Homework 4: Kernel Methods

Instructions: Your answers to the questions below, including plots and mathematical work, should be submitted as a single PDF file. It's preferred that you write your answers using software that typesets mathematics (e.g. LaTeX, LyX, or MathJax via iPython), though if you need to you may scan handwritten work. You may find the minted package convenient for including source code in your LaTeX document. If you are using LyX, then the listings package tends to work better.

1 Introduction

The problem set begins with a couple problems on kernel methods: the first explores what geometric information about the data is stored in the kernel matrix, and the second revisits kernel ridge regression with a direct approach, rather than using the Representer Theorem. At the end of the assignment you will find an Appendix that reviews some relevant definitions from linear algebra, and gives some review exercises (**not for credit**). Next we have a problem that explores an interesting way to re-express the Pegasos-style SSGD on any ℓ_2 -regularized empirical risk objective function (i.e. not just SVM). The new expression also happens to allow efficient updates in the sparse feature setting. In the next problem, we take a direct approach to kernelizing Pegasos. Finally we get to our coding problem, in which you'll have the opportunity to see how kernel ridge regression works with different kernels on a one-dimensional, highly non-linear regression problem. There is also an optional coding problem, in which you can code a kernelized SVM and see how it works on a classification problem with a two-dimensional input space. The problem set ends with two theoretical problems. The first of these reviews the proof of the Representer Theorem. The second applies Lagrangian duality to show the equivalence of Tikhonov and Ivanov regularization (this material is optional).

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

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 $X^T Xw + \lambda Iw = X^T y$. 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 exercises on psd and spd matrices in the Appendix.)

   $\nabla J(w) = 2X^T(Xw - y) + 2\lambda Iw$
   $\nabla J(w) = 0 \Rightarrow X^T Xw + \lambda Iw = X^T y \Rightarrow (X^T X + \lambda I)w = X^T y \Rightarrow w = (X^T X + \lambda I)^{-1} X^T y$

   From the conclusion of $A.2$ and $B.3$, we know that it's invertible.

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

   $$w = \frac{1}{\lambda}(X^T y - X^T Xw)$$
   Hence, $w = X^T \alpha$ where $\alpha = \frac{1}{\lambda}(y - Xw)$

3. Based on the fact that $w = X^T \alpha$, explain why we say w is "in the span of the data."
   Since $w = \Sigma_{i=0}^{n} \alpha_i x_i$, it is a combination of some $x_i's$ in the training set. So w is in the span of 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.)

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

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.)

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

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.]

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

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 $v_i(w)$

[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).

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 $\eta^{(t)} = 1/(\lambda t)$, $t = 1, 2, \ldots$, 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.

(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

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

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 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 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 minimizer 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.

$$y_i \left\langle w^{(t)}, x_j \right\rangle = y_i \left\langle \Sigma_{t=1}^{n} \alpha_i^{(t)} x_i, x_j \right\rangle = y_i \Sigma_{t=1}^{n} \alpha_i^{(t)} k(x_i, x_j) = y_i K_{j\cdot} \alpha^{(t)}$$

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 $w^{(t+1)} = \sum_{i=1}^{n} \alpha_i^{(t+1)} x_i$.
   If we don't have a margin violation for the next step,

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

$$\Rightarrow \alpha_i^{(t+1)} = (1 - \eta^{(t)}\lambda) \alpha_i^{(t)}$$

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

$$w^{t+1} = (1 - \eta^{(t)}\lambda) w^{(t)} + \eta^{(t)} y_j x_j$$
$$X^T \alpha^{(t+1)} = (1 - \eta^{(t)}\lambda) X^{(t)} \alpha^{(t)} + \eta^{(t)} y_j x_j$$
$$\alpha^{(t+1)} = (1 - \eta^{(t)}\lambda) \alpha^{(t)} + \eta^{(t)} y_j [0, 0, ..., 1, ..0] \text{ where only the jth entry is 1.}$$

Algorithm 3: Pegasos Algorithm

input: Training set $(k_1, y_1), \ldots, (k_n, y_n) \in \mathbf{R}^d \times \{-1, 1\}$ 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 $y_j K_{j\cdot} \alpha^{(t)} < 1$
    $\alpha^{(t+1)} = (1 - \eta^{(t)}\lambda) \alpha^{(t)}$
    $\alpha_j^{(t+1)} = \alpha_j^{(t)} + \eta^{(t)} y_j$
  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 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 4 above.

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[3] 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 $\langle w, \psi(x_i) \rangle$.

[3] 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 $\langle w, x_i \rangle$.

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.

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[4], 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).

[4] 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.

```
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
    diff_mat = scipy.spatial.distance.cdist(X1,X2,'sqeuclidean')
    return np.exp(-(diff_mat)/(2*sigma**2))

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

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.

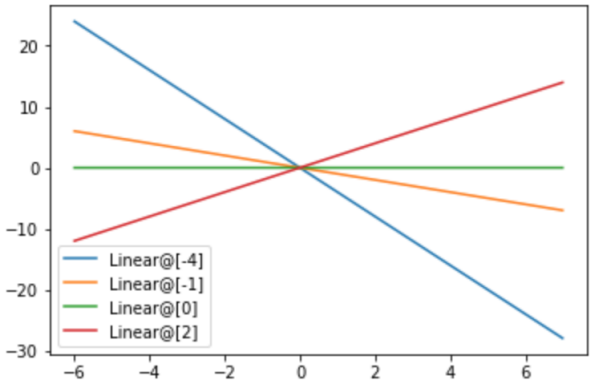
```
X0= np.array([[-4],[-1],[0],[2]])
linear_kernel(X0,X0)
```

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

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.

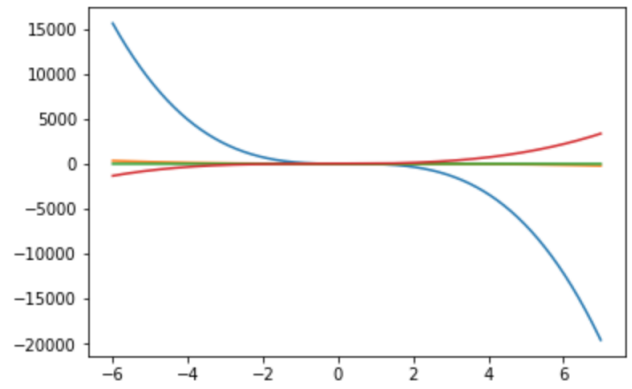
   (a) Plot the set of functions $x \mapsto k_{\text{linear}}(x_0, x)$ for $x_0 \in \mathcal{D}_X$ and for $x \in [-6, 6]$.

```
# PLot kernel machine functions

plot_step = .01
xpts = np.arange(-6.0, 7, plot_step).reshape(-1,1)
prototypes = np.array([-4,-1,0,2]).reshape(-1,1)

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



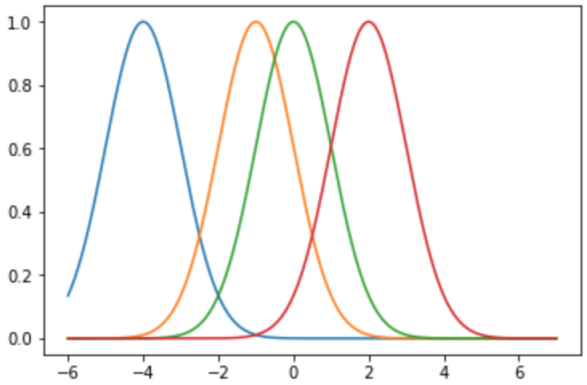
(b) Plot the set of functions $x \mapsto k_{\text{poly}(1,3)}(x_0, x)$ for $x_0 \in \mathcal{D}_X$ and for $x \in [-6, 6]$.

```
for i in range(len(prototypes)):
    label = "Poly@"+str(prototypes[i,:])
    plt.plot(xpts, y_poly[i,:], label=label)
```



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

```
for i in range(len(prototypes)):
    label = "RBF@"+str(prototypes[i,:])
    plt.plot(xpts, y_RBF[i,:], label=label)
```



(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.

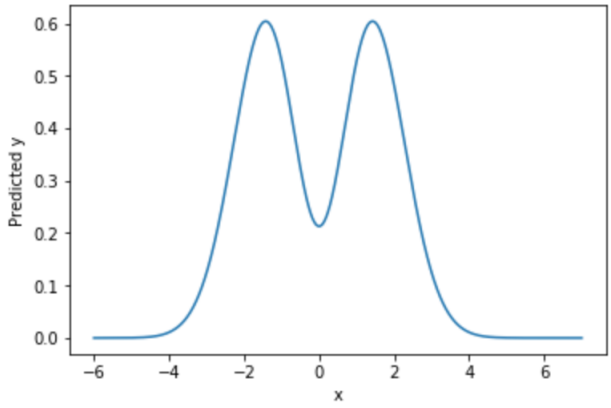
Note: For this problem, and for other problems below, it may be helpful to use partial application on your kernel functions. For example, if your polynomial kernel function has signature `polynomial_kernel(W, X, offset, degree)`, you can write `k = functools.partial(polynomial_kernel, offset=2, degree=2)`, 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.

```
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
            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,
                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)
```

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

```
array([[0.01077726],
       [0.52880462],
       [0.21306132],
       [0.48230437]])
```

```
plot_step = .001
xpts = np.arange(-6.0, 7, plot_step).reshape(-1,1)
plt.plot(xpts, machine.predict(xpts), label=label)
plt.xlabel('x')
plt.ylabel('Predicted y')
plt.show()
```



6.3 Kernel Ridge Regression

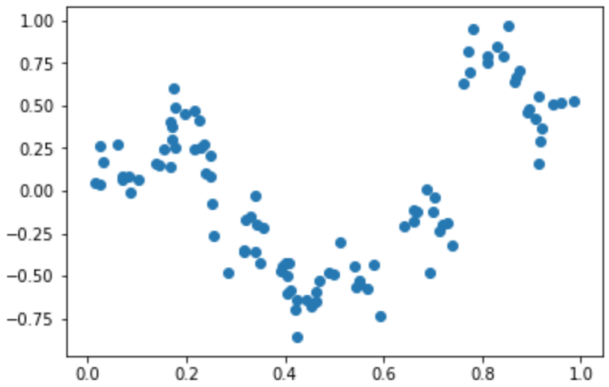
In the zip file for this assignment, you'll find a training and test set, along with some skeleton code. We're considering a one-dimensional regression problem, in which $\mathcal{X} = \mathcal{Y} = \mathcal{A} = \mathbf{R}$. We'll fit this data using kernelized ridge regression, and we'll compare the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

```
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)
```

```
plt.scatter(x_train,y_train)
```

```
<matplotlib.collections.PathCollection at 0xa190f5da0>
```



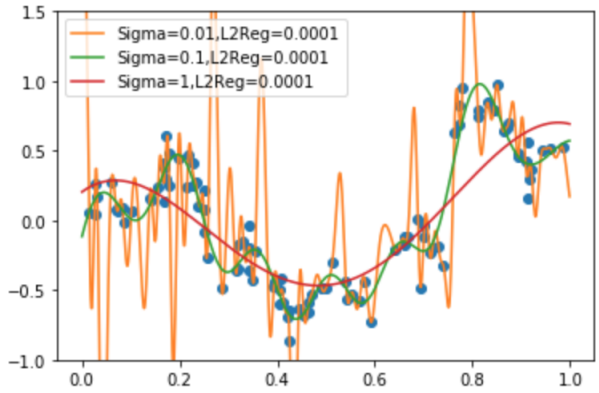
2. In a previous problem, we showed that in kernelized ridge regression, the final prediction function is $f(x) = \sum_{i=1}^{n} \alpha_i k(x_i, x)$, where $\alpha = (\lambda I + K)^{-1} y$ and $K \in \mathbf{R}^{n \times n}$ is the kernel matrix of the training data: $K_{ij} = k(x_i, x_j)$, for $x_1,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

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

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?

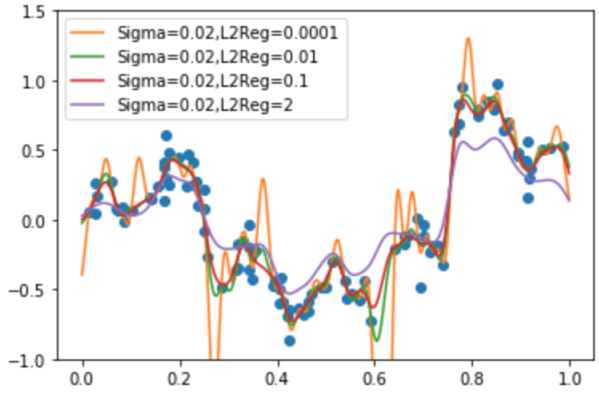
   Small sigmas are more likely to over fit while large sigmas are less fit.

```
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 [.01,0.1,1]:
    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()
```



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 $\lambda \to \infty$?

```
plot_step = .001
xpts = np.arange(0 , 1, plot_step).reshape(-1,1)
plt.plot(x_train,y_train,'o')
sigma= 0.02
l2regs= [.0001,.01,0.1,2]
for l2reg in l2regs:
    k = functools.partial(RBF_kernel, sigma=sigma)
    f = train_kernel_ridge_regression(x_train, y_train, k, l2reg=l2reg)
    label = "Sigma="+str(sigma)+",L2Reg="+str(l2reg)
    plt.plot(xpts, f.predict(xpts), label=label)
plt.legend(loc = 'best')
plt.ylim(-1,1.5)
plt.show()
```



As $\lambda \to \infty$, the prediction function becomes more and more flat, eventually to a flat line.

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.
Linear: l2reg =5
RBF: l2reg =0.5, sigma = 0.05
Polynomial: l2reg = 0.01, offset = 1, degree=6

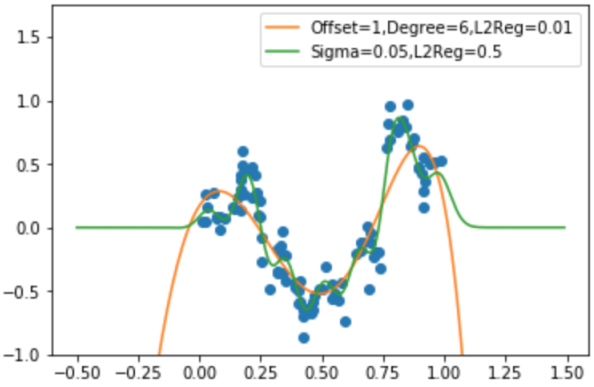
	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_train_score
637	-	linear	5	-	-	0.16451	0.20659
638	-	linear	1	-	-	0.16454	0.20651
639	-	linear	0.5	-	-	0.16455	0.2065
636	-	linear	10	-	-	0.16459	0.20678
635	-	linear	50	-	-	0.16568	0.20825
634	-	linear	100	-	-	0.16644	0.20916
633	-	linear	500	-	-	0.1676	0.2105
632	-	linear	1000	-	-	0.16781	0.21074

	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_train_score
9	-	RBF	0.5	-	0.05	0.01398	0.01492
5	-	RBF	0.1	-	0.05	0.01518	0.01219
13	-	RBF	1	-	0.05	0.01545	0.01794
1	-	RBF	0.05	-	0.05	0.01619	0.01157
0	-	RBF	0.05	-	0.01	0.01862	0.00493
4	-	RBF	0.1	-	0.01	0.0187	0.0059
2	-	RBF	0.05	-	0.1	0.02083	0.02283
6	-	RBF	0.1	-	0.1	0.02232	0.02415
8	-	RBF	0.5	-	0.01	0.02313	0.01293
10	-	RBF	0.5	-	0.1	0.02661	0.02811

	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_train_score
29	6	polynomial	0.01	1	-	0.0327	0.0495
17	5	polynomial	0.001	1	-	0.03409	0.04394
21	5	polynomial	0.01	10	-	0.0358	0.04348
36	7	polynomial	0.001	10	-	0.03685	0.04081
38	7	polynomial	0.01	1	-	0.03716	0.04536
24	5	polynomial	0.1	10	-	0.0374	0.05874
41	7	polynomial	0.1	1	-	0.03793	0.06316
33	6	polynomial	0.1	10	-	0.0381	0.0428
18	5	polynomial	0.001	10	-	0.03905	0.04281
26	6	polynomial	0.001	1	-	0.03914	0.0428
30	6	polynomial	0.01	10	-	0.03966	0.04252
35	7	polynomial	0.001	1	-	0.04001	0.04255
39	7	polynomial	0.01	10	-	0.04004	0.04214
42	7	polynomial	0.1	10	-	0.04013	0.04239
27	6	polynomial	0.001	10	-	0.04033	0.04234
20	5	polynomial	0.01	1	-	0.04052	0.06622

32	6	polynomial	0.1	1	-	0.04874	0.07771
23	5	polynomial	0.1	1	-	0.06131	0.0905
16	5	polynomial	0.001	0.1	-	0.06423	0.09556
25	6	polynomial	0.001	0.1	-	0.06471	0.09594
34	7	polynomial	0.001	0.1	-	0.07526	0.10767
19	5	polynomial	0.01	0.1	-	0.09748	0.12846
28	6	polynomial	0.01	0.1	-	0.1083	0.14177
22	5	polynomial	0.1	0.1	-	0.11251	0.14767
37	7	polynomial	0.01	0.1	-	0.11705	0.15254
31	6	polynomial	0.1	0.1	-	0.12183	0.15766
40	7	polynomial	0.1	0.1	-	0.12565	0.16177

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.



The prediction function with RBF kernel better fits the data than the polynomial kernel possibly because it's more flexible.

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 $\ell\left(\hat{y}, y\right) = \left(\hat{y} - y\right)^2$.

$$\begin{aligned}
\text{The Bayes decision function is } E(y|x) &= E(f(x) + \varepsilon|x) \\
&= E(f(x)|x) + E(\varepsilon) \\
&= f(x) \\
\text{Bayesian risk is } E((f(x) - y)^2) &= E(f(x) - f(x) - \epsilon)^2 \\
&= Var(\epsilon) + E^2(\epsilon) \\
&= 0.1^2
\end{aligned}$$

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*

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

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.)

   $\|x - m_0\|^2 = \langle x - m_0, x - m_0\rangle \geq 0$ and $\langle x - m_0, x - m_0\rangle = 0 \iff x - m_0 = 0$
   That is, $m_0 = x$
   Hence, $\|m_0\| = \|x\|$ only when $m_0 = x$

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

   Fix any $w \in \mathcal{H}$, let $w_m = Proj_M w$. Residual $w - w_M$ is orthogonal to x for all $x \in M$. $\langle w, x_i\rangle = \langle w_M + w - w_M, x_i\rangle = \langle w_M, x_i\rangle + \langle w - w_M, x_i\rangle = \langle w_m, x_i\rangle$. Hence, $L(\langle w, x_1\rangle, ..., \langle w, x_n\rangle) = L(\langle w_M, x_1\rangle, ..., \langle w_M, x_n\rangle)$.
   Now, first consider the case when $\|w\| < \|w^*\|$. Let w be the projection of $\|w^*\|$ onto span of the $\psi(x)'s$. Since R is non-decreasing, we have $R(\|w_M\|) < R(\|w\|)$, combining with the fact that $L(\langle w^*, x_1\rangle, ..., \langle w^*, x_n\rangle) = L(\langle w, x_1\rangle, ..., \langle w, x_n\rangle)$. w^* could not be a minimizer of $J(w)$.
   Next, consider the case $\|w\| = \|w^*\|$, from part one, we have $w = w^*$. w by definition is in the span of $\psi(x_i)$, therefore w^* should be in the span of $\psi(x_i)$ as well. That is, all minimizers must have the form claimed when R is strictly increasing.

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 $g(x) = f(Ax + b)$. 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

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).)

Let $r = \Omega(f^*)$. Suppose the Tikhonov solution is not the Ivanov solution, and let f^{**} denote the Ivanov solution. Since f^{**} is the Ivanov solution, we have $\phi(f^{**}) < \phi(f^*)$ Hence, $\phi(f^{**}) + \lambda\Omega(f^{**}) \leq \phi(f^{**}) + \lambda\Omega(f^*) < \phi(f^*) + \lambda\Omega(f^*)$, which contradicts the fact that f^* is the Tikhonov solution. Hence, f^* is the Ivanov solution.

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:

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

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 $w^* \in \arg\min_{w \in \mathbf{R}^d} \left[\phi(w) + \lambda\Omega(w)\right]$.

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

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.]

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 $w^* \in \arg\min_{w \in \mathbf{R}^d} \left[\phi(w) + \lambda^*\Omega(w)\right]$.

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 $\Sigma = \operatorname{diag}(\sigma_1, \ldots, \sigma_n)$, then $\Sigma^{-1} = \operatorname{diag}\left(\sigma_1^{-1}, \ldots, \sigma_n^{-1}\right)$. 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 $x^T(M + N)x = x^T Mx + x^T Nx$.)