Subgradients

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

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

There may be 0, 1, or infinitely many subgradients at any point. The *subdifferential* of f at a point $\boldsymbol{x}$, denoted $\partial f(\boldsymbol{x})$, is the set of all subgradients of f at $\boldsymbol{x}$. A good reference for subgradients are the course notes on Subgradients by Boyd et al. Below we derive a property that will make our life easier for finding a subgradient of the hinge loss.

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 $g \in \partial f(\boldsymbol{x})$.

(1)

Show: $f(z) \geq f(x) + g^T(z-x)$ for all z where $g \in \mathbb{R}^d$ is a subgradient of f

Given: $f_1, \cdots, f_m : \mathbb{R}^d \to \mathbb{R}$ and convex

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

1) Since $f(x) = f_k(x)$ and $f(z) = \max_{i=1,\ldots,m} f_i(z)$,

$f(z) = f_k(z)$ or $f(z) > f_k(z)$

$\hookrightarrow f(z) \geq f_k(z)$

2) By the definition of a subgradient, for $g \in \partial f_k(x)$, $f_k(z) \geq f_k(x) + g^T(z-x)$

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

$f(z) \geq f_k(x) + g^T(z-x) \longrightarrow g \in \partial f(x)$

(2)

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

if $1 - y_i w^T x_i \leq 0$, $J(w) = 0$ $\quad g = 0 \in \partial_w J$

if $1 - y_i w^T x_i > 0$, $J(w) = 1 - y_i w^T x_i$ $\quad g = -y_i x_i \in \partial_w J$

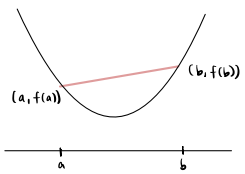
$$g(w) = \begin{cases} 0 & \text{if } 1 - y_i w^T x_i \leq 0 \\ -y_i x_i & \text{otherwise} \end{cases}$$

(3)

3. Suppose we have function $f : \mathbb{R}^n \to \mathbb{R}$ which is sub-differentiable everywhere, i.e. $\partial f \neq \emptyset$ for all $x \in \mathbb{R}^n$. Show that f is convex. Note, in the general case, a function is convex if for all x, y in the domain of f and for all $\theta \in (0, 1)$,

$$\theta f(a) + (1-\theta) f(b) \geq f(\theta a + (1-\theta)(b))$$

Hint: Suppose f is not convex, then by definition, there exists a point in some interval: $x \in (a, b)$, such that $f(x_0)$ lies above the line connection $(a, f(a)), (b, f(b))$. Is this possible if the function s sub-differentiable everywhere?



$\partial f \neq 0$ for $\theta \in (0, 1)$

$f : \mathbb{R}^n \to \mathbb{R}$ and sub-differentiable everywhere $\longrightarrow$ f has $g(x)$ for all x

$\theta f(a) + (1-\theta) f(b) \geq f(\theta a + (1-\theta)(b))$

$\theta f(a) + f(b) - \theta f(b) \geq f(\theta a + b - \theta b)$

$\theta(f(a) - f(b)) + f(b) \geq f(\theta(a-b) + b)$

$f(a) - f(b) \geq \dfrac{f(b + \theta(a-b)) - f(b)}{\theta}$

As $\theta \downarrow$, $f(a) - f(b) \geq \nabla f^T(b)(a-b)$

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

4

4. Consider the SVM objective function for a single training point[2]: $J_i(\boldsymbol{w}) = \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \max\{0, 1 - y_i \boldsymbol{w}^T \boldsymbol{x}_i\}$. 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.

hinge loss $\ell(m) = \max(0, 1-m)$ where $m = y_i w^T x_i$

$\nabla_w \ell(m) = \nabla_w \ell(y_i w^T x_i)$

$= \ell'(y_i w^T x_i) y_i x_i$

$= \begin{cases} 0 & y_i w^T x_i > 1 \\ -y_i x_i & y_i w^T x_i < 1 \\ \text{undefined} & y_i w^T x_i = 1 \end{cases}$

$J_i(w) = \frac{\lambda}{2}\|w\|^2 + \max\{0, 1 - y_i w^T x_i\}$

if $1 - y_i w^T x_i < 0$, $J_i(w) = \frac{\lambda}{2}\|w\|^2 + 0 \longrightarrow \nabla_w J_i(w) = \lambda w$

if $1 - y_i w^T x_i > 0$, $J_i(w) = \frac{\lambda}{2}\|w\|^2 + (1 - y_i w^T x_i) \longrightarrow \nabla_w J_i(w) = \lambda w - y_i x_i$

$\nabla_w J_i(w) = \partial J_i(w) = \partial\left(\frac{\lambda}{2}\|w\|^2\right) + \partial \max\{0, 1 - y_i w^T x_i\} = \begin{cases} \lambda w & \text{if } y_i w^T x_i > 1 \\ \lambda w - y_i x_i & \text{if } y_i w^T x_i < 1 \\ \text{undefined} & \text{if } y_i w^T x_i = 1 \end{cases}$

$= \ell'(y_i w^T x_i) y_i x_i$

5

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

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

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.)

$J_i(w) = \frac{\lambda}{2}\|w\|^2 + \max\{0, 1 - y_i w^T x_i\}$

$\partial \frac{\lambda}{2}\|w\|^2 = \lambda w$

$\partial \max\{0, 1 - y_i w^T x_i\} = \begin{cases} -y_i x_i & \text{if } y_i w^T x_i < 1 \\ 0 & \text{if } y_i w^T x_i \geq 1 \end{cases}$

$\partial J_i(w) = \partial \frac{\lambda}{2}\|w\|^2 + \partial \max\{0, 1 - y_i w^T x_i\} = \begin{cases} \lambda w - y_i x_i & \text{for } y_i w^T x_i < 1 \\ \lambda w & \text{for } y_i w^T x_i \geq 1 \end{cases} = g_w$

$g_w \in \partial J_i(w) \longrightarrow$ Subgradient of $J_i(w) = \nabla_w J_i(w) = g_w$

6

Q6. Write a function that converts an example (a list of words) into a sparse bag-of-words representation.

You may find Python's Counter3 class to be useful here. Note that a Counter is itself a dictionary.

```
from collections import Counter
```

```
def bag_of_words(review):
    word_dict = []
    for x in review:
        word_dict.append(Counter(x))
    return word_dict
```

```
review = load_and_shuffle_data()
```

```
review_class = []
for i in range(len(review)):
    review_class.append(review[i][-1])
    del review[i][-1]

wordCounts = bag_of_words(review)
```

7

Q7. 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.

```
train_X = wordCounts[:1500]
test_X =  wordCounts[1500:2000]
train_y = review_class[:1500]
test_y = review_class[1500:2000]
```

8

8. 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.

```
import time
```

```
def pegasos(x, y, lambda_reg=1, iteration=100):
    start = time.time()
    w = {}
    t = 0
    num_instances = len(x)
    for i in range(iteration):
        for j in range(num_instances):
            t += 1
            eta = 1 / (lambda_reg * t)
            increment(w, -eta*lambda_reg, w)
            if y[j] * dotProduct(w, x[j]) < 1:
                increment(w, eta*y[j], x[j])
        i += 1
        end = time.time()
    print('Total Time: {}'.format(end - start))
    return w
```

9

9. 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. [4]

$$w_{t+1} = (1 - \eta_t \lambda) w_t + \eta_t y_j x_j$$

$$W = s \cdot W \longrightarrow W_t = s_t W_t$$

$$w_{t+1} = (1 - \eta_t \lambda) w_t + \eta_t y_j x_j$$

$$W_{t+1} = \frac{1}{s_{t+1}} (1 - \eta_t \lambda) w_t + \eta_t y_j x_j$$

$$W_{t+1} = W_t + \frac{1}{s_{t+1}} \eta_t y_j x_j$$

```
def pegasos2(x, y, lambda_reg=1, iteration=100):
    start = time.time()
    t = 1
    s = 1
    W = {}
    num_instances = len(x)
    for i in range(iteration):
        for j in range(num_instances):
            t += 1
            eta = 1 / (lambda_reg * t)
            s = (1 - eta * lambda_reg)*s
            if (y[j] * dotProduct(W, x[j])) < 1/s:
                increment(W, 1/s*eta*y[j], x[j])
        w = {}
        increment(w, s, W)
        i += 1
        end = time.time()
    print('Total Time: {}'.format(end - start))
    return w
```

10

10. 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.

```
pegasos(train_X, train_y, lambda_reg = 1, iteration = 5)
```

```
Total Time: 138.7857894897461

{'saving': 0.00013333333333333207,
 'silverman': -0.011200000000000024,
 'is': 0.01480000000000006,
 'a': 0.01360000000000002,
 'good': 0.012000000000000056,
 'example': -0.007599999999999965,
 'of': 0.00933333333333329,
 'comedy': 0.003999999999999986,
 'gone': -0.007466666666666671,
 'bad': -0.12453333333333307,
 'as': 0.024399999999999988,
 'love': 0.01746666666666663,
 'story': 0.012533333333333342,
 'it': 0.01719999999999995,
 'however': 0.00333333333333333,
 'falls': -0.02386666666666657,
 'flat': -0.019999999999999966,
 'on': -0.04786666666666662
```

```
pegasos2(train_X, train_y, lambda_reg = 1, iteration = 5)
```

```
Total Time: 1.5059928894042969

{'saving': 0.00013333155579256136,
 'silverman': -0.011198506865751185,
 'is': 0.014798026929742622,
 'a': 0.013598186908412163,
 'good': 0.011998400213304835,
 'example': -0.007598986801759739,
 'of': 0.009332089054792634,
 'comedy': 0.003999466737768281,
 'gone': -0.007465671243834124,
 'bad': -0.12451673110251928,
 'as': 0.02439674710038649,
 'love': 0.017464338088254823,
 'story': 0.012531662445007242,
 'it': 0.017197706972403585,
 'however': 0.0033328889481402365,
 'falls': -0.023863484868684073,
 'flat': -0.0199973336888414,
 'on': -0.04786028529529373
```

Q8. Approach (slower)

Time = 138.79 s.

Q9. Approach (faster)

Time = 1.506 s.

Both approaches essentially give the same result.

11

11. 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 $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$.

```
def classification_error(x, y, w):
    num_instances = len(x)
    loss = 0
    for i in range(num_instances):
        prediction = dotProduct(w, x[i])
        if(prediction * y[i] < 0):
            loss += 1
    loss = loss / num_instances
    return loss
```

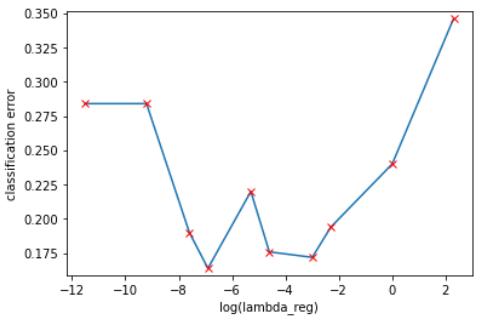
```
w = pegasos2(train_X, train_y, lambda_reg = 1, iteration = 10)
loss = classification_error(train_X, train_y, w)
print(loss)
```

```
Total Time: 4.096717596054077
0.12866666666666668
```

12

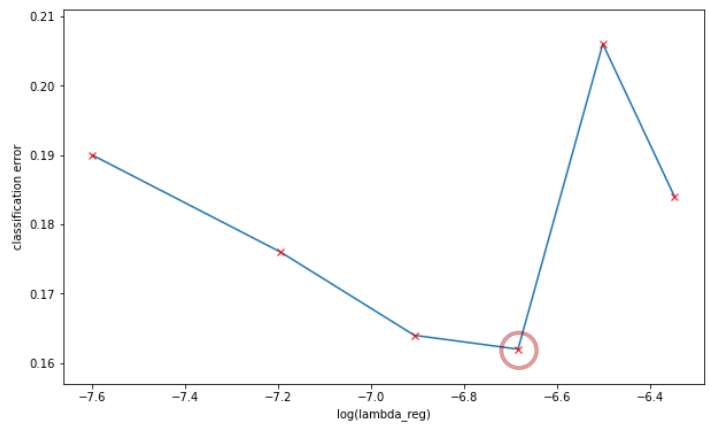
12. Search for the regularization parameter that gives the minimal percent error on your test set. You should now use your faster Pegasos implementation, and run it to convergence. A good search strategy is to start with a set of regularization parameters spanning a broad range of orders of magnitude. 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.)

```
import matplotlib.pyplot as plt

lambdas_regs = [0.00001, 0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 1, 10]

w_list = []
for lambda_reg in lambdas_regs:
    w_list.append(pegasos2(train_X, train_y, lambda_reg = lambda_reg, iteration = 10))

losses = []
for w in w_list:
    loss = classification_error(test_X, test_y, w)
    losses.append(loss)

plt.plot(np.log(lambdas_regs),losses)
plt.plot(np.log(lambdas_regs),losses, 'rx')
plt.xlabel('log(lambda_reg)')
plt.ylabel('classification error')
plt.ylim(min(losses)-0.005, max(losses)+0.005)
plt.show()

print('lambda_regs:', lambdas_regs)
print('losses:', losses)
print('lambda that minimizes classification error is ', lambdas_regs[np.argmin(losses)])
```




```
lambda_regs: [1e-05, 0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 1, 10]
losses: [0.284, 0.284, 0.19, 0.164, 0.22, 0.176, 0.172, 0.194, 0.24, 0.346]
lambda that minimizes classification error is  0.001
```

```
lambdas_regs = np.arange(0.0005, 0.002, 0.00025)
```




```
lambda_regs: [0.0005  0.00075 0.001   0.00125 0.0015  0.00175]
losses: [0.19, 0.176, 0.164, 0.162, 0.206, 0.184]
lambda that minimizes classification error is  0.00125
```

Lambda = 0.00125

Error = 0.162

13

Error Analysis

Recall that the *score* is the value of the prediction $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$. 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.

13. 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?

14. (Optional) Choose an input example $\boldsymbol{x} = (x_1, \ldots, x_d) \in \mathbb{R}^d$ 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. Can you think of new features that might help fix a problem? (Think of making groups of words.)

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

for $\lambda > 0$.

15. 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.)

$$X = \begin{pmatrix} -\boldsymbol{x}_1- \\ \vdots \\ -\boldsymbol{x}_n- \end{pmatrix}.$$

15

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

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

$$J(w) = (Xw - y)^T(Xw - y) + \lambda w^Tw$$

$$= w^TX^TXw - 2y^TXw + y^Ty + \lambda w^Tw$$

$$\nabla_w J(w) = 2X^TXw - 2X^Ty + 2\lambda w = 2(X^TXw - X^Ty + \lambda w)$$

Set $\nabla_w J(w) = 0$ to find w that minimizes $J(w)$

$$0 = X^TXw - X^Ty + \lambda w = w(X^TX + \lambda) - X^Ty$$

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

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

$$x^TMx \geq 0.$$

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).

Justify that the matrix $X^TX + \lambda I$ is invertible

Theorem

The following conditions are each necessary and sufficient for a symmetric matrix M to be positive semidefinite:

- *M can be factorized as $M = R^TR$, for some matrix R.*
- *All eigenvalues of M are greater than or equal to 0.*

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

16

$$\frac{1}{\lambda}\left(X^TXw + \lambda Iw = X^Ty\right)$$

$$\frac{1}{\lambda}X^TXw + Iw = \frac{1}{\lambda}X^Ty$$

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

$$= X^T\underbrace{\frac{1}{\lambda}(y - Xw)}_{\alpha}$$

$$= X^T\alpha$$

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

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

17

$$w = X^T\alpha = \sum_{i=1}^{n}\alpha_i x_i$$

w is the linear combination of the vector x_i at each input data point which indicates that w is in the span of the data

$$w \in \text{span}(x_1, \ldots, x_n)$$

18

18. 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 $\boldsymbol{\alpha}$.)

From Q16: $\alpha = \frac{1}{\lambda}(y - Xw)$

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

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

$$\lambda\alpha + XX^T\alpha = y$$

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

19

19. 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.)

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

20

20. 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.)

$$f(x) = x^T w^* = \hat{y} \qquad k_x = \begin{pmatrix} x^Tx_1 \\ \vdots \\ x^Tx_n \end{pmatrix} =$$

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

21

21. Write functions that compute the RBF kernel $k_{\mathrm{RBF}(\sigma)}(x, x') = \exp\left(-\|x - x'\|^2 / \left(2\sigma^2\right)\right)$ and the polynomial kernel $k_{\mathrm{poly}(a,d)}(x, x') = (a + \langle x, x' \rangle)^d$. The linear kernel $k_{\mathrm{linear}}(x, x') = \langle x, x' \rangle$, 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`.

```
import numpy as np
import matplotlib.pyplot as plt
import sklearn
from scipy.spatial.distance import cdist
import functools

%matplotlib inline
```

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

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

22

Q22. Use the linear kernel function defined in the code to compute the kernel matrix on the set of points x0 ∈ DX = {−4, −1, 0, 2}. Include both the code and the output.

```
x = [-4,-1,0,2]
kernel_matrix = np.zeros((len(x), len(x)))
for i in range(len(x)):
    for j in range(len(x)):
        kernel_matrix[i,j] = linear_kernel(x[i], x[j])
print(kernel_matrix)
```

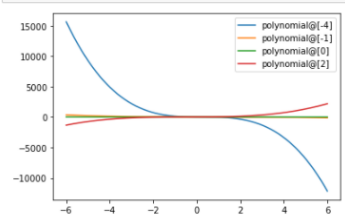
```
[[16.  4.  0. -8.]
 [ 4.  1.  0. -2.]
 [ 0.  0.  0.  0.]
 [-8. -2.  0.  4.]]
```

23

Q23. Suppose we have the data set D_X,y = {(−4, 2),(−1, 0),(0, 3),(2, 5)} (in each set of parentheses, the first number is the value of xi and the second number the corresponding value of the target yi). Then by the representer theorem, the final prediction function will be in the span of the functions x 7→ k(x0, x) for x0 ∈ D_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 7→ klinear(x̄0, x) for x0 ∈ X and for x ∈ [−6, 6] has been provided for the linear kernel.

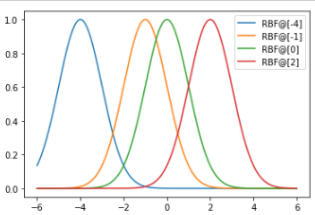
(a) Plot the set of functions x → kpoly(1,3)(x0, x) for x0 ∈ DX and for x ∈ [−6, 6].

```
xpts = np.arange(-6.0, 6, plot_step).reshape(-1,1)
prototypes = np.array([-4,-1,0,2]).reshape(-1,1)
offset = 1
degree = 3

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

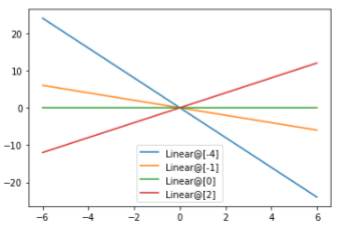


(b) Plot the set of functions x → kRBF(1)(x0, x) for x0 ∈ X and for x ∈ [−6, 6]

```
plot_step = .01
sigma = 1
xpts = np.arange(-6.0, 6, plot_step).reshape(-1,1)
prototypes = np.array([-4,-1,0,2]).reshape(-1,1)

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




```
plot_step = .01
xpts = np.arange(-6.0, 6, plot_step).reshape(-1,1)
prototypes = np.array([-4,-1,0,2]).reshape(-1,1)

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



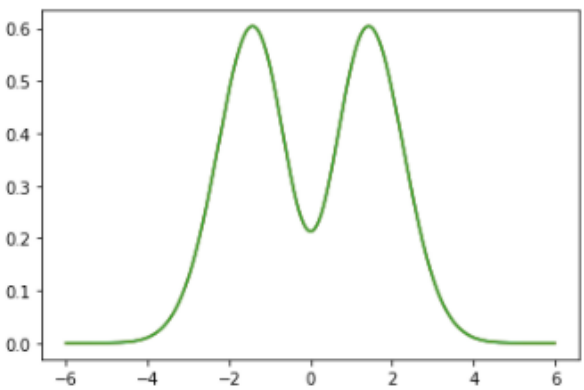
24

24. 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. 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.

Q24. 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 αi 1, −1, 1. Plot the resulting function.

```
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)
        """
        # TODO
        pred = np.dot(self.weights.T, self.kernel(self.training_points, X))
        return pred.T
```

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

for i in range(len(prototypes)):
    k = functools.partial(RBF_kernel, sigma=sigma)
    f = Kernel_Machine(k, prototypes, weights)
    plt.plot(xpts, f.predict(xpts))
plt.show()
```



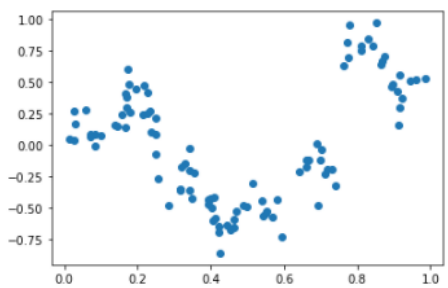
25

Q25. 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.plot(x_train,y_train,'o')
```

```
[<matplotlib.lines.Line2D at 0x180da6612b0>]
```



26

26. 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.

Q26. 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
    N = X.shape[0]
    alpha = np.dot(np.linalg.inv(l2reg * np.identity(N) + kernel(X, X)), y)
    return Kernel_Machine(kernel, X, alpha)
```

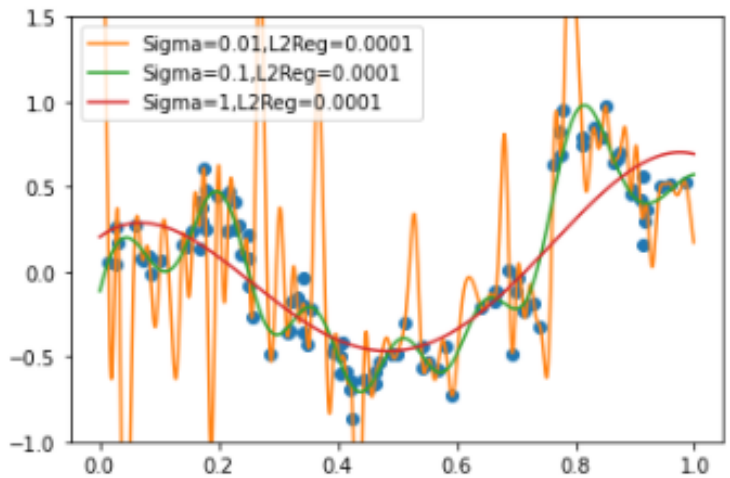
27

27. 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?

Sigma = 0.01 would be more likely to overfit and
Sigma = 1 would be less likely to overfit.

Q27. 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?

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

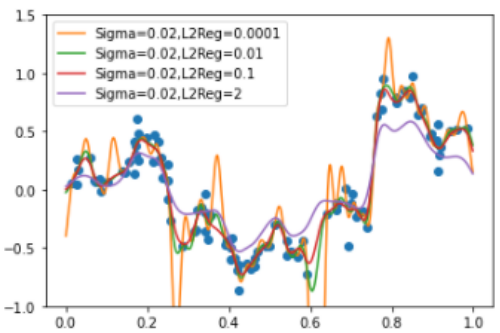


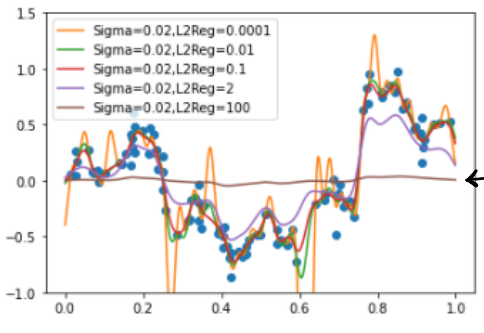
28

28. 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 $\lambda \to \infty$?

As $\lambda \to \infty$, the prediction converges to $y = 0$. The best prediction that can be made is 0 for all values of x when $\lambda \to \infty$.

```
plot_step = .001
xpts = np.arange(0 , 1, plot_step).reshape(-1,1)
plt.plot(x_train,y_train,'o')
sigma= .02
for l2reg in [.0001,.01,.1,2]:
    k = functools.partial(RBF_kernel, sigma=sigma)
    f = train_kernel_ridge_regression(x_train, y_train, k, l2reg=l2reg)
    label = "Sigma="+str(sigma)+",L2Reg="+str(l2reg)
    plt.plot(xpts, f.predict(xpts), label=label)
plt.legend(loc = 'best')
plt.ylim(-1,1.5)
plt.show()
```





$\lambda = 100$

29

29. 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 Kernel

```
param_grid = [{'kernel':['linear'],'l2reg': [10,5,4.5,4,3.5,3,1,0,.01]}]
kernel_ridge_regression_estimator = KernelRidgeRegression()
grid = GridSearchCV(kernel_ridge_regression_estimator,
                    param_grid,
                    cv = predefined_split,
                    scoring = make_scorer(mean_squared_error,greater_is_better = False),
                    return_train_score=True
                   )
grid.fit(np.vstack((x_train,x_test)),np.vstack((y_train,y_test)))
```

```
GridSearchCV(cv=PredefinedSplit(test_fold=array([-1, -1, ...,  0,  0])),
             estimator=KernelRidgeRegression(),
             param_grid=[{'kernel': ['linear'],
                          'l2reg': [10, 5, 4.5, 4, 3.5, 3, 1, 0, 0.01]}],
             return_train_score=True,
             scoring=make_scorer(mean_squared_error, greater_is_better=False))
```

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

	param_kernel	param_l2reg	mean_test_score	mean_train_score
3	linear	4.00	0.164510	0.206563
4	linear	3.50	0.164510	0.206550
2	linear	4.50	0.164511	0.206577
5	linear	3.00	0.164512	0.206538
1	linear	5.00	0.164513	0.206592
6	linear	1.00	0.164540	0.206506
8	linear	0.01	0.164569	0.206501
0	linear	10.00	0.164591	0.206780
7	linear	0.00	291.525260	197.048967

Polynomial Kernel

```
param_grid = [{'kernel':['polynomial'],'offset':[-2,-1,0,1,2,3,4], 'degree':[2,3,4,5,6],\
              'l2reg':[.1,0.05,0.0475,0.045,0.0425,0.04,0.035,0.03,.01]}]

kernel_ridge_regression_estimator = KernelRidgeRegression()
grid = GridSearchCV(kernel_ridge_regression_estimator,
                    param_grid,
                    cv = predefined_split,
                    scoring = make_scorer(mean_squared_error,greater_is_better = False),
                    return_train_score=True
                   )
grid.fit(np.vstack((x_train,x_test)),np.vstack((y_train,y_test)))
```

```
GridSearchCV(cv=PredefinedSplit(test_fold=array([-1, -1, ...,  0,  0])),
             estimator=KernelRidgeRegression(),
             param_grid=[{'degree': [2, 3, 4, 5, 6], 'kernel': ['polynomial'],
                          'l2reg': [0.1, 0.05, 0.0475, 0.045, 0.0425, 0.04,
                                    0.035, 0.03, 0.01],
                          'offset': [-2, -1, 0, 1, 2, 3, 4]}],
             return_train_score=True,
             scoring=make_scorer(mean_squared_error, greater_is_better=False))
```

```
pd.set_option('display.max_rows', 30)
df = pd.DataFrame(grid.cv_results_)
# Flip sign of score back, because GridSearchCV likes to maximize,
# so it flips the sign of the score if "greater_is_better=FALSE"
df['mean_test_score'] = -df['mean_test_score']
df['mean_train_score'] = -df['mean_train_score']
cols_to_keep = ["param_degree", "param_kernel","param_l2reg" ,"param_offset",
        "mean_test_score","mean_train_score"]
df_toshow = df[cols_to_keep].fillna('-')
df_toshow.sort_values(by=["mean_test_score"])
```

	param_degree	param_kernel	param_l2reg	param_offset	mean_test_score	mean_train_score
284	6	polynomial	0.0425	2	0.032413	0.048308
277	6	polynomial	0.0450	2	0.032422	0.048770
291	6	polynomial	0.0400	2	0.032429	0.047846
270	6	polynomial	0.0475	2	0.032453	0.049231
263	6	polynomial	0.0500	2	0.032504	0.049690

Linear Kernel

L2 reg: 4.0

RBF Kernel

L2 reg: 0.0695

Sigma: 0.0685

Polynomial Kernel:

degree: 6

L2 reg: 0.0425

offset: 2

RBF Kernel

```
param_grid = [{'kernel': ['RBF'],'sigma':[0.06,0.067,0.0685,0.069,0.07,0.08], 'l2reg': [0.071,0.0705,0.07,0.0695,0.069,0.06]}]
kernel_ridge_regression_estimator = KernelRidgeRegression()
grid = GridSearchCV(kernel_ridge_regression_estimator,
                    param_grid,
                    cv = predefined_split,
                    scoring = make_scorer(mean_squared_error,greater_is_better = False),
                    return_train_score=True
                   )
grid.fit(np.vstack((x_train,x_test)),np.vstack((y_train,y_test)))
```

```
GridSearchCV(cv=PredefinedSplit(test_fold=array([-1, -1, ...,  0,  0])),
             estimator=KernelRidgeRegression(),
             param_grid=[{'kernel': ['RBF'],
                          'l2reg': [0.071, 0.0705, 0.07, 0.0695, 0.069, 0.06],
                          'sigma': [0.06, 0.067, 0.0685, 0.069, 0.07, 0.08]}],
             return_train_score=True,
             scoring=make_scorer(mean_squared_error, greater_is_better=False))
```

```
pd.set_option('display.max_rows', 30)
df = pd.DataFrame(grid.cv_results_)
# Flip sign of score back, because GridSearchCV likes to maximize,
# so it flips the sign of the score if "greater_is_better=FALSE"
df['mean_test_score'] = -df['mean_test_score']
df['mean_train_score'] = -df['mean_train_score']
cols_to_keep = ["param_kernel","param_l2reg","param_sigma",
        "mean_test_score","mean_train_score"]
df_toshow = df[cols_to_keep].fillna('-')
df_toshow.sort_values(by=["mean_test_score"])
```

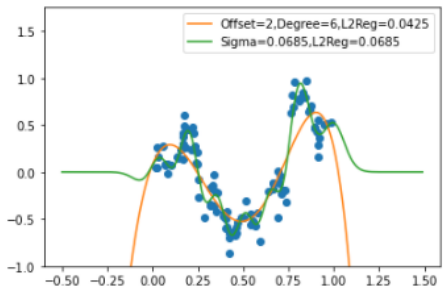
	param_kernel	param_l2reg	param_sigma	mean_test_score	mean_train_score
20	RBF	0.0695	0.0685	0.013832	0.014478
26	RBF	0.0690	0.0685	0.013832	0.014469
14	RBF	0.0700	0.0685	0.013832	0.014487
8	RBF	0.0705	0.0685	0.013832	0.014496
2	RBF	0.0710	0.0685	0.013832	0.014505
...	...	...	...	...	...
29	RBF	0.0690	0.0800	0.015417	0.017156
23	RBF	0.0695	0.0800	0.015432	0.017175
17	RBF	0.0700	0.0800	0.015447	0.017194
11	RBF	0.0705	0.0800	0.015462	0.017213
5	RBF	0.0710	0.0800	0.015477	0.017231

36 rows × 5 columns

30

30. 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.

```
## Plot the best polynomial and RBF fits you found
plot_step = .01
xpts = np.arange(-.5 , 1.5, plot_step).reshape(-1,1)
plt.plot(x_train,y_train,'o')
#Plot best polynomial fit
offset= 2
degree = 6
l2reg = 0.0425
k = functools.partial(polynomial_kernel, offset=offset, degree=degree)
f = train_kernel_ridge_regression(x_train, y_train, k, l2reg=l2reg)
label = "Offset="+str(offset)+",Degree="+str(degree)+",L2Reg="+str(l2reg)
plt.plot(xpts, f.predict(xpts), label=label)
#Plot best RBF fit
sigma = 0.0685
l2reg= 0.0685
k = functools.partial(RBF_kernel, sigma=sigma)
f = train_kernel_ridge_regression(x_train, y_train, k, l2reg=l2reg)
label = "Sigma="+str(sigma)+",L2Reg="+str(l2reg)
plt.plot(xpts, f.predict(xpts), label=label)
plt.legend(loc = 'best')
plt.ylim(-1,1.75)
plt.show()
```



RBF kernel (green) fits the data points better than the polynomial kernel.

31

31. 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 $\epsilon \sim \mathcal{N}(0, 0.1^2)$ (so $var(\epsilon) = 0.1^2$). The final point is $(x, f(x) + \epsilon)$. What is the Bayes decision function and the Bayes risk for the loss function $\ell(\hat{y}, y) = (\hat{y} - y)^2$.