Net Id: dpj7913 Name: Devashish Joshi

DSGA-1003 Machine Learning

HW-3

1. Suppose $f_1, \ldots, f_m : \mathbb{R}^d \to \mathbb{R}$ are convex functions, and $f(\boldsymbol{x}) = \max_{i=1,\ldots,m} f_i(\boldsymbol{x})$. Let k be any index for which $f_k(\boldsymbol{x}) = f(\boldsymbol{x})$, and choose $g \in \partial f_k(\boldsymbol{x})$ (a convex function on $\mathbb{R}^d$ has a non-empty subdifferential at all points). Show that $g \in \partial f(\boldsymbol{x})$.

$$f(x) = \begin{cases} f_1(x) & \text{if } f_1(x) \text{ is the maximum} \\ f_2(x) & \text{if } f_2(x) \text{ is the maximum} \\ . & \\ . & \\ . & \\ f_m(x) & \text{if } f_m(x) \text{ is the maximum} \end{cases} \tag{1}$$

Here, let the range of x in which $f_1(x)$ is maximum be denoted by X_1

Similarly when $f_2(x)$ is maximum, let range be X_2

It is given that k is an index for which $f_k(x) = f(x)$, and this means that we are restricted to the range X_k.

In this range X_k, $f_k(x)$ is the maximum, so $f_k(x) = f(x)$

$f(x)$ is non-differentiable at intersection points. The sub-differentiation of $f(x)$ at these points includes all values in the **range: [differentiation of the smaller convex function, differentiation of the greater convex function]**. This is because all the given functions are convex

> Thus, a subgradient selected such that $g \in \partial f_k(x)$ will be included in this range (as at non-intersection points, **in range** X_k, $\partial f_k(x)$ is exactly equal to $\partial f(x)$, and at the intersection points, $\partial f_k(x) \in \partial f(x)$).

So, if $g \in \partial f_k(x)$, then g also $\in \partial f(x)$

2. Give a subgradient of the hinge loss objective $J(\boldsymbol{w}) = \max\{0, 1 - y\boldsymbol{w}^T\boldsymbol{x}\}$.

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

$$J(w) = \begin{cases} 1 - y\boldsymbol{w}^T(x) & \text{if } y\boldsymbol{w}^T(x) < 1 \\ 0 & \text{if } y\boldsymbol{w}^T(x) \geq 1 \end{cases} \tag{2}$$

If the sub-differentiation is $\partial J(w)$, then

$$\partial J(w) = \begin{cases} -yx & \text{if } y\boldsymbol{w}^T(x) < 1 \\ 0 & \text{if } y\boldsymbol{w}^T(x) > 1 \\ [-yx, 0] & \text{if } y\boldsymbol{w}^T(x) = 1 \end{cases} \tag{3}$$

When $y\boldsymbol{w}^T(x) = 1$, then the sub-differentiation can take any value from $[-yx, 0]$ if $-yx$ is positive, and the range is $[0, -yx]$ if $-yx$ is negative. All sub-gradients will lie $\in \partial J(w)$.

One possible sub-gradient is:

$$\partial J(w) = \begin{cases} -yx & \text{if } y\boldsymbol{w}^T(x) < 1 \\ 0 & \text{if } y\boldsymbol{w}^T(x) \geq 1 \end{cases} \tag{4}$$

3. Consider the SVM objective function for a single training point[1]: $J_i(\boldsymbol{w}) = \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \max\{0, 1 - y_i \boldsymbol{w}^T \boldsymbol{x}_i\}$. The function $J_i(\boldsymbol{w})$ is not differentiable everywhere. Specify where the gradient of $J_i(w)$ is not defined. Give an expression for the gradient where it is defined.

   Gradient not defined at those values of w for which $y_i \boldsymbol{w}^T \boldsymbol{x}_i = 1$

   w is:

$$(1/y_i) * (x_i^{-1})^T$$

   When $w^T \neq (1/y_i) * x_i^{-1}$

$$\text{Gradient of } J_i(w) = \begin{cases} \lambda w - y_i x_i & \text{if } y_i w^T x_i < 1 \\ \lambda w & \text{if } y w^T x > 1 \end{cases} \tag{5}$$

[1] Recall that if i is selected uniformly from the set $\{1, \ldots, n\}$, then this objective function has the same expected value as the full SVM objective function.

4. Show that a subgradient of $J_i(w)$ is given by

$$\boldsymbol{gw} = \begin{cases} \lambda \boldsymbol{w} - y_i \boldsymbol{x}_i & \text{for } y_i \boldsymbol{w}^T \boldsymbol{x}_i < 1 \\ \lambda \boldsymbol{w} & \text{for } y_i \boldsymbol{w}^T \boldsymbol{x}_i \geq 1. \end{cases}$$

You may use the following facts without proof: 1) If $f_1, \ldots, f_n : \mathbb{R}^d \to \mathbb{R}$ are convex functions and $f = f_1 + \cdots + f_n$, then $\partial f(\boldsymbol{x}) = \partial f_1(\boldsymbol{x}) + \cdots + \partial f_n(\boldsymbol{x})$. 2) For $\alpha \geq 0$, $\partial (\alpha f)(x) = \alpha \partial f(x)$. (Hint: Use the first part of this problem.)

Solution:
A subgradient is defined over all values. In the previous question, the expression of gradient is given where it's defined.
The subgradient is equal to the gradient for those values of w where gradient is defined

Now, we will find the value of the subgradient at that value of w where the gradient is not defined.

Note that the function $J_i(w)$ is a convex function because:

1. first term $\frac{\lambda}{2} w^T w$ is a quadratic function, and we assume λ is ≥ 0
2. second term is a max function of 2 convex functions
3. sum of 2 convex functions is a convex function

So, the subgradient can be any value in range [min($\lambda w - y_i x_i$, λw), max($\lambda w - y_i x_i$, λw)] at value of w for which $y_i w^T x_i = 1$

So, a subgradient of $J_i(w)$ is

$$\boldsymbol{gw} = \begin{cases} \lambda \boldsymbol{w} - y_i \boldsymbol{x}_i & \text{for } y_i \boldsymbol{w}^T \boldsymbol{x}_i < 1 \\ \lambda \boldsymbol{w} & \text{for } y_i \boldsymbol{w}^T \boldsymbol{x}_i \geq 1. \end{cases}$$

5. Write a function that converts an example (a list of words) into a sparse bag-of-words representation. You may find Python's Counter[2] class to be useful here. Note that a Counter is itself a dictionary.

[11]:
```
def list_to_dict(reviews):
    sparse_bag_of_words_reviews = []

    for review in reviews:
        cnt = Counter()
        for word in review:
            cnt[word] += 1

        sparse_bag_of_words_reviews.append(cnt)

    return sparse_bag_of_words_reviews
```

[2] https://docs.python.org/2/library/collections.html

6. Load all the data and split it into 1500 training examples and 500 validation examples. Format the training data as a list X_train of dictionaries and y_train as the list of corresponding 1 or -1 labels. Format the test set similarly.

   Solution:
   The functions folder_list, read_data and increment were not modified at all, so are not included here.
   The function load_and_shuffle data was modified, and is included
   The function list_to_dict is used here, which was defined in Q5

[1]:
```
import os
import numpy as np
import random
from collections import Counter
from numpy import float128

def load_and_shuffle_data():
    '''
    pos_path is where you save positive review data.
    neg_path is where you save negative review data.
    '''
    pos_path = "data_reviews/pos"
    neg_path = "data_reviews/neg"

    pos_reviews = folder_list(pos_path,1)
    neg_reviews = folder_list(neg_path,-1)

    reviews =  pos_reviews + neg_reviews
    # Construct y such that there are first the positive reviews with value␣
 ↪1
    # and then negative reviews with value -1
    y = [1]*len(pos_reviews) + [-1]*len(neg_reviews)

    # Shuffle both arrays together
    temp = list(zip(reviews, y))
    random.shuffle(temp)
    reviews, y = zip(*temp)

    return reviews, y
```

[ ]:
```
reviews, y = load_and_shuffle_data()
```

[3]:
```
sparse_bag_of_words_reviews = list_to_dict(reviews)
```

[3]:
```
w_initial = Counter()
```

[4]:
```
X = sparse_bag_of_words_reviews

X_train, X_test = X[:1500], X[1500:]

y_train, y_test = y[:1500], y[1500:]
```

7. Implement the Pegasos algorithm to run on a sparse data representation. The output should be a sparse weight vector $\boldsymbol{w}$ represented as a dictionary. Note that our Pegasos algorithm starts at $w = 0$, which 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 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.

Solution:
Note: We initialize $t = 1$ instead of 0 because we want to compare the 2 Pegasos algorithms, and the second one has $t = 1$ as the initialization.

We use the worst loss from the last 10 epochs to check for convergence. Loss after an epoch has to better than the worst loss from last 10 epochs

We will also limit the number of epochs to a fixed threshold, as we don't want to train the slower algorithm to convergence (this will take too much time)

[10]:
```
from numpy import float128
import time
from collections import deque
```

[11]:
```
def hinge_loss(X, y, W, lambda_, scale=1):
    return lambda_/2 * dotProduct(W, W)*scale*scale + \
                1/len(X) * sum([ max(0, 1-y_i*dotProduct(X_i, W)*scale)␣
 ↪for X_i, y_i in zip(X, y) ])
```

[36]:
```
def pegasos_algorithm(X, y, w, lambda_=0.1, max_epochs=500):
    epoch = 0
    t=1
    scale=float128(1)

    initial_loss = hinge_loss(X, y, w, lambda_)
    loss_values = deque([initial_loss])
    delta_loss = -1.0

    while epoch<max_epochs and (delta_loss<-(1e-10) or epoch<=10):
        epoch+=1
        print('Epoch number: '+str(epoch))

        start_time = time.time()

        np.random.seed(epoch)
```

```
        p = np.random.RandomState(seed=epoch).permutation(len(X))

        for j in p:
            t+=1
            eta = 1/(t*lambda_)

            scale = (1-(1/t))

            for key, value in w.items():
                w[key] = value * scale

            if y[j]*dotProduct(X[j], w)<1:
                for key, value in X[j].items():
                    w[key] = w.get(key, 0) + eta*value*y[j]

        end_time = time.time()
        print("Time taken for epoch: "+str(end_time-start_time))

        loss = hinge_loss(X, y, w, lambda_)
        delta_loss = loss-max(loss_values)

        loss_values.append(loss)

        if len(loss_values)>=10:
            loss_values.popleft()
```

[37]:
```
w_slow = w_initial.copy()
pegasos_algorithm(X_train, y_train, w_slow, max_epochs=3)
```

8. 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:

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

Implement the Pegasos algorithm with the (s, W) representation described above. [3]

Solution:
Given update of s and W is equivalent to update of w because when multiplying the second equation on both sides by s_{t+1}, and then using value from first equation, the lhs is: w_{t+1}, and rhs is $(1 - \eta_t\lambda)w_t + \eta_t y_j x_j$

Now we will implement the modified algorithm

[39]:
```
def fast_pegasos_algorithm(X, y, W, max_epochs=200, lambda_=10):
    epoch = 0
    t=1
    scale=float128(1)

    initial_loss = hinge_loss(X, y, W, lambda_, scale)
    loss_values = deque([initial_loss])
    delta_loss = -1.0

    while epoch<max_epochs and (delta_loss<-(1e-10) or epoch<=10 ):
        epoch+=1
        print('Epoch number: '+str(epoch))

        start_time = time.time()

        p = np.random.RandomState(seed=epoch).permutation(len(X))

        for j in p:
            t+=1
            eta = 1/(t*lambda_)
            scale *= (1-(1/t))

            if (y[j]*dotProduct(X[j], W)*scale)<1:
                for key, value in X[j].items():
                    W[key] = W.get(key, 0) + 1/scale*eta*value*y[j]

        end_time = time.time()
```

[3] 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.

```
        print("Time taken for epoch: "+str(end_time-start_time))

        loss = hinge_loss(X, y, W, lambda_, scale)
        delta_loss = loss-max(loss_values)

        loss_values.append(loss)

        if len(loss_values)>=10:
            loss_values.popleft()

    for key, value in W.items():
        W[key] = value * scale
```

[40]:
```
w_fast = w_initial.copy()
fast_pegasos_algorithm(X_train, y_train, w_fast, max_epochs=3)
```

9. Run both implementations of Pegasos on the training data for a couple epochs. 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.

Solution:
Running both the implementations, the outputs are:

First (Slow) algo:

[37]:
```
w_slow = w_initial.copy()
all_loss = pegasos_algorithm(X_train, y_train, w_slow, max_epochs=3)
```

```
Epoch number: 1
Time taken for epoch: 4.334391117095947
Epoch number: 2
Time taken for epoch: 7.12932825088501
Epoch number: 3
Time taken for epoch: 7.741525888442993
```

.

Second (Fast) algo:

[40]:
```
w_fast = w_initial.copy()
all_loss = fast_pegasos_algorithm(X_train, y_train, w_fast, max_epochs=3)
```

```
Epoch number: 1
Time taken for epoch: 0.7983889579772949
Epoch number: 2
Time taken for epoch: 0.8564321994781494
Epoch number: 3
Time taken for epoch: 0.8295848369598389
```

The difference between the weights are in order of negative 8 of base 10.

Output of $w\_fast - w\_slow$ (of just the first few words):

```
[42]: Counter({"there's": 1.9093534770051071663e-08,
               'no': 2.6094201288602322917e-08,
               'reason': 1.5012219506776208503e-08,
               'to': 1.1130859808931355797e-08,
               'donnie': 6.6651855143301911133e-10,
               'brasco': 2.8482559431237495146e-09,
               'opening': 1.2330593201510704253e-09,
               'credits': 4.7011775161075364O7e-09,
               'proclaim': 1.1108642523884748947e-11,
               .
               .
               .
```

10. Write a function classification_error that takes a sparse weight vector w, a list of sparse vectors X and the corresponding list of labels y, and returns the fraction of errors when predicting y_i using $\text{sign}(\boldsymbol{w}^T \boldsymbol{x}_i)$. In other words, the function reports the 0-1 loss of the linear predictor $f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x}$.

Solution:

[33]:
```
def classification_error(X, y, W, scale=1):
    error=0

    for X_i, y_i in zip(X, y):
        y_pred=-1

        if dotProduct(W, X_i)*scale>0:
            y_pred=1

        if not y_pred==y_i:
            error+=1

    error/=len(X)

    return error
```

11. 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. Plot the test errors you obtained as a function of the parameters λ you tested. (Hint: the error you get with the best regularization should be closer to 15% than 20%. If not, maybe you did not train to convergence.)

Solution:
First, the values of λ selected were as follows:

$[0.0001, 0.01, 1, 100]$
For these values, the number of epochs was limited to 100, classification errors achieved were $3\%, 12\%, 8\%, 51\%$

This means the best range to zoom in on would be in $[0.0001, 0.01]$.

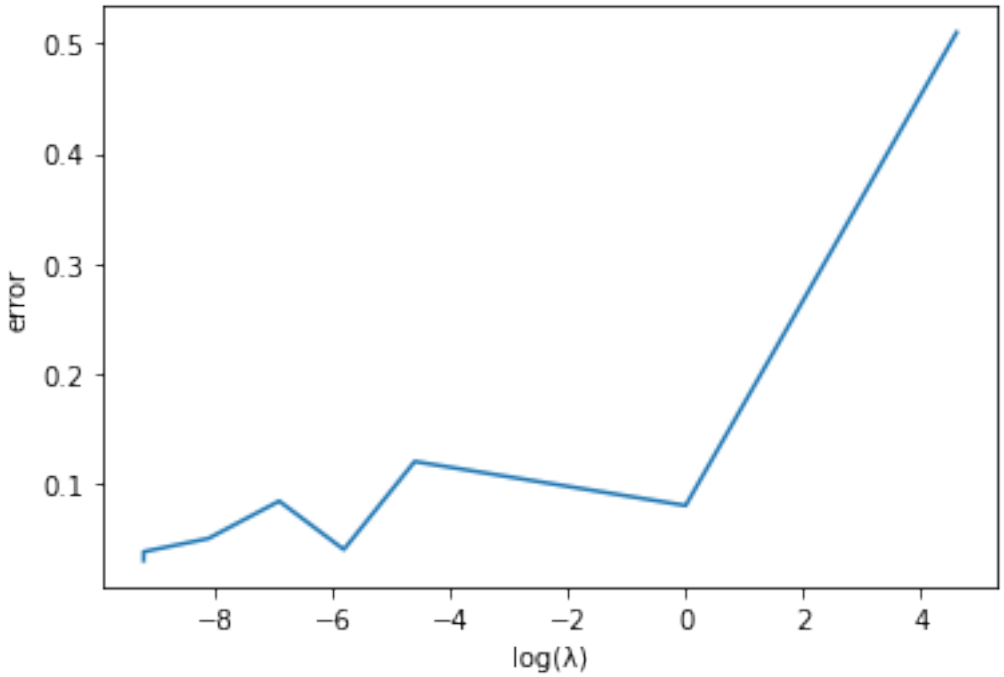
So, the values of λ selected were as follows:

$[0.0001, 0.0003, 0.001, 0.003, 0.01]$
For these values, the number of epochs was limited to 100, classification errors achieved were $3\%, 5\%, 8\%, 4\%, 12\%$

There may be multiple areas possible to be zoom in here, but we already have achieved an error better than what was given in the hint (closer to 15% than 20%), so this is where we stop

Best regularization parameter: 0.0001

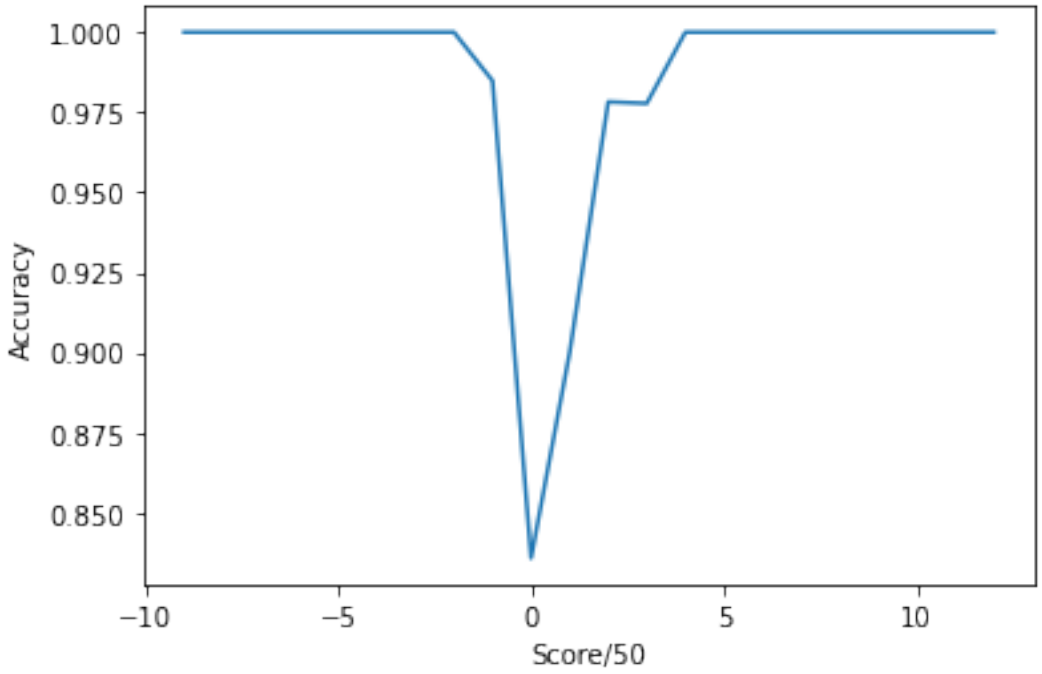
Graph:


0.5
0.4
0.3
0.2
0.1
error
−8
−6
−4
−2
0
2
4
log(λ)

12. Break the predictions on the test set 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?

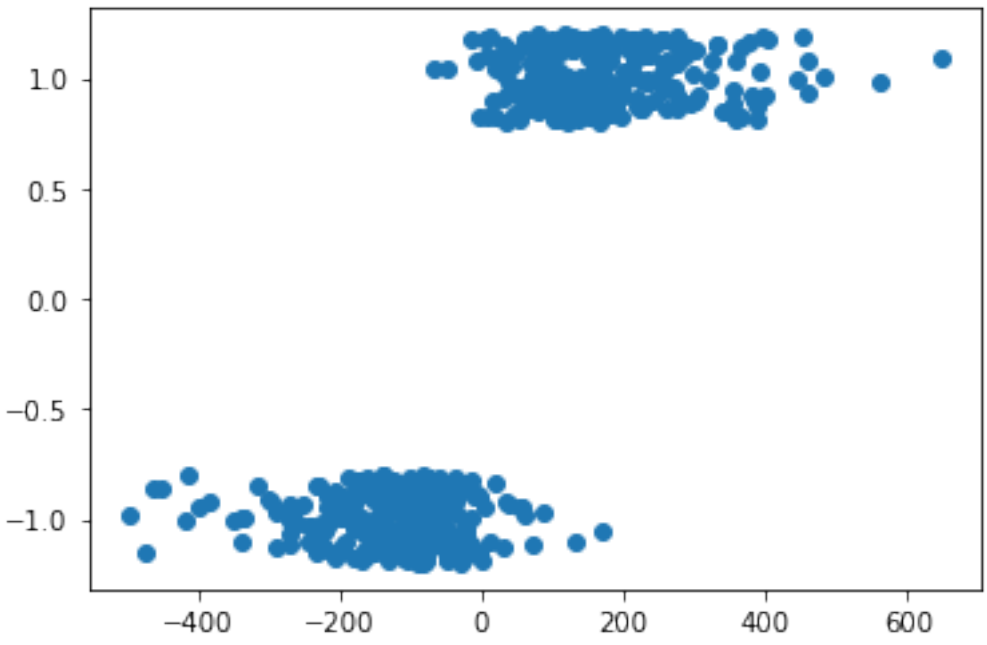
Solution:
Create 21 groups of different sizes, such that score values lie in ranges of 50. This means scores from 0 to 50 will be in 1 group, 50 to 100 in the next and so on. This also includes in negative direction, i.e. -50 to 0 in 1 group, -100 to -50 in next and so on

Plot the graph as follows:



It is evident from the graph that there is a positive correlation between higher magnitude scores and accuracy

Note: Adding a small random variable in y_{test} (this gives a visualization of the density), for each row, and plotting it shows the following (a different way to confirm the same):


1.0
0.5
0.0
−0.5
−1.0
−400
−200
0
200
400
600

13. Optional

14. Show that for $\boldsymbol{w}$ to be a minimizer of $J(\boldsymbol{w})$, we must have $X^TX\boldsymbol{w} + \lambda I\boldsymbol{w} = X^Ty$. Show that the minimizer of $J(\boldsymbol{w})$ is $\boldsymbol{w} = (X^TX + \lambda I)^{-1}X^Ty$. Justify that the matrix $X^TX + \lambda I$ is invertible, for $\lambda > 0$. (You should use properties of positive (semi)definite matrices. If you need a reminder look up the Appendix.)

Solution:

$$J(\boldsymbol{w}) = ||X\boldsymbol{w} - y||^2 + \lambda||\boldsymbol{w}||^2,$$

Differentiate wrt w,

$$\frac{\partial J(w)}{\partial w} = \frac{\partial}{\partial w}\left((Xw - y)^T(Xw - y) + \lambda w^Tw\right)$$
$$\frac{\partial J(w)}{\partial w} = \frac{\partial}{\partial w}\left((w^TX^TXw - w^TX^Ty - y^TXw + y^Ty) + \lambda w^Tw\right)$$
$$\frac{\partial J(w)}{\partial w} = 2X^TXw - 2X^Ty + 2\lambda Iw$$

w is a minimizer of $J(w)$ for that value of w for which the differentiation of $J(w)$ wrt w is 0

So, for w to be a minimizer,

$0 = 2X^TXw - 2X^Ty + 2\lambda Iw$

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

Part 1 proved.

For part 2, simply rewrite the equation as,

$(X^TX + \lambda I)w = X^Ty$

Multiply both sides of the equation by $(XX^T + \lambda I)^{-1}$ (on the left part)

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

Part 2 proved.

We will first check if X^TX is positive semi-definite

For XX^T to be positive semi-definite, xX^TXx should be ≥ 0

Let some $y = X^T x$, so for XX^T to be positive semi-definite, $y^T y$ should be ≥ 0

Now, let

$$y = \begin{matrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & & & \\ y_{m1} & y_{m2} & \cdots & y_{mn} \end{matrix}$$

$$y^T = \begin{matrix} y_{11} & y_{21} & \cdots & y_{n1} \\ y_{12} & y_{22} & \cdots & y_{n2} \\ \vdots & & & \\ y_{1m} & y_{2m} & \cdots & y_{nm} \end{matrix}$$

We see that $y^T y$ will have terms which are sum of squares, and so it is always ≥ 0

So, $X^T X$ is always positive semi-definite

Now, if $\lambda > 0$, the matrix $X^T X + \lambda I$ will be positive definite, as in $X^T X$, the diagonal elements will be strictly > 0

So, the matrix is $X^T X + \lambda I$ invertible.

Part 3 proved.

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

Solution:

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

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

So, $\alpha = \frac{1}{\lambda} y - \frac{1}{\lambda} Xw$

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

Solution:
Here, X^T is matrix with dimensions: [num_features, num_examples]

If the training examples are vectors: [x_1, x_2, ..., x_m], and α is a vector [α_1, α_2, ..., α_n] then $w*$ is a linear combination of the training inputs, as α is over the entire training set.

This means that the minimizer w is "in the span of the data"

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

Solution:

Assuming $\lambda > 0$

$$\alpha = \frac{1}{\lambda}y - \frac{1}{\lambda}Xw$$

As given in hint, replace $\boldsymbol{w}$ by $X^T\boldsymbol{\alpha}$

$$\alpha = \frac{1}{\lambda}y - \frac{1}{\lambda}XX^T\alpha$$

$$\lambda I\alpha = y - XX^T\alpha$$

$$(XX^T + \lambda I)\alpha = y$$

Multiply both sides of the equation on the left by $(XX^T + \lambda I)^{-1}$, (note, we have already proved that if $\lambda > 0$, the matrix is invertible)

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

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

Solution:

$$Xw = XX^T\alpha$$

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

$$= K(K + \lambda I)^{-1}y$$

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

$$k_{\boldsymbol{x}} = \begin{pmatrix} \boldsymbol{x}^T \boldsymbol{x}_1 \\ \vdots \\ \boldsymbol{x}^T \boldsymbol{x}_n \end{pmatrix}$$

to simplify the expression.)

Solution:

$f(x) = x^T w^*$

As $w = X^T \alpha$, so $w^* = X^T \alpha^*$

$f(x) = x^T X^T \alpha^*$

Using the hint, define the column vector

$$k_{\boldsymbol{x}} = \begin{pmatrix} \boldsymbol{x}^T \boldsymbol{x}_1 \\ \vdots \\ \boldsymbol{x}^T \boldsymbol{x}_n \end{pmatrix}$$

$f(x) = k_{\boldsymbol{x}} \alpha^*$

Use value of α^* which was derived from earlier,

$f(x) = k_{\boldsymbol{x}} (X_{train} X_{train}^T + \lambda I)^{-1} y_{train}$

20. 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 $W \in \mathbb{R}^{n_1 \times d}$ and $X \in \mathbb{R}^{n_2 \times d}$ and should return a matrix $M \in \mathbb{R}^{n_1 \times n_2}$ 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`.

Solution:

[2]:
```
### Kernel function generators

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

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

21. Use the linear kernel function defined in the code to compute the kernel matrix on the set of points $x_0 \in_X = \{-4, -1, 0, 2\}$. Include both the code and the output.

    Solution:

[15]:
```
X = np.transpose(np.array([-4, -1, 0, 2])).reshape(-1,1)

print(linear_kernel(X, X))
```

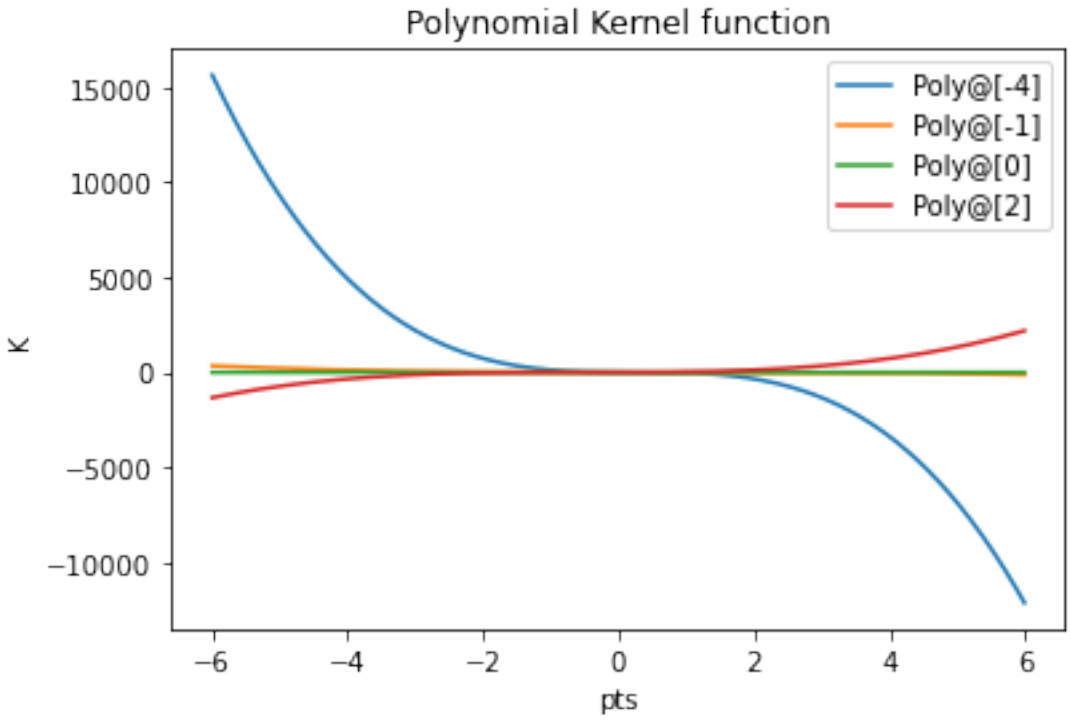
```
[[16  4  0 -8]
 [ 4  1  0 -2]
 [ 0  0  0  0]
 [-8 -2  0  4]]
```

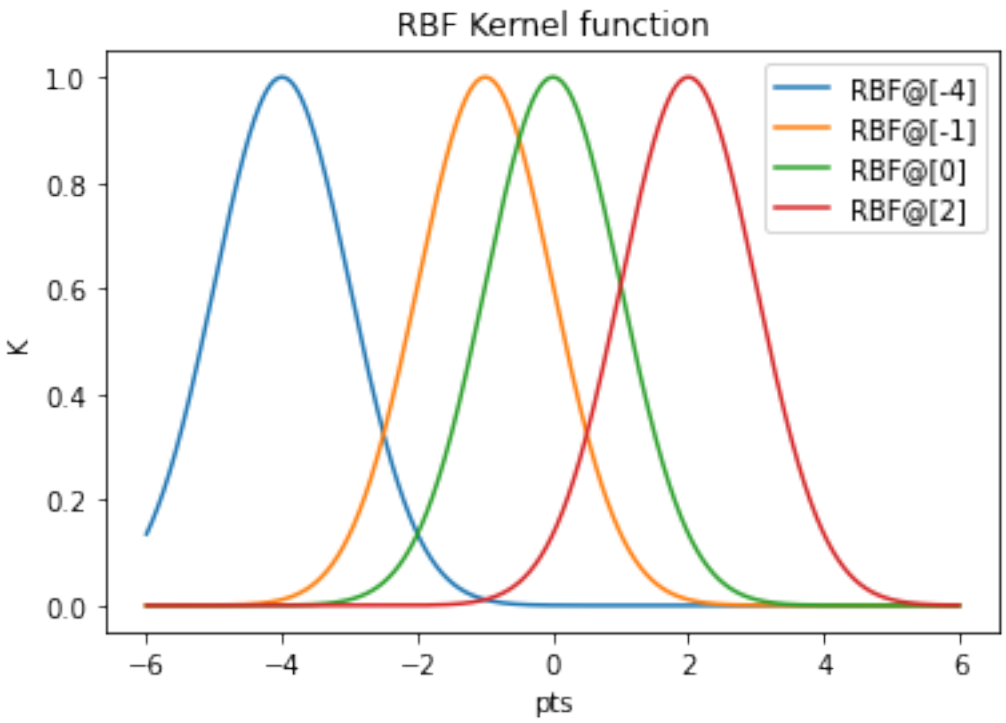
22. Suppose we have the data set $_{X,y} = \{(-4,2),(-1,0),(0,3),(2,5)\}$ (in each set of parentheses, the first number is the value of x_i and the second number the corresponding value of the target y_i). Then by the representer theorem, the final prediction function will be in the span of the functions $x \mapsto k(x_0, x)$ for $x_0 \in_X = \{-4,-1,0,2\}$. This set of functions will look quite different depending on the kernel function we use. The set of functions $x \mapsto k_{\text{linear}}(x_0, x)$ for $x_0 \in_X$ and for $x \in [-6,6]$ has been provided for the linear kernel.

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

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

    Solution:




RBF Kernel function
RBF@[-4]
RBF@[-1]
RBF@[0]
RBF@[2]
K
pts
1.0
0.8
0.6
0.4
0.2
0.0
−6
−4
−2
0
2
4
6

23. 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 $x_1, \ldots, x_n \in$ are the inputs in the training set. We will use the class `Kernel_Machine` in the skeleton code to make prediction with different kernels. Complete the `predict` function of the class `Kernel_Machine`. Construct a `Kernel_Machine` object with the RBF kernel (sigma=1), with prototype points at $-1, 0, 1$ and corresponding weights α_i $1, -1, 1$. Plot the resulting function.

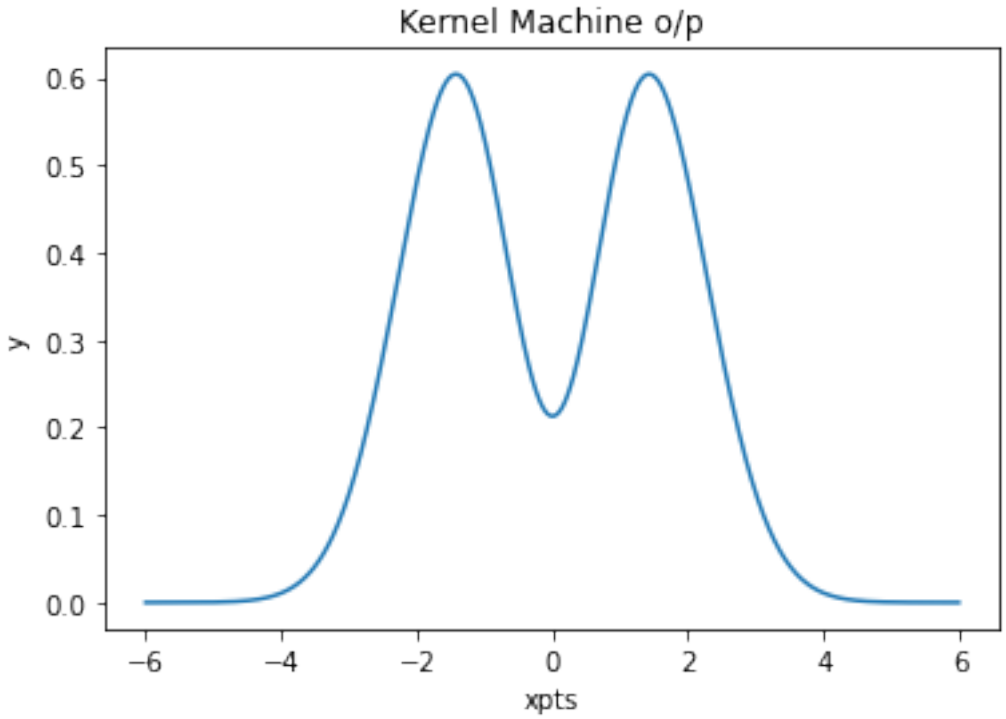
Solution:

[107]:
```
class Kernel_Machine(object):
    def __init__(self, kernel, training_points, weights):
        """
        Args:
            kernel(X1,X2) - a function return the cross-kernel matrix␣
↪between rows of X1 and rows of X2 for kernel k
            training_points - an nxd matrix with rows x_1,..., x_n
            weights - a vector of length n with entries alpha_1,...,alpha_n
        """

        self.kernel = kernel
        self.training_points = training_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 is
                Sum_{i=1}^R alpha_i k(x_j, mu_i)
        """
        v = []
        for X_j in X:
            s=0
            for alpha_i, training_points_i in zip(self.weights, self.
↪training_points):
                s+=alpha_i*kernel(training_points_i.reshape(-1,1), X_j.
↪reshape(-1,1))

            v.append(s[0])
        return v
```

[108]:
```
xpts = np.arange(-6.0, 6, plot_step).reshape(-1,1)
prototypes = np.array([-1,0,1]).reshape(-1,1)
```

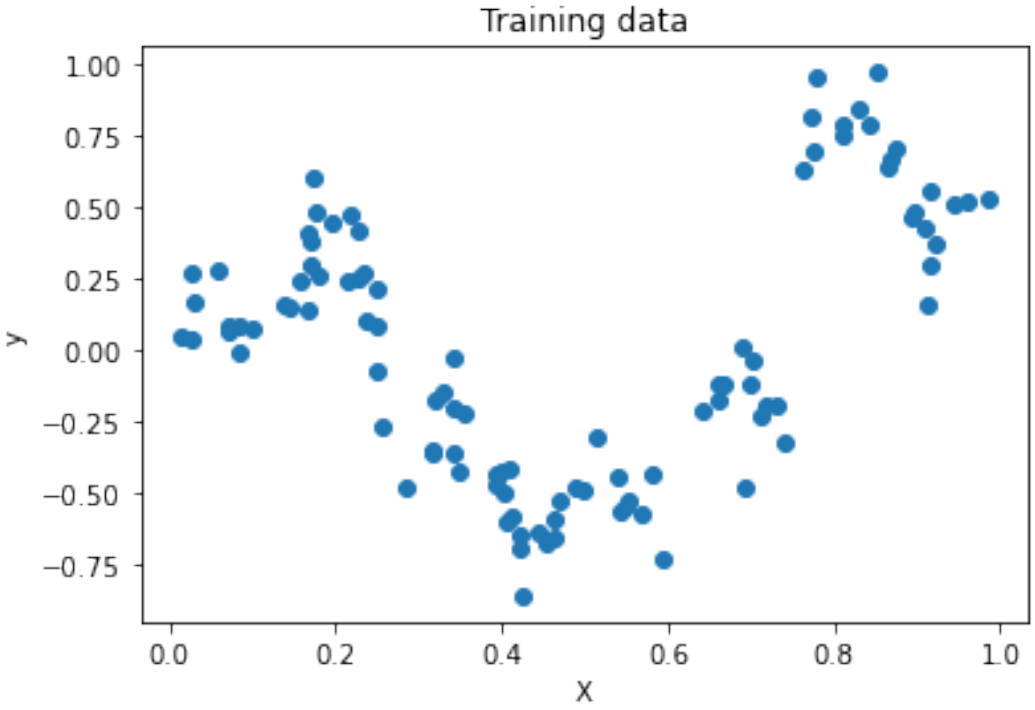
```
# RBF kernel
kernel = functools.partial(RBF_kernel, sigma=1)
weights = np.array([1, -1, 1])

kernel_machine = Kernel_Machine(kernel, prototypes, weights)
```

[109]:
```
output = kernel_machine.predict(xpts)
```



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

Solution:



25. In a previous problem, we showed that in kernelized ridge regression, the final prediction function is $f(\boldsymbol{x}) = \sum_{i=1}^{n} \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x})$, where $\alpha = (\lambda I + K)^{-1} y$ and $K \in \mathbb{R}^{n \times n}$ is the kernel matrix of the training data: $K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$, for $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$. In terms of kernel machines, α_i is the weight on the kernel function evaluated at the training point $\boldsymbol{x}_i$. 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.

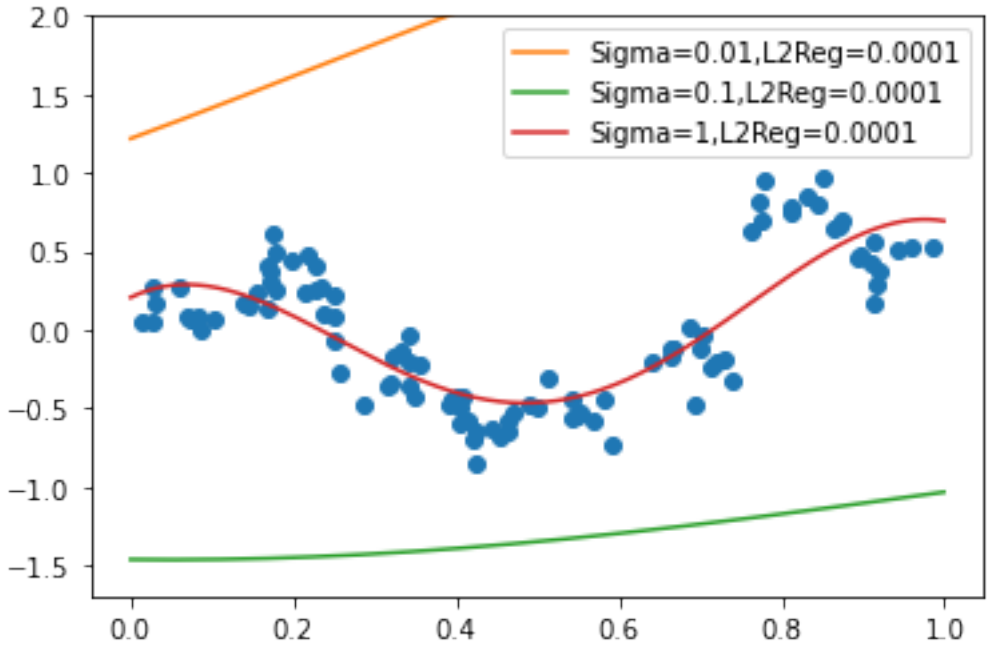
Solution:

[15]:
```
def train_kernel_ridge_regression(X, y, kernel, l2reg):
    alpha = np.matmul( np.linalg.inv(l2reg*np.identity(X.
 ↪shape[0])+kernel(X, X)), y)
    return Kernel_Machine(kernel, X, alpha)
```

26. 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?

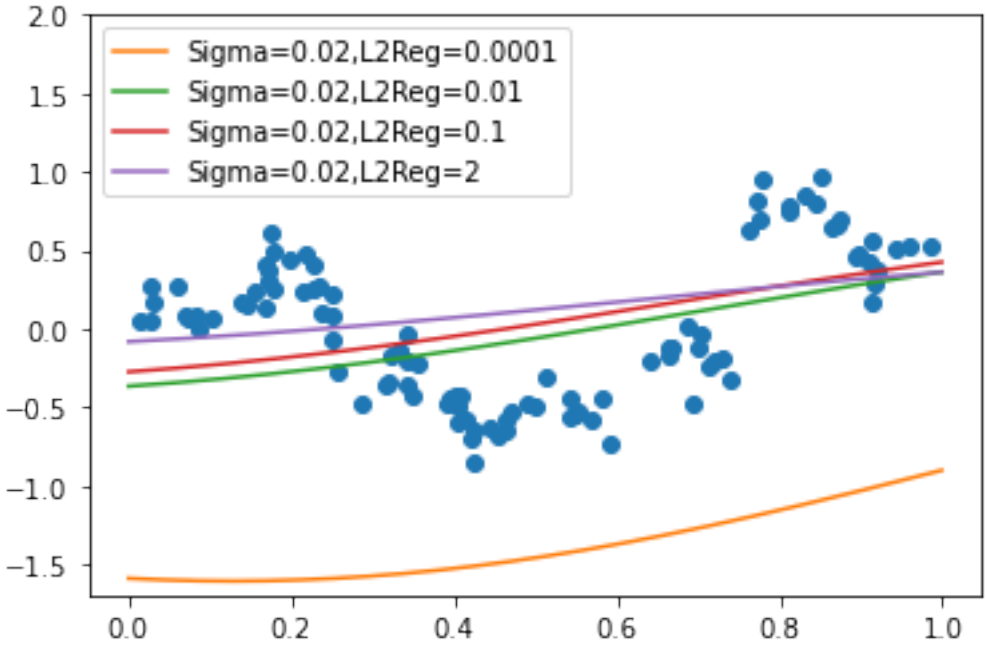
Solution:

Increasing the value of sigma, causes the model to be more likely to over fit, and decreasing sigma causes less over fitting



27. 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 $\lambda \to \infty$?

Solution:
As $\lambda \to \infty$, the prediction becomes a straight line (very simple function, least over fit)



28. 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.

Solution:
First we focus on RBF kernel. Simply running the code given (with no change) gives the following output:

RBF kernel values:

	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_train_score
0	-	RBF	32.0000	-	0.1	1.648971e-01	2.074632e-01
1	-	RBF	32.0000	-	1	1.627928e-01	2.056596e-01
2	-	RBF	32.0000	-	10	1.622715e-01	2.054899e-01
3	-	RBF	16.0000	-	0.1	1.645465e-01	2.068072e-01
4	-	RBF	16.0000	-	1	1.592047e-01	2.019913e-01
5	-	RBF	16.0000	-	10	1.593943e-01	2.028953e-01
6	-	RBF	8.0000	-	0.1	1.649354e-01	2.069001e-01
7	-	RBF	8.0000	-	1	1.542559e-01	1.969124e-01
8	-	RBF	8.0000	-	10	1.630288e-01	2.068375e-01
9	-	RBF	4.0000	-	0.1	1.656908e-01	2.074293e-01
10	-	RBF	4.0000	-	1	1.476088e-01	1.901032e-01
11	-	RBF	4.0000	-	10	2.075878e-01	2.509341e-01
12	-	RBF	2.0000	-	0.1	1.677423e-01	2.092091e-01
13	-	RBF	2.0000	-	1	1.379683e-01	1.803044e-01
14	-	RBF	2.0000	-	10	4.419941e-01	4.803429e-01
15	-	RBF	1.0000	-	0.1	1.748776e-01	2.157998e-01
16	-	RBF	1.0000	-	1	1.235387e-01	1.656641e-01
17	-	RBF	1.0000	-	10	1.462745e+00	1.476919e+00
18	-	RBF	0.5000	-	0.1	1.953929e-01	2.353589e-01
19	-	RBF	0.5000	-	1	1.043390e-01	1.459140e-01
20	-	RBF	0.5000	-	10	5.518899e+00	5.442224e+00
21	-	RBF	0.2500	-	0.1	2.398297e-01	2.785454e-01
22	-	RBF	0.2500	-	1	8.452570e-02	1.247461e-01
23	-	RBF	0.2500	-	10	2.061682e+01	2.026899e+01
24	-	RBF	0.1250	-	0.1	3.049516e-01	3.425032e-01
25	-	RBF	0.1250	-	1	7.025240e-02	1.081655e-01
26	-	RBF	0.1250	-	10	7.430498e+01	7.340419e+01
27	-	RBF	0.0625	-	0.1	3.534025e-01	3.903477e-01
28	-	RBF	0.0625	-	1	6.363177e-02	9.884273e-02
29	-	RBF	0.0625	-	10	2.641754e+02	2.631076e+02

The least test error is achived when $signma = 1$ and $\lambda = 0.0625$. So we zoom in around this range

	param_kernel	param_l2reg	param_sigma	mean_test_score	mean_train_score
0	RBF	0.031250	0.9	0.414674	0.430876
1	RBF	0.031250	1.0	0.061963	0.094808
2	RBF	0.031250	1.1	0.756722	0.848047
3	RBF	0.037163	0.9	0.407629	0.424306

Small change in sigma causes huge change in the error, so we fix sigma at 1

For best l2reg, zoom in on 0.03125

4	RBF	0.022097	1	0.061933	0.093822
5	RBF	0.026278	1	0.061914	0.094262
6	RBF	0.031250	1	0.061963	0.094808

Best l2reg is 0.026278 and best sigma is 1

Now we look at Polynomial kernel. Same as earlier, simply running the code given (with no change) gives the following output:

Polynomial kernel values:

30	2	polynomial	10.0000	-1	-	4.478202e-01	5.029079e-01
31	2	polynomial	10.0000	0	-	4.660597e-01	5.194659e-01
32	2	polynomial	10.0000	1	-	1.863629e-01	2.322696e-01
33	2	polynomial	0.1000	-1	-	5.725288e-01	6.200173e-01
34	2	polynomial	0.1000	0	-	7.901373e+03	7.945967e+03
35	2	polynomial	0.1000	1	-	3.772447e+00	3.931724e+00
36	2	polynomial	0.0100	-1	-	2.372731e+00	2.508005e+00
37	2	polynomial	0.0100	0	-	7.986320e+05	8.030104e+05
38	2	polynomial	0.0100	1	-	1.118178e+01	1.154418e+01
39	3	polynomial	10.0000	-1	-	1.546087e-01	1.962359e-01
40	3	polynomial	10.0000	0	-	4.639152e-01	5.174766e-01
41	3	polynomial	10.0000	1	-	1.843632e-01	2.300383e-01
42	3	polynomial	0.1000	-1	-	6.314149e-01	6.773705e-01
43	3	polynomial	0.1000	0	-	1.057247e+04	1.064058e+04
44	3	polynomial	0.1000	1	-	4.078107e-01	4.609754e-01
45	3	polynomial	0.0100	-1	-	2.493726e+00	2.618036e+00
46	3	polynomial	0.0100	0	-	1.073569e+06	1.080338e+06
47	3	polynomial	0.0100	1	-	1.773526e+00	1.890207e+00
48	4	polynomial	10.0000	-1	-	1.818061e-01	2.218471e-01
49	4	polynomial	10.0000	0	-	3.964837e-01	4.486164e-01
50	4	polynomial	10.0000	1	-	1.776609e-01	2.224695e-01
51	4	polynomial	0.1000	-1	-	3.170284e+00	3.230240e+00
52	4	polynomial	0.1000	0	-	1.097670e+04	1.105096e+04
53	4	polynomial	0.1000	1	-	2.361113e-01	2.789282e-01
54	4	polynomial	0.0100	-1	-	1.843381e+00	1.891255e+00
55	4	polynomial	0.0100	0	-	1.120269e+06	1.127701e+06
56	4	polynomial	0.0100	1	-	7.440426e-01	8.051115e-01

Best degree=3, lambda=10, offset=-1

So, zooming in on these values, we get
Now we get degree=3, lambda=9, offset=-1.5 to be the best values

	param_offset	param_degree	param_kernel	param_l2reg	mean_test_score	mean_train_score
0	-1.5	2	polynomial	9	0.197788	0.244337
1	-1.0	2	polynomial	9	0.318654	0.370039
2	-0.5	2	polynomial	9	0.216505	0.263740
3	-1.5	2	polynomial	10	0.200389	0.246993
4	-1.0	2	polynomial	10	0.447820	0.502908
5	-0.5	2	polynomial	10	0.193461	0.239632
6	-1.5	2	polynomial	11	0.204206	0.250932
7	-1.0	2	polynomial	11	0.903456	0.968935
8	-0.5	2	polynomial	11	0.180899	0.226364
9	-1.5	3	polynomial	9	0.152589	0.195772
10	-1.0	3	polynomial	9	0.153947	0.195382
11	-0.5	3	polynomial	9	0.153016	0.194880
12	-1.5	3	polynomial	10	0.153492	0.196638
13	-1.0	3	polynomial	10	0.154609	0.196236
14	-0.5	3	polynomial	10	0.153715	0.195738
15	-1.5	3	polynomial	11	0.154411	0.197522
16	-1.0	3	polynomial	11	0.155327	0.197097
17	-0.5	3	polynomial	11	0.154418	0.196566
18	-1.5	4	polynomial	9	0.158515	0.201671
19	-1.0	4	polynomial	9	0.175478	0.215721
20	-0.5	4	polynomial	9	0.155663	0.198957
21	-1.5	4	polynomial	10	0.159072	0.202235
22	-1.0	4	polynomial	10	0.181806	0.221847
23	-0.5	4	polynomial	10	0.155342	0.198538
24	-1.5	4	polynomial	11	0.159577	0.202747
25	-1.0	4	polynomial	11	0.195524	0.235127
26	-0.5	4	polynomial	11	0.155470	0.198597

Now we look at linear kernel. There is only 1 parameter lambda, so we will change the code given to have more values in table:
Linear kernel values:

	param_kernel	param_l2reg	mean_test_score	mean_train_score
0	linear	100.0	0.166070	0.209192
1	linear	30.0	0.168619	0.212907
2	linear	10.0	0.303559	0.352903
3	linear	3.0	2.665810	2.739310
4	linear	1.0	26.176560	26.360742
5	linear	0.3	304.515779	305.563245
6	linear	0.1	2780.395034	2787.984770

Test score minimum at value 100, zoom in on 100.

	param_kernel	param_l2reg	mean_test_score	mean_train_score
5	linear	50	0.165336	0.208872
4	linear	100	0.166070	0.209192
3	linear	150	0.166667	0.209696
2	linear	200	0.166999	0.209994
1	linear	250	0.167204	0.210182
0	linear	300	0.167342	0.210312

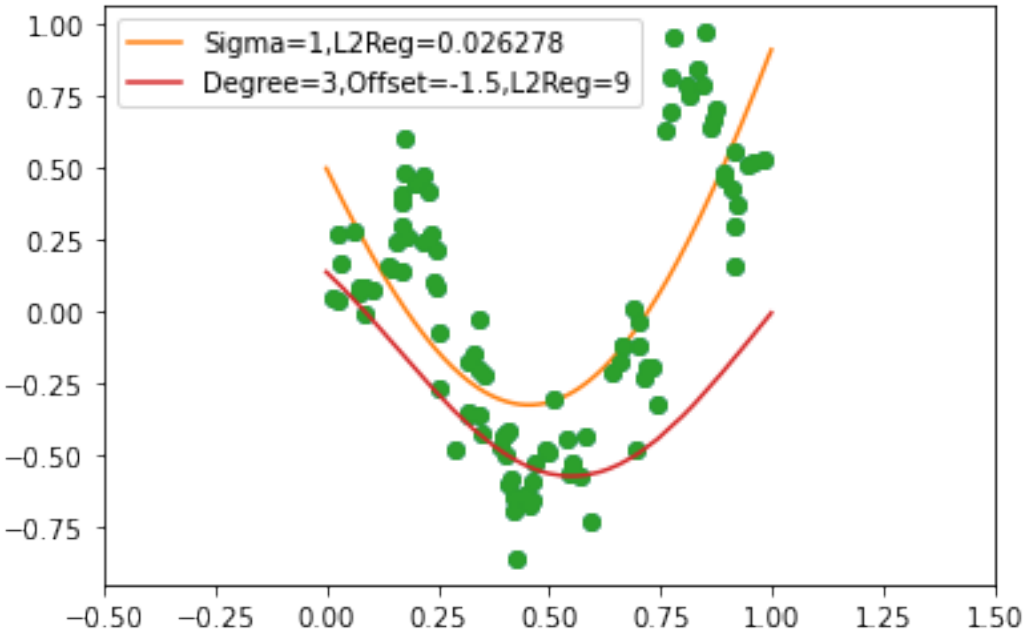
Zoom in further around range 50 to 100

2	50	linear	0.165336	0.208872
3	55	linear	0.165309	0.208758
4	60	linear	0.165348	0.208728

Best value at: 55

29. 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.

Solution:
Comment: RBF kernel is better, as it covers more points and has lower error



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

Solution:
The final point is $(x, f(x) + \epsilon)$. This means $y = f(x) + \epsilon$

$\hat{y}$ is the predicted value from the Bayes predictor

Bayes predictor would be $f(x)+$ expected value of ϵ, as ϵ is a random variable

Expected value is the mean. Given mean is 0 for , so bayes predictor is:
f(x)

Bayes risk is: $(\hat{y} - y)^2$

$= (f(x) - f(x) - \epsilon)^2$

$= \epsilon^2$

ϵ is a random variable, so ϵ^2 is also a random variable

Bayes risk is the expected value of ϵ^2

Using the formula for variance of a random variable:

Variance = $E(\epsilon^2) - (E(\epsilon))^2$

Given variance is 0.01, and $E(\epsilon)$ is the mean, which is 0.

So, $E(\epsilon^2) = 0.01$

Bayes risk is 0.01