DS-GA 1003: Machine Learning (Spring 2020)

Homework 4: Support Vector Machines and Kernels

Due: Monday, March 23, 2020 at 11:59pm

In the problem set, we will be working with text for sentiment analysis. You will get some practice with subgradients by relating the Perceptron Algorithm from Homework 1 to stochastic subgradient descent. Building on our understanding of stochastic subgradient descent and learning rates, we will implement the Pegasos Algorithm for Support Vector Machines. Since we are working with text, we need to have sparse representations of feature vectors, where only the non-zero entries are explicitly recorded. Following some practice with linear kernels, polynomial kernels and radial basis function kernels, we will incorporate kernels into the code. You will study kernels for both Support Vector Machines and Ridge Regression.

Instructions. You should upload your code and plots to Gradescope. Please map the Gradescope entry on the rubric to your name and NetId. You must follow the policies for submission detailed in Homework 0.

Datasets. For Support Vector Machines, we will be using the Polarity Dataset v2.0, constructed by Pang and Lee. It has the full text from 2000 movies reviews: 1000 reviews are classified as "positive" and 1000 as "negative." Our goal is to predict whether a review has positive or negative sentiment from the text of the review. Each review is stored in a separate file: the positive reviews are in a folder called "pos", and the negative reviews are in "neg".

The data is provided in `data.zip`. We have provided some code in `load.py` to assist with reading these files. The code removes some special symbols from the reviews. We have provided some utilities in `util.py` for calculations with sparse representations of vectors as dictionaries. Please check the supporting code in `skeleton_code_svm.ipyn` for some approaches to dividing the implementation.

For Ridge Regression, we will be using the datasets in `krr-train.txt` and `krr-test.txt`. We have provided you with supporting code in `skeleton_code_rr.ipynb` that adapts the code from Homework 3.

1 Perceptron

Recall that a vector $g \in \mathbf{R}^d$ is a **subgradient** of $f : \mathbf{R}^d \to \mathbf{R}$ at x if for all z,

$$f(z) \geq f(x) + g^T(z - x).$$

Remember that there may be 0, 1, or infinitely many subgradients at any point. The **subdifferential** of f at a point x, denoted $\partial f(x)$, is the set of all subgradients of f at x.

Just as there is a calculus for gradients, there is a calculus for subgradients. For our purposes, we can usually get by using the definition of subgradient directly. However, in the first problem below we derive a property that will make our life easier for finding a subgradient of the hinge loss and perceptron loss.

1 Suppose $f_1, \dots, f_m : \mathbf{R}^d \to \mathbf{R}$ are convex functions, and

$$f(x) = \max_{i=1,\dots,m} f_i(x).$$

Let k be any index for which $f_k(x) = f(x)$, and choose $g \in \partial f_k(x)$. We are using the fact that a convex function on $\mathbf{R}^d$ has a non-empty subdifferential at all points. Show that $g \in \partial f(x)$.

2 Give a subgradient of

$$J(w) = \max\left\{0, 1 - yw^T x\right\}.$$

We studied the perceptron algorithm in Homework 1. Suppose we have a labeled training set $(x_1, y_1), \dots, (x_n, y_n) \in \mathbf{R}^d \times \{-1, 1\}$. In the perceptron algorithm, we are looking for a hyperplane that perfectly separates the classes. That is, we're looking for $w \in \mathbf{R}^d$ such that

$$y_i w^T x_i > 0 \; \forall i \in \{1, \dots, n\}.$$

Visually, this would mean that all the x's with label $y = 1$ are on one side of the hyperplane $\left\{x \mid w^T x = 0\right\}$, and all the $x's$ with label $y = -1$ are on the other side. When such a hyperplane exists, we say that the data are **linearly separable**. The perceptron algorithm is given in Algorithm 1.

Algorithm 1: Perceptron Algorithm

```
input: Training set $(x_1, y_1), \dots, (x_n, y_n) \in \mathbf{R}^d \times \{-1, 1\}$
$w^{(0)} = (0, \dots, 0) \in \mathbf{R}^d$
$k = 0$ # step number
repeat
  all_correct = TRUE
  for $i = 1, 2, \dots, n$ # loop through data
    if $(y_i x_i^T w^{(k)} \leq 0)$
      $w^{(k+1)} = w^{(k)} + y_i x_i$
      all_correct = FALSE
    else
      $w^{(k+1)} = w^{(k)}$
    end if
    $k = k + 1$
  end for
until (all_correct == TRUE)
return $w^{(k)}$
```

The **perceptron loss** is given by

$$\ell(\hat{y}, y) = \max\{0, -\hat{y}y\}.$$

In this problem we will see why this loss function has this name.

3 Show that if $\left\{x \mid w^T x = 0\right\}$ is a separating hyperplane for a training set $\mathcal{D} = \left((x_1, y_1), \ldots, (x_n, y_n)\right)$, then the average perceptron loss on $\mathcal{D}$ is 0. Thus any separating hyperplane of $\mathcal{D}$ is an empirical risk minimizer for perceptron loss.

4 Let $\mathcal{H}$ be the linear hypothesis space consisting of functions $x \mapsto w^T x$. Consider running stochastic subgradient descent (SSGD) to minimize the empirical risk with the perceptron loss. We'll use the version of SSGD in which we cycle through the data points in each epoch. Show that if we use a fixed step size 1, we terminate when our training data are separated, and we make the right choice of subgradient, then we are exactly doing the Perceptron algorithm.

5 Suppose the perceptron algorithm returns w. Show that w is a linear combination of the input points. That is, we can write $w = \sum_{i=1}^{n} \alpha_i x_i$ for some $\alpha_1, \ldots, \alpha_n \in \mathbf{R}$. The x_i for which $\alpha_i \neq 0$ are called support vectors. Give a characterization of points that are support vectors and not support vectors.

2 Sparse Representations

Recall from Homework 1 that a basic way to represent text documents for machine learning is with a "bag-of-words" representation. Here we will use a variant where every possible word is a feature, and the value of a word feature is the number of times that word appears in the document. Of course, most words will not appear in any particular document, and those counts will be zero. Rather than store a huge number of zeros, we use a sparse representation, in which we only store the counts that are nonzero. The counts are stored in a key/value store (such as a dictionary in Python). For example, "Harry Potter and Harry Potter II" would be represented as the following Python dict: `x={’Harry’:2, ’Potter’:2, ’and’:1, ’II’:1}`. We will be using linear classifiers of the form $f(x) = w^T x$, and we can store the w vector in a sparse format as well, such as `w={’minimal’:1.3, ’Harry’:-1.1, ’viable’:-4.2, ’and’:2.2, ’product’:9.1}`. The inner product between w and x would only involve the features that appear in both `x` and `w`, since whatever doesn't appear is assumed to be zero. For this example, the inner product would be `x[Harry] * w[Harry] + x[and] * w[and] = 2*(-1.1) + 1*(2.2)`. To help you along, we've included two functions for working with sparse vectors: 1) a dot product between two vectors represented as dict's and 2) a function that increments one sparse vector by a scaled multiple of another vector, which is a very common operation. These functions are located in `util.py`. It is worth reading the code, even if you intend to implement it yourself. You may get some ideas on how to make things faster.

1 Load all the data and randomly split it into 1500 training examples and 500 validation examples.

2 Write a function that converts an example (e.g. a list of words) into a sparse bag-of-words representation. You may find Python's Counter to be useful. Note that a Counter is also a dictionary.

3 SVM with via Pegasos

In this question you will build an SVM using the Pegasos algorithm. To align with the notation used in the Pegasos paper[1], we're considering the following formulation of the SVM objective function:

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

Note that, for simplicity, we are leaving off the unregularized bias term b. Pegasos is stochastic subgradient descent using a step size rule $\eta_t = 1/(\lambda t)$. The pseudocode is given below:

```
Input: $\lambda > 0$. Choose $w_1 = 0, t = 0$
While termination condition not met
  For $j = 1, \ldots, m$ (assumes data is randomly permuted)
    $t = t + 1$
    $\eta_t = 1/(t\lambda)$;
    If $y_j w_t^T x_j < 1$
      $w_{t+1} = (1 - \eta_t\lambda)w_t + \eta_t y_j x_j$
    Else
      $w_{t+1} = (1 - \eta_t\lambda)w_t$
```

1 [Written] Consider the "stochastic" SVM objective function, which is the SVM objective function with a single training point: $J_i(w) = \frac{\lambda}{2}\|w\|^2 + \max\left\{0, 1 - y_i w^T x_i\right\}$. Recall that if i is selected uniformly from the set $\{1, \ldots, m\}$, then this stochastic objective function has the same expected value as the full SVM objective function. The function $J_i(\theta)$ is not differentiable everywhere. Give an expression for the gradient of $J_i(w)$ where it's defined, and specify where it is not defined.

2 [Written] Show that a subgradient of $J_i(w)$ is given by

$$g = \begin{cases} \lambda w - y_i x_i & \text{for } y_i w^T x_i < 1 \\ \lambda w & \text{for } y_i w^T x_i \geq 1. \end{cases}$$

You may use the following facts without proof: 1) If $f_1, \ldots, f_m : \mathbf{R}^d \to \mathbf{R}$ are convex functions and $f = f_1 + \cdots + f_m$, then $\partial f(x) = \partial f_1(x) + \cdots + \partial f_m(x)$. 2) For $\alpha \geq 0$, $\partial(\alpha f)(x) = \alpha \partial f(x)$. [Hint: Use the rules provided and the calculation in the first problem.]

3 [Written] Show that if your step size rule is $\eta_t = 1/(\lambda t)$, then doing SGD with the subgradient direction from the previous problem is the same as given in the pseudocode.

4 Implement the Pegasos algorithm to run on a sparse data representation. The output should be a sparse weight vector w. Note that our Pegasos algorithm starts at $w = 0$. In a sparse representation, this corresponds to an empty dictionary. **Note:** With this problem, you will need to take some care to code things efficiently. In particular, be aware that making copies

[1] Shalev-Shwartz et al.'s "Pegasos: Primal Estimated sub-GrAdient SOlver for SVM".

of the weight dictionary can slow down your code significantly. If you want to make a copy of your weights (e.g. for checking for convergence), make sure you don't do this more than once per epoch. **Also**: If you normalize your data in some way, be sure not to destroy the sparsity of your data. Anything that starts as 0 should stay at 0.

5 Note that in every step of the Pegasos algorithm, we rescale every entry of w_t by the factor $(1-\eta_t\lambda)$. Implementing this directly with dictionaries is very slow. We can make things significantly faster by representing w as $w = sW$, where $s \in \mathbf{R}$ and $W \in \mathbf{R}^d$. You can start with $s = 1$ and W all zeros (i.e. an empty dictionary). Note that both updates (i.e. whether or not we have a margin error) start with rescaling w_t, which we can do simply by setting $s_{t+1} = (1-\eta_t\lambda)\, s_t$. If the update is $w_{t+1} = (1-\eta_t\lambda)w_t + \eta_t y_j x_j$, then **verify that the Pegasos update step is equivalent to**:

$$\begin{aligned} s_{t+1} &= (1-\eta_t\lambda)\, s_t \\ W_{t+1} &= W_t + \frac{1}{s_{t+1}}\eta_t y_j x_j. \end{aligned}$$

There is one subtle issue with the approach described above: if we ever have $1-\eta_t\lambda = 0$, then $s_{t+1} = 0$, and we'll have a divide by 0 in the calculation for W_{t+1}. This only happens when $\eta_t = 1/\lambda$. With our step-size rule of $\eta_t = 1/(\lambda t)$, it happens exactly when $t = 1$. So one approach is to just start at $t = 2$. More generically, note that if $s_{t+1} = 0$, then $w_{t+1} = 0$. Thus an equivalent representation is $s_{t+1} = 1$ and $W = 0$. Thus if we ever get $s_{t+1} = 0$, simply set it back to 1 and reset W_{t+1} to zero, which is an empty dictionary in a sparse representation. **Implement the Pegasos algorithm with the (s, W) representation described above**.

6 Run both implementations of Pegasos on the training data for a couple epochs (using the bag-of-words feature representation described above). Make sure your implementations are correct by verifying that the two approaches give essentially the same result. Report on the time taken to run each approach.

7 Write a function that takes a sparse weight vector w and a collection of (x, y) pairs, and returns the percent error when predicting y using $\text{sign}(w^T x)$. In other words, the function reports the 0-1 loss of the linear predictor $x \mapsto w^T x$.

8 Using the bag-of-words feature representation described above, search for the regularization parameter that gives the minimal percent error on your test set. (You should now use your faster Pegasos implementation, and run it to convergence.) A good search strategy is to start with a set of regularization parameters spanning a broad range of orders of magnitude. Then, continue to zoom in until you're convinced that additional search will not significantly improve your test performance. Once you have a sense of the general range of regularization parameters that give good results, you do not have to search over orders of magnitude every time you change something (such as adding a new feature).

4 Error Analysis

The natural language processing domain is particularly nice in that one can often interpret why a model has performed well or poorly on a specific example, and sometimes it is not very difficult to come up with ideas for new features that might help fix a problem. The first step in this process is to look closely at the errors that our model makes.

1 Choose an input example $x = (x_1, \ldots, x_d) \in \mathbf{R}^d$ that the model got wrong. We want to investigate what features contributed to this incorrect prediction. One way to rank the importance of the features to the decision is to sort them by the size of their contributions to the score. That is, for each feature we compute $|w_i x_i|$, where w_i is the weight of the ith feature in the prediction function, and x_i is the value of the ith feature in the input x. Create a table of the most important features, sorted by $|w_i x_i|$, including the feature name, the feature value x_i, the feature weight w_i, and the product $w_i x_i$. Attempt to explain why the model was incorrect. Can you think of a new feature that might be able to fix the issue? Include a short analysis for at least 2 incorrect examples.

2 [Optional] Recall that the "score" is the value of the prediction $f(x) = w^T x$. We like to think that the magnitude of the score represents the confidence of the prediction. This is something we can directly verify or refute. Break the predictions into groups based on the score (you can play with the size of the groups to get a result you think is informative). For each group, examine the percentage error. You can make a table or graph. Summarize the results. Is there a correlation between higher magnitude scores and accuracy?

3 [Optional] Our objective is not differentiable when $y_i w^T x_i = 1$. Investigate how often and when we have $y_i w^T x_i = 1$ (or perhaps within a small distance of 1 – this is for you to explore) . Describe your findings. If we didn't know about subgradients, one might suggest just skipping the update when $y w^T x_i = 1$. Does this seem reasonable? What about shortening the step size by a small percentage?

5 Kernels

Before incorporating kernels into the Pegasos algorithm, we want to get some practice working with relationships between features.

1 For any two documents x and z, define $k(x, z)$ to equal the number of unique words that occur in both x and z (i.e., the size of the intersection of the sets of words in the two documents). Is this function a kernel? Justify your answer. Remember $k(x, z)$ is a kernel if there exists $\phi(x)$ such that $k(x, z) = \phi(x)^T \phi(z)$.

2 One way to construct kernels is to build them from simpler ones. Assuming $k_1(x, z)$ and $k_2(x, z)$ are kernels, then one can show that so are these:

(a) (scaling) $f(x)f(z)k_1(x, z)$ for any function $f(x) \in \mathcal{R}$,

(b) (sum) $k(x, z) = k_1(x, z) + k_2(x, z)$,

(c) (product) $k(x,z) = k_1(x,z)k_2(x,z)$.

Using the above rules and the fact that $k(x,z) = x^T z$ is a kernel, show that the following is also a kernel:

$$\left(1 + \left(\frac{x}{||x||_2}\right)^T \left(\frac{z}{||z||_2}\right)\right)^3 .$$

6 Kernel Pegasos

Intuitively, how can we understand the Pegasos algorithm?

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

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

after any number of steps t. 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.

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.

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

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

7 Kernel Ridge Regression

We want to adapt the approach from Homework 3 to incorporate kernels into Ridge Regression.

7.1 Dual Form of Ridge Regression

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 $w = (X^T X + \lambda I)^{-1} X^T y$. You do not need to justify that the matrix $X^T X + \lambda I$ is invertible, for $\lambda > 0$ because we discussed these properties of matrices together in Tutoring Session 6.

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.

3. Based on the fact that $w = X^T \alpha$, explain why we say 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. Try replacing w by $X^T \alpha$ in the expression for α, and then solve for α.)

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

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. 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.

7.2 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 our implementations. For simplicity, and because it is by far the most common situation, 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 $k_{\text{poly}(a,d)}(x,x') = \left(a + \langle x,x'\rangle\right)^d$. 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).

   For the RBF kernel, you may use the scipy function `cdist(X1,X2,'sqeuclidean')` in the package `scipy.spatial.distance`.

2. 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.). 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.

7.3 Implementing Kernel Ridge Regression

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.

Acknowledgement: This problem set is based partly on assignments developed by Percy Liang, Cynthia Rudin and David Rosenberg.