha^{(t+1)} = (1-\eta^{(t)}\lambda)\alpha^{(t)}$
until bored
return $\alpha^{(t)}$

4. [Optional] 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.

Algorithm 4: Updated Kernelized Pegasos Algorithm

input: Kernel matrix K, label $\{y_1, y_2, ..., y_n\} \in \{-1, 1\}$ *and* $\lambda > 0$.
$\alpha^{(1)} = (0, \ldots, 0) \in \mathbf{R}^d$
$s^{(0)} = 1$
$t = 0$ # step number
repeat
  $t = t + 1$
  $\eta^{(t)} = 1/(t\lambda)$ # step multiplier
  $s^{(1)} = 0$
  randomly choose j in $1, \ldots, n$
  if $y_j K_{j.} s^{(t-1)} \alpha(1) < 1$
    $s^{(t+1)} = (1 - \eta^{(t)}\lambda) s^{(t)} + \eta^{(t)} y_j$
  else
    $s^{(t+1)} = ((1 - \eta^{(t)}\lambda)) s^{(t)}$
until bored
$\alpha^{(t)} = s^{(t-1)} \alpha^{(1)}$
return $\alpha^{(t)}$

6 Kernel Methods: Let's Implement

In this section you will get the opportunity to code kernel ridge regression and, optionally, kernelized SVM. To speed things along, we've written a great deal of support code for you, which you can find in the Jupyter notebooks in the homework zip file.

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

Consider the following optimization problem on a data set $(x_1, y_1), \ldots (x_n, y_n) \in \mathbf{R}^d \times \mathcal{Y}$:

$$\min_{w \in \mathbf{R}^d} R\left(\sqrt{\langle w, w \rangle}\right) + L\left(\langle w, x_1 \rangle, \ldots, \langle w, x_n \rangle\right),$$

where $w, x_1, \ldots, x_n \in \mathbf{R}^d$, and $\langle \cdot, \cdot \rangle$ is the standard inner product on $\mathbf{R}^d$. The function $R : [0, \infty) \to \mathbf{R}$ is nondecreasing and gives us our regularization term, while $L : \mathbf{R}^n \to \mathbf{R}$ is arbitrary[2] and gives us our loss term. We noted in lecture that this general form includes soft-margin SVM and ridge regression, though not lasso regression. Using the representer theorem, we showed if the optimization problem has a solution, there is always a solution of the form $w = \sum_{i=1}^n \alpha_i x_i$, for some $\alpha \in \mathbf{R}^n$. Plugging this into the our original problem, we get the following "kernelized" optimization problem:

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

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

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

and we can recover the original $w \in \mathbf{R}^d$ by $w = \sum_{i=1}^n \alpha_i x_i$.

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

More interestingly, we can replace k by another kernel $k''(x_i, x_j)$ for which we do not even know or cannot explicitly write down a corresponding feature map ψ. Our main example of this is the RBF kernel

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

for which the corresponding feature map ψ is infinite dimensional. In this case, we cannot recover w since it would be infinite dimensional. Predictions must be done using $\alpha \in \mathbf{R}^n$, with $f(x) = \sum_{i=1}^n \alpha_i k(x_i, x)$.

Your implementation of kernelized methods below should not make any reference to w or to a feature map ψ. Your "learning" routine should return α, rather than w, and your prediction function should also use α rather than w. This will allow us to work with kernels that correspond to infinite-dimensional feature vectors.

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

6.2 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 of the linear kernel (Bauckhage's article on calculating Euclidean distance matrices may be helpful).

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.

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.

   (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 $x \in [-6, 6]$.

   (d) 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 each input in the training set $x_1, \ldots, x_n$ 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

[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.

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.

6.3 Kernel Ridge Regression

In the zip file for this assignment, you'll find a training and test set, along with some skeleton code. We're considering a one-dimensional regression problem, in which $\mathcal{X} = \mathcal{Y} = \mathcal{A} = \mathbf{R}$. We'll fit this data using kernelized ridge regression, and we'll compare the results using several different kernel functions. Because the input space is one-dimensional, we can easily visualize the results.

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

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 kernel 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.

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?

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 $\lambda \to \infty$?

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 use sklearn's GridSearchCV. Note: Because of the small dataset size, these models can be fit extremely fast, so there is no excuse for not doing extensive hyperparameter tuning.

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.

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 $\varepsilon \sim \mathcal{N}\left(0, 0.1^2\right)$

(so $\text{Var}(\varepsilon) = 0.1^2$). The final point is $(x, f(x) + \varepsilon)$. What is the Bayes decision function and the Bayes risk for the loss function $\ell(\hat{y}, y) = (\hat{y} - y)^2$.

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 inspired.

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

6.4 [Optional] Kernelized Support Vector Machines with Kernelized Pegasos

1. [Optional] 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?

2. [Optional] 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.

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

7 Representer Theorem

Recall the following theorem from lecture:

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

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

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.)

   Solution:

$$\begin{gathered}\text{When } \|m_0\| = \|x_0\|, \|m_0\|^2 = \|x\|^2 \\ \because \|x\|^2 = \|m_0\|^2 + \|x - m_0\|^2 \\ \therefore \|x - m_0\|^2 = 0 \\ \langle x - m_0, x - m_0\rangle = 0 \\ \iff x - m_0 = 0 \iff m_0 = x\end{gathered}$$

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 $\|w\| = \|w^*\|$.)

   Solution: Suppose we have one minimizer w that does not have the form, which means it does not in the span of $\psi(x_i)$, let its projection onto the a closed subspace M of Hilbert space (which here is the span of $\psi(x_i)$) to be w_m then from the above question, we have proved that: $||w|| > ||w_m||$.
   Then $(w - w_m) \perp M$, so $w - w_m \perp \psi(x_i)$
   $\langle w, \psi(x_i)\rangle = \langle w_m + w - w_m, \psi(x_i)\rangle = \langle w_m, \psi(x_i)\rangle + \langle w - w_m, \psi(x_i)\rangle = \langle w_m, \psi(x_i)\rangle$
   Then because R is strictly increasing,

$$\begin{aligned}J(w) &= R\left(\|w\|\right) + L\left(\langle w, \psi(x_1)\rangle, \ldots, \langle w, \psi(x_n)\rangle\right) \\ &= R\left(\|w\|\right) + L\left(\langle w_m, \psi(x_1)\rangle, \ldots, \langle w_m, \psi(x_n)\rangle\right) \quad > R\left(\|w_m\|\right) + L\left(\langle w_m, \psi(x_1)\rangle, \ldots, \langle w_m, \psi(x_n)\rangle\right) = J(w_m)\end{aligned}$$

   Thus we find a contradiction that if w is a minimizer but not in the span, we can always find w's projection onto M that achieves smaller J(w). Thus all minimizers should have the form claimed.

3. [Optional] 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:
$L\left(\langle w, \psi(x_1)\rangle, \ldots, \langle w, \psi(x_n)\rangle\right) = L\left(\psi(x_1)^T w, \ldots, \psi(x_n)^T w\right) = L([\psi]w)$
$\because L$ is a convex function
$\therefore L([\psi]w)$is a convex function of w
And $R(||w||)$ is a convex function of w, so $J(w)$ is a convex function of w

8 Ivanov and Tikhonov Regularization

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.

8.1 Tikhonov optimal implies Ivanov optimal

Let $\phi : \mathcal{F} \to \mathbf{R}$ be any performance measure of $f \in \mathcal{F}$, and let $\Omega : \mathcal{F} \to [0, \infty)$ 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 $\lambda \geq 0$ we have the Tikhonov regularization solution

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

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

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

(Hint: Start by figuring out what r should be. 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).)

Solution:

Suppose f^* is not the optimum in (2), which means there $\exists f$ that satisfies $\phi(f) < \phi(f^*)$.
Also let $r = \lambda\Omega(f^*)$, then we have $\Omega(f^*) \leq r$ and $\Omega(f) \leq r$.
then we have $\phi(f) + \lambda\Omega(f) < \phi(f^*) + \lambda\Omega(f^*)$, so we find a new minimizer for (1)
Thus our assumption contradicts the fact that f^* solves (1).

8.2 [Optional] 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:

$$\begin{aligned} \text{minimize} \quad & \phi(w) \\ \text{subject to} \quad & \Omega(w) \leq r, \end{aligned}$$

for any $r \geq 0$. Assume that strong duality holds for this optimization problem and that the dual solution is attained (e.g. Slater's condition would suffice). Then we will show that there exists a $\lambda \geq 0$ such that $w^* \in \arg\min_{w \in \mathbf{R}^d} [\phi(w) + \lambda\Omega(w)]$.

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

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

2. [Optional] 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.]
   Solution:
   $$g(\lambda) = inf_w L(w, \lambda) = inf_w(\phi(w) + \lambda(\Omega(w) - r))$$
   $$\max g(\lambda)$$
   $$s.t. \lambda \geq 0$$

3. [Optional] 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^*. [Hint: You can use the same approach we used when we derived that strong duality implies complementary slackness.] **Conclude the proof** by showing that for the choice of $\lambda = \lambda^*$, we have $w^* \in \arg\min_{w \in \mathbf{R}^d} [\phi(w) + \lambda^*\Omega(w)]$.
   solution:
   $$\begin{aligned} \phi(w^*) = g(\lambda^*) &= \inf L(W, \lambda^*) \\ &\leq L(w^*, \lambda^*) \\ &= \phi(w^*) + \lambda^*(\omega(w^*) - r) \\ &\leq \phi(w^*) \end{aligned}$$

   $\therefore$ the minimum in the expression for $g(\lambda^*)$ is attained at w^*.

4. [Optional] The conclusion of the previous problem allows $\lambda = 0$, which means we're not actually regularizing at all. This will happen when the constraint in the Ivanov optimization problem is not active. That is, we'll need to take $\lambda = 0$ whenever the solution w^* to the Ivanov optimization problem has $\Omega(w^*) < r$. **Show this.** However, consider the following condition (suggested in [**?**]):
   $$\inf_{w \in \mathbf{R}^d} \phi(w) < \inf_{\{w | \Omega(w) \leq r\}} \phi(w).$$
   This condition simply says that we can get a strictly smaller performance measure (e.g. we can fit the training data strictly better) if we remove the Ivanov regularization. With this additional condition, show that if $\lambda^* \in \arg\max_{\lambda \geq 0} g(\lambda)$ then $\lambda^* > 0$. Moreover, show that the solution w^* satisfies $\Omega(w^*) = r$ – that is, the Ivanov constraint is active.

8.3 [Optional] 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. [Optional] Show that the Ivanov form of ridge regression

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

   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 $r > 0$.)

A Positive Semidefinite Matrices

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

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

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

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

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

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

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

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

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

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

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

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

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

we have

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

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

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

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

B Positive Definite Matrices

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.]

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

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

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

In [144]:
```
import numpy as np
import matplotlib.pyplot as plt
import sklearn
import scipy.spatial
import functools

%matplotlib inline
```

6.2.1

In [145]:
```
### Kernel function generators
def linear_kernel(X1, X2):
    """
    Computes the linear kernel between two sets of vectors.
    Args:
        X1 - an n1xd matrix with vectors x1_1,...,x1_n1 in the rows
        X2 - an n2xd matrix with vectors x2_1,...,x2_n2 in the rows
    Returns:
        matrix of size n1xn2, with x1_i^T x2_j in position i,j
    """
    return np.dot(X1,np.transpose(X2))

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

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

6.2.2

In [146]:
```
prototypes = np.array([-4,-1,0,2]).reshape(-1,1)
linear_kernel(prototypes, prototypes)
```

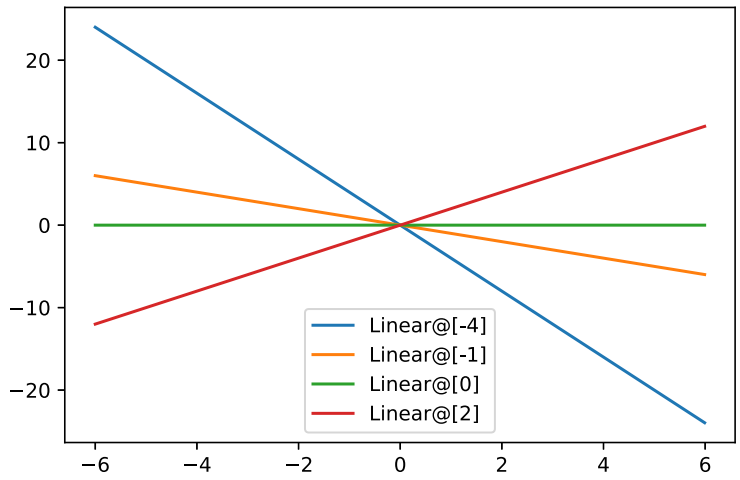
Out[146]:
```
array([[16,  4,  0, -8],
       [ 4,  1,  0, -2],
       [ 0,  0,  0,  0],
       [-8, -2,  0,  4]])
```

6.2.3

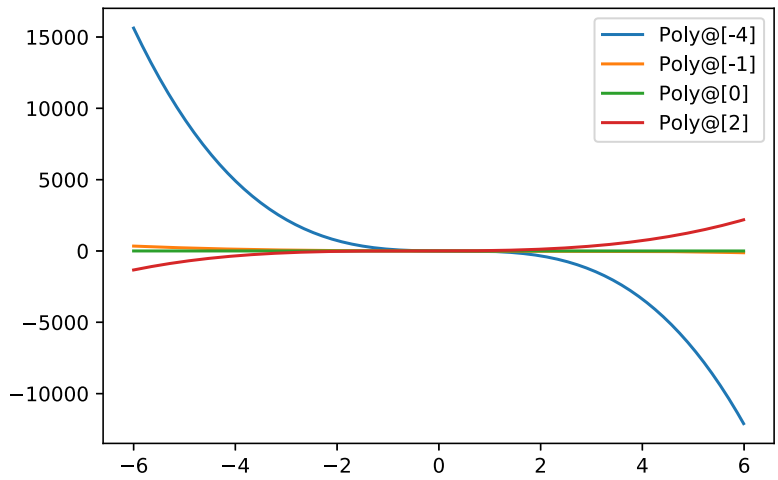
In [147]:
```
xpts = np.arange(-5.0, 6, .01).reshape(-1,1)
xpts
```

Out[147]:
```
array([[-5.  ],
       [-4.99],
       [-4.98],
       ...,
       [ 5.97],
       [ 5.98],
       [ 5.99]])
```

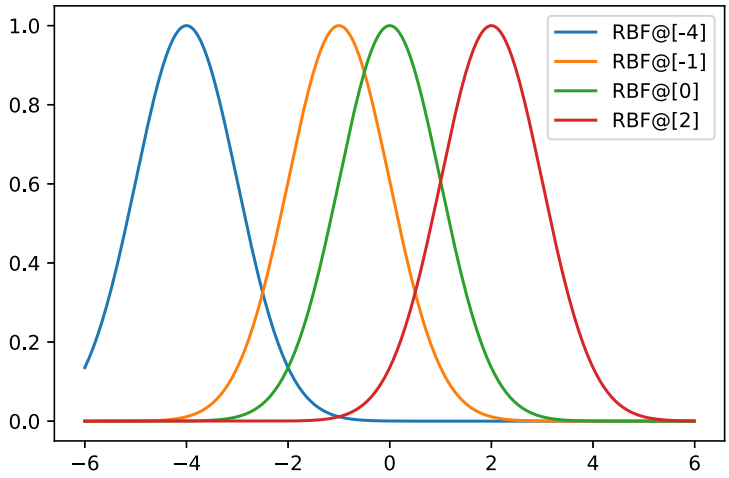
In [148]:
```
# PLot kernel machine functions
%config InlineBackend.figure_format = 'svg'
plot_step = .01
xpts = np.arange(-6.0, 6, plot_step).reshape(-1,1)
prototypes = np.array([-4,-1,0,2]).reshape(-1,1)

# 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()
```



In [149]:
```
# Poly kernel
y = polynomial_kernel(prototypes, xpts, 1, 3)
for i in range(len(prototypes)):
    label = "Poly@"+str(prototypes[i,:])
    plt.plot(xpts, y[i,:], label=label)
plt.legend(loc = 'best')
plt.show()
```



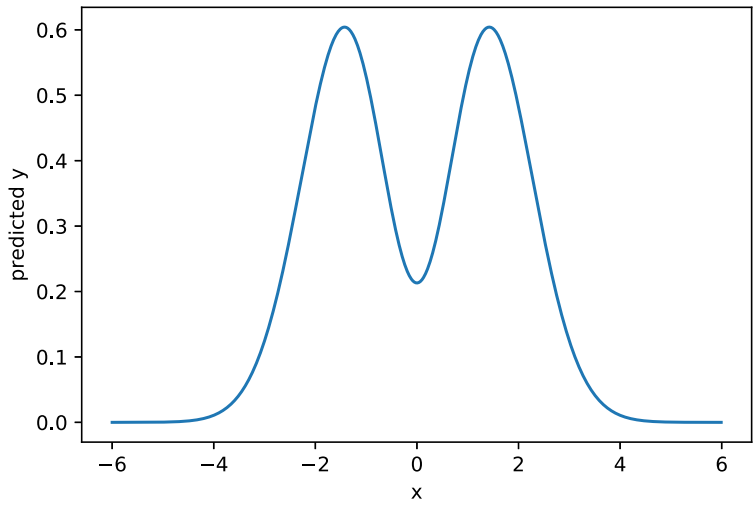
In [150]:
```
# RBF kernel
y = RBF_kernel(prototypes, xpts, 1)
for i in range(len(prototypes)):
    label = "RBF@"+str(prototypes[i,:])
    plt.plot(xpts, y[i,:], label=label)
plt.legend(loc = 'best')
plt.show()
```



In [151]:
```
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 and rows of X2 for kern
            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, jth entry of return vector
                Sum_{i=1}^R w_i k(x_j, mu_i)
        """
        # TODO
        k = self.kernel(X,self.prototype_points)
        return np.dot(k,self.weights)
```

In [152]:
```
prototypes = np.array([-1,0,1]).reshape(-1,1)
weights = np.array([1,-1,1]).reshape(-1,1)
k = functools.partial(RBF_kernel,sigma = 1)
km = Kernel_Machine(k,prototypes,weights)
```

In [153]:
```
plot_step = .01
xpts = np.arange(-6.0, 6, plot_step).reshape(-1,1)
plt.plot(xpts, km.predict(xpts), label=label)
plt.xlabel('x')
plt.ylabel('predicted y')
plt.show()
```

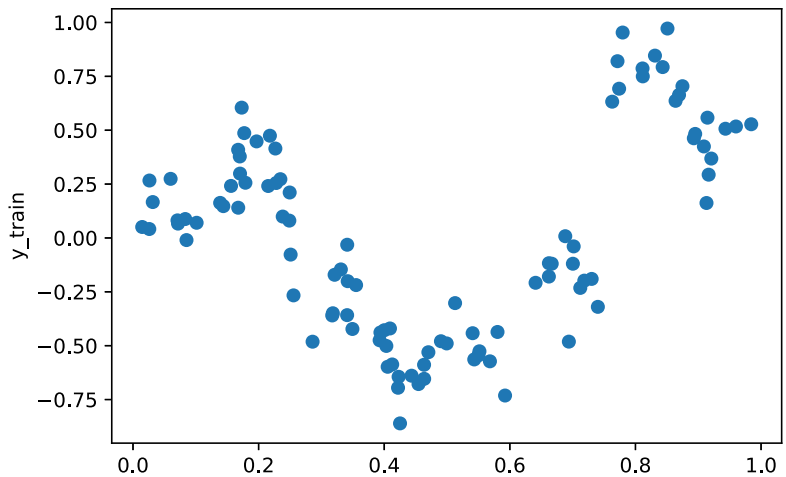


6.3.1

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

In [154]:
```
data_train,data_test = np.loadtxt("krr-train.txt"),np.loadtxt("krr-test.txt")
x_train, y_train = data_train[:,0].reshape(-1,1),data_train[:,1].reshape(-1,1)
x_test, y_test = data_test[:,0].reshape(-1,1),data_test[:,1].reshape(-1,1)
```

In [155]:
```
plt.plot(x_train,y_train,'o')
plt.xlabel('x_train')
plt.ylabel('y_train')
plt.show()
```



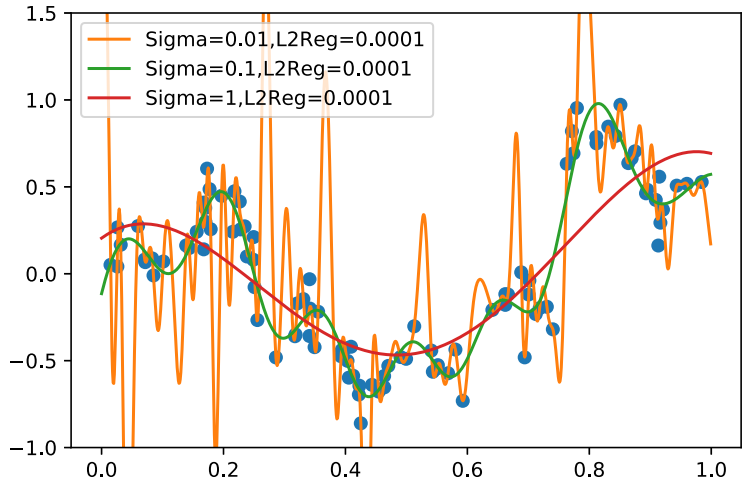
6.3.2

In [156]:
```
def train_kernel_ridge_regression(X, y, kernel, l2reg):
    # TODO
    dim = X.shape[0]
    alpha = np.dot(np.linalg.inv(l2reg * np.identity(dim) + kernel(X,X)),y)
    return Kernel_Machine(kernel, X, alpha)
```

6.3.3

- From the graph below, we can see that: sigma=0.01 is more likely to over fit while sigma =1 is less.

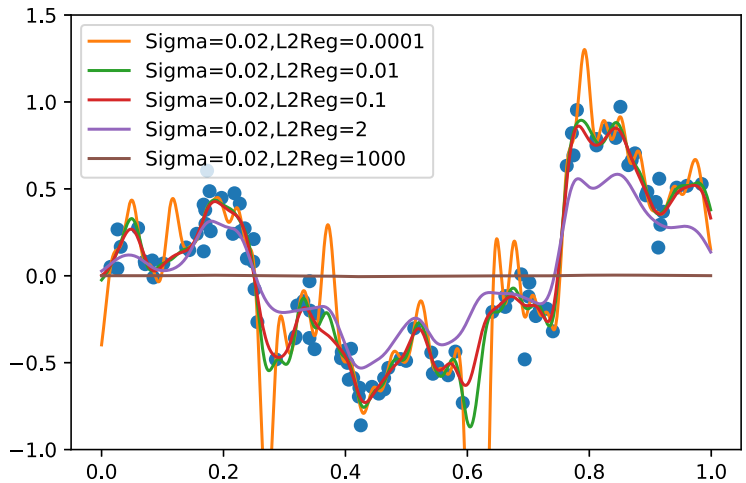
In [157]:
```
plot_step = .001
xpts = np.arange(0 , 1, plot_step).reshape(-1,1)
plt.plot(x_train,y_train,'o')
l2reg = 0.0001
for sigma in [.01,.1,1]:
    k = functools.partial(RBF_kernel, sigma=sigma)
    f = train_kernel_ridge_regression(x_train, y_train, k, l2reg=l2reg)
    label = "Sigma="+str(sigma)+",L2Reg="+str(l2reg)
    plt.plot(xpts, f.predict(xpts), label=label)
plt.legend(loc = 'best')
plt.ylim(-1,1.5)
plt.show()
```



6.3.4

- When $\lambda \rightarrow \infty$, the prediction function becomes a constant(looking like a straight line y=0 from the figure)

In [158]:
```
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,.1,2,1000]:
    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()
```



6.3.5

In [159]:
```
from sklearn.base import BaseEstimator, RegressorMixin, ClassifierMixin

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

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

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

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

        return self

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

        return(self.kernel_machine_.predict(X))

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

In [160]:
```
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 [191]:
```
param_grid = [{'kernel': ['RBF'],'sigma':[0.02,0.04,0.05,0.06,0.07,0.1,0.5], 'l2reg': np.exp2(-np.arange(-1,5,0.
              {'kernel':['polynomial'],'offset':np.arange(-5,5,0.5), 'degree':np.arange(2,10,1),'l2reg':np.exp2(
              {'kernel':['linear'],'l2reg': [10,5,4,3,2,1,0.1,.01,0.005]}]
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 machine
                    )
grid.fit(np.vstack((x_train,x_test)),np.vstack((y_train,y_test)))
```

Out[191]:
```
GridSearchCV(cv=PredefinedSplit(test_fold=array([-1, -1, ...,  0,  0])),
       error_score='raise-deprecating',
       estimator=KernelRidgeRegression(degree=2, kernel='RBF', l2reg=1, offset=1, sigma=1),
       fit_params=None, iid='warn', n_jobs=None,
       param_grid=[{'kernel': ['RBF'], 'sigma': [0.02, 0.04, 0.05, 0.06, 0.07, 0.1, 0.5], 'l2reg': array([2.
, 1.86607, 1.7411 , 1.6245 , 1.51572, 1.41421, 1.31951,
       1.23114, 1.1487 , 1.07177, 1.     , 0.93303, 0.87055, 0.81225,
       0.75786, 0.70711, 0.65975, 0.61557, 0.57435, 0.53589, 0.5    ...
       0.03125, 0.0221 ])}, {'kernel': ['linear'], 'l2reg': [10, 5, 4, 3, 2, 1, 0.1, 0.01, 0.005]}],
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring=make_scorer(mean_squared_error, greater_is_better=False),
       verbose=0)
```

In [192]:
```
pd.set_option('display.max_rows', 20)
df = pd.DataFrame(grid.cv_results_)
# Flip sign of score back, because GridSearchCV likes to maximize,
# so it flips the sign of the score if "greater_is_better=FALSE"
df['mean_test_score'] = -df['mean_test_score']
df['mean_train_score'] = -df['mean_train_score']
cols_to_keep = ["param_degree", "param_kernel","param_l2reg" ,"param_offset","param_sigma",
        "mean_test_score","mean_train_score"]
df_toshow = df[cols_to_keep].fillna('-')
df_toshow.sort_values(by=["mean_test_score"])
```

```
/Users/jr/anaconda3/lib/python3.7/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are acces
sing a training score ('split0_train_score'), which will not be available by default any more in 0.21. If you
need training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/Users/jr/anaconda3/lib/python3.7/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are acces
sing a training score ('mean_train_score'), which will not be available by default any more in 0.21. If you ne
ed training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
/Users/jr/anaconda3/lib/python3.7/site-packages/sklearn/utils/deprecation.py:125: FutureWarning: You are acces
sing a training score ('std_train_score'), which will not be available by default any more in 0.21. If you nee
d training scores, please set return_train_score=True
  warnings.warn(*warn_args, **warn_kwargs)
```

Out[192]:

	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_train_score
248	-	RBF	0.176777	-	0.06	0.013805	0.014446
255	-	RBF	0.164938	-	0.06	0.013808	0.014337
241	-	RBF	0.189465	-	0.06	0.013810	0.014562
262	-	RBF	0.153893	-	0.06	0.013819	0.014233
234	-	RBF	0.203063	-	0.06	0.013824	0.014686
269	-	RBF	0.143587	-	0.06	0.013838	0.014136
361	-	RBF	0.058315	-	0.07	0.013843	0.014489
368	-	RBF	0.054409	-	0.07	0.013845	0.014409
227	-	RBF	0.217638	-	0.06	0.013847	0.014817
354	-	RBF	0.062500	-	0.07	0.013847	0.014575
...	...	...	...	...	...	...	...
2524	8	polynomial	0.176777	-3	-	26.118150	14.846931
1408	5	polynomial	2.828427	-1	-	42.627495	39.888049
1349	4	polynomial	0.031250	-0.5	-	83.919292	74.118846
745	3	polynomial	4.000000	-2.5	-	176.827125	155.535490
1687	5	polynomial	0.022097	-1.5	-	207.650239	160.865511
1665	5	polynomial	0.031250	-2.5	-	209.931191	163.009402
2043	7	polynomial	2.828427	-3.5	-	210.301447	163.146338
2922	9	polynomial	0.044194	-4	-	362.248725	208.787529
2647	8	polynomial	0.022097	-1.5	-	442.130500	252.449933
1642	5	polynomial	0.044194	-4	-	1487.962466	1139.643705

2989 rows × 7 columns

In [193]:
```
# a more convenient way to look at the table
grid
nbinstall(overwrite=True)  # copies javascript dependencies to your /nbextensions folderd_toshow = df[show].filln
d.show_grid(df_toshow,show_toolbar=True)
```

Add Row | Remove Row

	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_train_s
2961	9	polynomial	0.0221	-4.5	-	0.03036	0.04016
2940	9	polynomial	0.03125	-5.0	-	0.03042	0.0403
2960	9	polynomial	0.0221	-5.0	-	0.03049	0.03973
2962	9	polynomial	0.0221	-4.0	-	0.03109	0.0417
2941	9	polynomial	0.03125	-4.5	-	0.03136	0.04193
2920	9	polynomial	0.04419	-5.0	-	0.03201	0.04257
1974	6	polynomial	0.04419	2.0	-	0.03242	0.04862
2013	6	polynomial	0.0221	1.5	-	0.03244	0.04823
1955	6	polynomial	0.0625	2.5	-	0.03249	0.04756
2136	7	polynomial	0.70711	3.0	-	0.03255	0.04765
2155	7	polynomial	0.5	2.5	-	0.03257	0.0494
1936	6	polynomial	0.08839	3.0	-	0.03257	0.04727
2078	7	polynomial	2	4.0	-	0.03259	0.04889
2097	7	polynomial	1.41421	3.5	-	0.0326	0.04942
1917	6	polynomial	0.125	3.5	-	0.03261	0.04741
2059	7	polynomial	2.82843	4.5	-	0.03263	0.0488

- 1. Linear kernel

In [177]:
```
q.get_changed_df()
```

Out[177]:

	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_train_score
5822	-	linear	4.000	-	-	0.164510	0.206563
5823	-	linear	3.000	-	-	0.164512	0.206538
5821	-	linear	5.000	-	-	0.164513	0.206592
5824	-	linear	2.000	-	-	0.164522	0.206518
5825	-	linear	1.000	-	-	0.164540	0.206506
5826	-	linear	0.100	-	-	0.164565	0.206501
5827	-	linear	0.010	-	-	0.164569	0.206501
5828	-	linear	0.005	-	-	0.164569	0.206501
5820	-	linear	10.000	-	-	0.164591	0.206780

- 2. RBF kernel
  (the best RBF: l2reg = 0.176777, sigma = 0.06)

In [176]:
```
q.get_changed_df()
```

Out[176]:

	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_train_score
248	-	RBF	0.176777	-	0.06	0.013805	0.014446
255	-	RBF	0.164938	-	0.06	0.013808	0.014337
241	-	RBF	0.189465	-	0.06	0.013810	0.014562
262	-	RBF	0.153893	-	0.06	0.013819	0.014233
234	-	RBF	0.203063	-	0.06	0.013824	0.014686
269	-	RBF	0.143587	-	0.06	0.013838	0.014136
361	-	RBF	0.058315	-	0.07	0.013843	0.014489
368	-	RBF	0.054409	-	0.07	0.013845	0.014409
227	-	RBF	0.217638	-	0.06	0.013847	0.014817
354	-	RBF	0.062500	-	0.07	0.013847	0.014575
...	...	...	...	...	...	...	...
69	-	RBF	1.071773	-	0.5	0.058221	0.093002
62	-	RBF	1.148698	-	0.5	0.059118	0.094221
55	-	RBF	1.231144	-	0.5	0.060095	0.095519
48	-	RBF	1.319508	-	0.5	0.061154	0.096897
41	-	RBF	1.414214	-	0.5	0.062297	0.098356
34	-	RBF	1.515717	-	0.5	0.063527	0.099897
27	-	RBF	1.624505	-	0.5	0.064843	0.101521
20	-	RBF	1.741101	-	0.5	0.066248	0.103227
13	-	RBF	1.866066	-	0.5	0.067739	0.105014
6	-	RBF	2.000000	-	0.5	0.069317	0.106880

420 rows × 7 columns

- 3. Polynomial Kernel
  (the best Poly: degree =9, l2reg = 0.022097, offset = -4.5)

In [194]:
```
q.get_changed_df()
```

Out[194]:

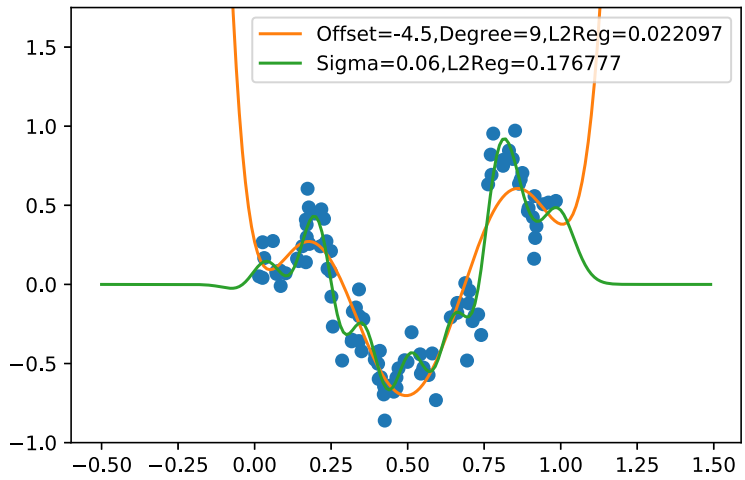
	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_train_score
2961	9	polynomial	0.022097	-4.5	-	0.030362	0.040158
2940	9	polynomial	0.031250	-5	-	0.030418	0.040295
2960	9	polynomial	0.022097	-5	-	0.030488	0.039725
2962	9	polynomial	0.022097	-4	-	0.031091	0.041698
2941	9	polynomial	0.031250	-4.5	-	0.031362	0.041928
2920	9	polynomial	0.044194	-5	-	0.032005	0.042569
1974	6	polynomial	0.044194	2	-	0.032416	0.048621
2013	6	polynomial	0.022097	1.5	-	0.032440	0.048233
1955	6	polynomial	0.062500	2.5	-	0.032487	0.047561
2136	7	polynomial	0.707107	3	-	0.032550	0.047654
...	...	...	...	...	...	...	...
2524	8	polynomial	0.176777	-3	-	26.118150	14.846931
1408	5	polynomial	2.828427	-1	-	42.627495	39.888049
1349	4	polynomial	0.031250	-0.5	-	83.919292	74.118846
745	3	polynomial	4.000000	-2.5	-	176.827125	155.535490
1687	5	polynomial	0.022097	-1.5	-	207.650239	160.865511
1665	5	polynomial	0.031250	-2.5	-	209.931191	163.009402
2043	7	polynomial	2.828427	-3.5	-	210.301447	163.146338
2922	9	polynomial	0.044194	-4	-	362.248725	208.787529
2647	8	polynomial	0.022097	-1.5	-	442.130500	252.449933
1642	5	polynomial	0.044194	-4	-	1487.962466	1139.643705

2560 rows × 7 columns

6.3.6

Answer: it seems that the RBF kernel fits data better(it captures more twist in the curve), and the polynomial kernel seems to be more general.

In [196]:
```
## 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= -4.5
degree = 9
l2reg = 0.022097
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 = 0.06
l2reg= 0.176777
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()
```



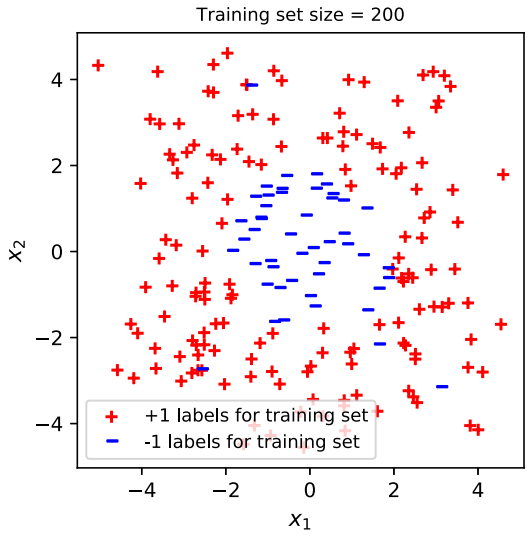
6.3.7

- from hw1, we know that the bayes decision function for square loss is: $E(y|x) = E(f(x) + \epsilon|x) = f(x)$
- from hw1, we know that the bayes risk for square loss is: $Var(y) = Var(f(x) + \epsilon) = Var(f(x)) + Var(\epsilon) = 0.01$

6.4.1

-**Answer**: not linearly separable,not quadratically separable. we can use some RBF kernel to separate the data properly.

In [197]:
```
# 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()
```



In [ ]:
```
def train_soft_svm(x_train, y_train, k, ...):
```

In [23]:
```
# 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 do

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-23-2d02ee1ae610> 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
```

In [ ]:
```

```