DSGA 1003 HW3 yp2201@nyu.edu

1. According to the definition of subgradient, for $g \in \partial f_k(z)$,

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

Note that $f_k(x) = f(x)$,

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

$\therefore f(z) \geq f(x) + g^T(z-x), \Rightarrow g \in \partial f(x).$

2. $J(w) = \max\{0, 1 - yw^Tx\}$

i) $0 \geq 1 - yw^Tx$: $J(w) = 0$, $\nabla J(w) = g(w) = 0$

ii) $0 < 1 - yw^Tx$: $J(w) = 1 - yw^Tx$, $\nabla J(w) = g(w) = -yx$

$\therefore \nabla J(w) = g(w) = \begin{cases} 0, & \text{if } 1 - yw^Tx \leq 0 \\ -yx, & \text{otherwise} \end{cases}$

3. If f is convex, for all $\theta \in (0,1)$

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

If there is some point $x_0 \in (a,b)$, $\partial f(x) \neq \phi$, $\exists g \in R^n$ s.t. $g \in \partial f(x)$,

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

$f(a) \geq f(x_0) + g^T(a-x_0)$

$f(b) \geq f(x_0) + g^T(b-x_0)$

$\rightarrow \theta a + (1-\theta) b = \theta(a-b) + b \leq b$

$\rightarrow$ as $a \leq b$, $(1-\theta)a \leq (1-\theta)b$,

$a \leq \theta a + (1-\theta) b$

$\rightarrow a \leq \theta a + (1-\theta) b = x_0 \leq b$

Applying $x_0 = \theta a + (1-\theta) b$

$\theta \times f(a) \geq f(\theta a + (1-\theta) b) + g^T(a - \theta a - (1-\theta) b)$

$(1-\theta) \times f(b) \geq f(\theta a + (1-\theta) b) + g^T(b - \theta a - (1-\theta) b)$

If f is convex, $g \in \partial f(x) \geq 0$

$\Rightarrow \theta f(a) \geq \theta \cdot f(\theta a + (1-\theta) b) + \theta \cdot g^T(a - \theta a - (1-\theta) b)$

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

$\theta f(a) + (1-\theta) f(b) \geq f(\theta a + (1-\theta) b) + \theta g^T(a - x_0) + (1-\theta) g^T(b - x_0) \geq f(\theta a + (1-\theta) b)$

$\therefore \theta f(a) + (1-\theta) f(b) \geq f(\theta a + (1-\theta) b)$, and f is convex

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

i) $1 - y_i w^T x_i > 0$, $\nabla J_i(w) = \lambda w - y_i x_i$

ii) $1 - y_i w^T x_i < 0$, $\nabla J_i(w) = \lambda w + 0 = \lambda w$

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

$$\nabla \ell(m) = \begin{cases} 0, & \text{if } 1 - y_i w^T x_i < 0 \\ -y_i x_i, & \text{if } 1 - y_i w^T x_i > 0 \\ \text{undefined}, & \text{if } 1 - y_i w^T x_i = 0 \end{cases}$$

$$\therefore \nabla J_i(w) = \nabla\left(\frac{\lambda}{2}\|w\|^2\right) + \underbrace{\nabla \max\{0, 1 - y_i w^T x_i\}}_{= \nabla \ell(m)}$$

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

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

As mentioned in Q4, ① $\nabla\left(\frac{\lambda}{2}\|w\|^2\right) = \lambda w$

② $\nabla \max\{0, 1 - y_i w^T x_i\} = \begin{cases} -y_i x_i, & \text{if } 1 - y_i w^T x_i > 0 \Rightarrow y_i w^T x_i < 1 \\ 0, & \text{if } y_i w^T x_i \geq 1 \end{cases}$

$$\therefore \nabla J_i(w) = gw = \begin{cases} \lambda w - y_i x_i, & \text{for } y_i w^T x_i < 1 \\ \lambda w, & \text{for } y_i w^T x_i \geq 1 \end{cases}$$

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.

:
```
# 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}.
import collections

def convert_bag_of_words(list_of_words):
    return collections.Counter(list_of_words)
```

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 corre- sponding 1 or -1 labels. Format the test set similarly.

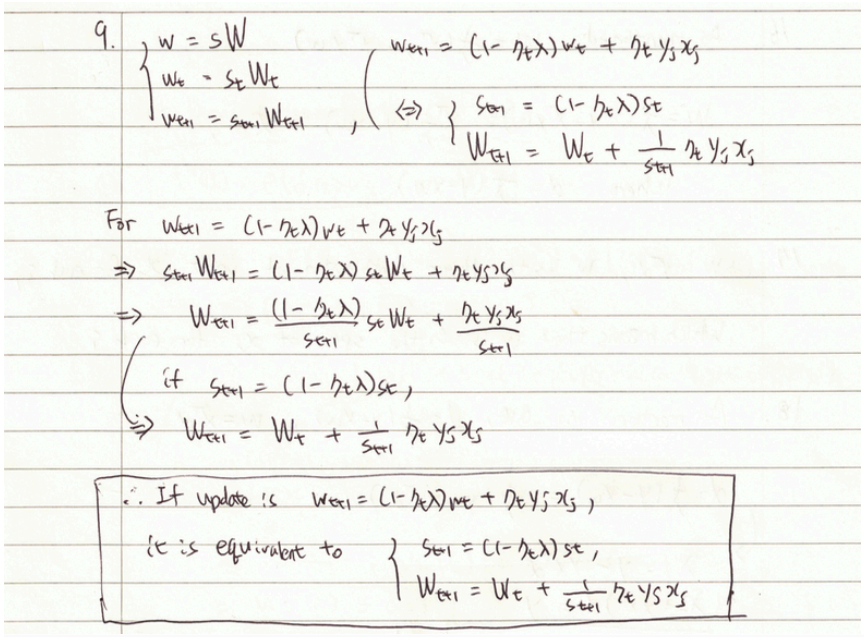
```
from sklearn.model_selection import train_test_split

# load data
list_of_words = load_and_shuffle_data()

# initialize feature, target value list
X, y = [], []

# For each word list, append to feature and target value list
# For features, convert into bag of words and append
for each_word_list in list_of_words:
    X.append(convert_bag_of_words(each_word_list[:-1]))
    y.append(each_word_list[-1])

# split into 1500 train, 500 valiation examples
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=2021)
```

Q8.

- Implement the Pegasos algorithm to run on a sparse data representation. The output should be a sparse weight vector 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.

```
: # This function is needed in advance to check if loss is converging (Same code as Q10)
def classification_error(X, y, w):
    # initialize error
    error = 0

    # iterate and if prediction is wrong, add 1
    for i in range(len(X)):
        pred = dotProduct(X[i], w)
        if (pred * y[i]) < 0:
            error += 1

    # divide by total count
    error /= len(X)

    return error
```

```
import time

def pegasos_algo(X_train, y_train, X_test, y_test, lambda_reg=1, num_iter=100):
    # Note that our Pegasos algorithm starts at w = 0, which corresponds to an empty dictionary.
    # initializing variables, and setting start time
    start = time.time()
    w = {}
    t = 0

    # initializing loss and adding initial result
    loss_check = []
    loss_check.append(classification_error(X_test, y_test, w))

    index = np.arange(len(X_train))

    # iterate by given num_iter
    for epoch in range(1, num_iter):

        # for every X, t = t+1, ηt = 1/tλ
        # wt+1 = (1 − ηtλ)wt for every condition, and add ηtyjxj if yjwtTxj <1
        np.random.seed(0)
        np.random.shuffle(index)
        for j in index:
            t += 1
            eta = 1 / (lambda_reg*t)

            # if y[j] * dotProduct(X[j], w) < 1, add
            if y_train[j] * dotProduct(X_train[j], w) < 1:
                increment(w, -eta*lambda_reg, w)
                increment(w, eta*y_train[j], X_train[j])
            else:
                increment(w, -eta*lambda_reg, w)

        # append calculated loss after iteration
        loss_check.append(classification_error(X_test, y_test, w))

        # converge checker: checking if loss difference is really small
        if abs(loss_check[epoch] - loss_check[epoch-1]) < 10**-8:
            print('labmda: {}, epoch {}, loss:{}'.format(lambda_reg, epoch, loss_check[epoch]))
            break

    end = time.time()
    print('lambda: {}, epoch: {}, loss: {}'.format(lambda_reg, epoch, classification_error(X_test, y_test, w)))
    print('total time spent:', end - start)

    # The output should be a sparse weight vector w represented as a dictionary.
    return w
```



Q9.

- Implement the Pegasos algorithm with the (s, W) representation described above.

```
import time

def pegasos_algo_v2(X_train, y_train, X_test, y_test, lambda_reg=1, num_iter=100):
    # Note that our Pegasos algorithm starts at w = 0, which corresponds to an empty dictionary.
    # initializing variables, and setting start time
    start = time.time()
    big_w = {}
    small_w = {}
    t = 1
    s = 1

    # initializing loss and adding initial result
    loss_check = []
    loss_check.append(classification_error(X_test, y_test, small_w))

    index = np.arange(len(X_train))

    # iterate by given num_iter
    for epoch in range(1, num_iter):

        # for every X, t = t+1, ηt = 1/tλ
        # wt+1 = (1 − ηtλ)wt for every condition, and add ηtyjxj if yjwtTxj <1
        np.random.seed(0)
        np.random.shuffle(index)
        for j in index:
            # start from t=2
            t += 1
            eta = 1 / (lambda_reg*t)
            s *= (1 - (eta * lambda_reg))

            # update big w
            if (y_train[j] * dotProduct(X_train[j], big_w)) < 1/s:
                increment(big_w, (1/s)*eta*y_train[j], X_train[j])

        # update small w: for every iteration, it should be initialized and just updated by result of big w
        small_w = {}
        increment(small_w, s, big_w)

        # append calculated loss after iteration
        loss_check.append(classification_error(X_test, y_test, small_w))

        # converge checker
        if abs(loss_check[epoch] - loss_check[epoch-1]) < 10**-8:
            print('(converged)labmda: {}, epoch: {}, loss: {}'.format(lambda_reg, epoch, loss_check[epoch]))
            break

    end = time.time()
    print('(final)lambda: {}, epoch: {}, loss: {}'.format(lambda_reg, epoch, \
                                                          classification_error(X_test, y_test, small_w)))
    print('total time spent:', end - start)

    # The output should be a sparse weight vector w represented as a dictionary.
    return small_w
```

Q10.

- 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.

Ans)

- Both algorithm gives the same result. Trivial difference is due to the shuffling of the index
- Time taken to run each approach is as follows:
  First approach: 20.95
  Second approach: 0.48

```
w1 = pegasos_algo(X_train, y_train, X_test, y_test, 0.01, 5)
w1
```

```
lambda: 0.01, epoch: 4, loss: 0.21
total time spent: 20.9457049369812

{"here's": -0.48333333333333317,
 'a': 0.15000000000000435,
 'word': 0.1999999999999996,
 'analogy': 0.08333333333333338,
 'amistad': 0.2833333333333348,
 'is': -0.5333333333333358,
 'to': -1.599999999999997,
 'the': 1.23333333333333,
 'lost': -0.4833333333333316,
 'world': 1.3999999999999981,
 'as': 1.5333333333333288,
```

```
w2 = pegasos_algo_v2(X_train, y_train, X_test, y_test, 0.01, 5)
w2
```

```
(final)lambda: 0.01, epoch: 4, loss: 0.21
total time spent: 0.47692227363586426

{"here's": -0.4832527912014646,
 'a': 0.1499750041659735,
 'word': 0.1999666722212962,
 'analogy': 0.08331944675887325,
 'amistad': 0.2832861189801694,
 'is': -0.5332444592567911,
 'to': -1.5997333777703695,
 'the': 1.2331278120313252,
 'lost': -0.48325279120146536,
 'world': 1.3997667055490712,
 'as': 1.5330778203632724,
```

Q11.

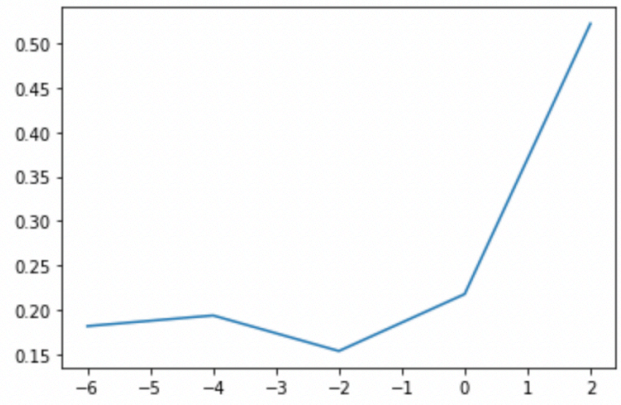
- Write a function classification error that takes a sparse weight vector w, a list of sparse vectors X and the corresponding list of labels y, and returns the fraction of errors when predicting yi using sign(wT xi). In other words, the function reports the 0-1 loss of the linear predictor f (x) = wT x.

```
def classification_error(X, y, w):
    # initialize error
    error = 0

    # iterate and if prediction is wrong, add 1
    for i in range(len(X)):
        pred = dotProduct(X[i], w)
        if (pred * y[i]) < 0:
            error += 1

    error /= len(X)

    return error
```

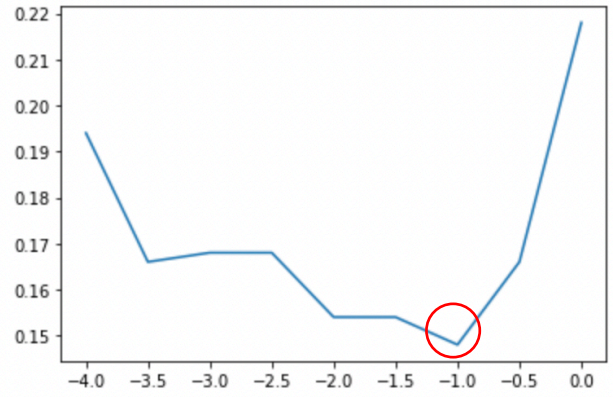
Q12.

- 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 addi- tional 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.)

```
# checking broad range: 10**-6 ~ 10**2
import matplotlib.pyplot as plt
%matplotlib inline

lambda_list = [10**-6, 10**-4, 10**-2, 10**0, 10**2]
error_list = []

for each_lambda in lambda_list:
    w = pegasos_algo_v2(X_train, y_train, X_test, y_test, each_lambda, 100)
    error_list.append(classification_error(X_test, y_test, w))

plt.plot(np.log10(lambda_list), error_list)
plt.show()
```




```
# zooming range: 10**-4 ~ 10**0
# (final)lambda: 0.1, loss: 0.148
import matplotlib.pyplot as plt
%matplotlib inline

lambda_list = [10**-4, 10**-3.5, 10**-3, 10**-2.5, 10**-2, 10**-1.5, 10**-1, 10**-0.5, 10**0]
error_list = []

for each_lambda in lambda_list:
    w = pegasos_algo_v2(X_train, y_train, X_test, y_test, each_lambda, 100)
    error_list.append(classification_error(X_test, y_test, w))

plt.plot(np.log10(lambda_list), error_list)
plt.show()
```



Q13.

- {BONUS} 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?

Ans)

- Logic: divide into 5 groups by absolute value of the score. First group has the largest values
- for group: 1, the percentage error is: 0.04
- for group: 2, the percentage error is: 0.05
- for group: 3, the percentage error is: 0.14
- for group: 4, the percentage error is: 0.18
- for group: 5, the percentage error is: 0.33

-> Findings: there are significant negative relationships between the percentage error and the absolute value of the score

```
# select optimal lambda in question 12
lambda_reg = 10**-1

# train the dataset and update w
small_w = pegasos_algo_v2(X_train, y_train, X_test, y_test, lambda_reg, 100)

# create a result list that will contain (score, label)
res_list = []

# iterate by length of y_test, and check if result is true or false
# append (absolute score, label) pair
for i in range(len(y_test)):
    res = 0
    pred = dotProduct(X_test[i], small_w)
    if (pred * y_test[i] < 0):
        res = 1
    res_list.append((abs(pred), res))

# sort the result by largest scores (descending order)
res_list_sorted = sorted(res_list, key=lambda x: x[0], reverse=True)

# set 5 groups, and make group per numbers
group = 5
group_per_num = len(y_test) / group

# iterate by group and calculate the percentge error
for i in range(group):
    small_group = res_list_sorted[(i)*int(group_per_num):(i+1)*int(group_per_num)]

    error_sum = 0

    for each_member in small_group:
        error_sum += each_member[1]

    error_sum /= len(small_group)

    print('for group: {}, the percentage error is: {}'.format(i+1, error_sum))
```

```
for group: 1, the percentage error is: 0.04
for group: 2, the percentage error is: 0.05
for group: 3, the percentage error is: 0.14
for group: 4, the percentage error is: 0.18
for group: 5, the percentage error is: 0.33
```

Q14.

(Optional) Choose an input example x = (x1,...,xd) ∈ Rd 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 |wixi|, where wi is the weight of the ith feature in the prediction function, and xi is the value of the ith feature in the input x. Create a table of the most important features, sorted by |wixi|, including the feature name, the feature value xi, the feature weight wi, and the product wixi. 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.)

Ans)

- the model is incorrect as it gives a high score on stopwords (i.e. have, are, to, been, and, a, the, as)
- Those should not be the feature that determines the result.

-> 1) I would remove stopwords in pre-processing stage
-> 2) I would try tf-idf methods, to decrease the effect of common words that affects every documents

```
# select optimal lambda in question 12
lambda_reg = 10**-1

# train the dataset and update w
small_w = pegasos_algo_v2(X_train, y_train, X_test, y_test, lambda_reg, 100)

for i in range(len(y_test)):
    res = 0
    pred = dotProduct(X_test[i], small_w)
    if (pred * y_test[i] < 0):
        res = 1
    res_list.append((abs(pred), res))
```

```
# index 2 and 3 are misclassified reviews and I will check those
res_list[2], res_list[6]
```

```
((0.6373227112881324, 1), (2.0316328061198137, 1))
```

```
# analysis for index 2

misclassified_list_idx2 = []

for word in X_test[2]:
    w = small_w.get(word, 0)
    x = X_test[2][word]

    misclassified_list_idx2.append((word, abs(w*x), w, x, w*x))

misclassified_list_idx2_sorted = sorted(misclassified_list_idx2, key=lambda x: x[1], reverse=True)

misclassified_list_idx2_sorted[:7]
```

```
[('have', 0.28532857785703164, -0.07133214446425791, 4, -0.28532857785703164),
 ('are', 0.2179963667272206, -0.0363327277878701, 6, -0.2179963667272206),
 ('to', 0.20766320561323787, -0.029666172230462553, 7, -0.20766320561323787),
 ('been', 0.18699688338527648, -0.03116614723087941, 6, -0.18699688338527648),
 ('and', 0.1459975667072225, 0.036499391676805625, 4, 0.1459975667072225),
 ('true', 0.14433092781786938, 0.07216546390893469, 2, 0.14433092781786938),
 ('a', 0.12799786670222132, 0.021332977783703553, 6, 0.12799786670222132)]
```

```
# analysis for index 6

misclassified_list_idx6 = []

for word in X_test[6]:
    w = small_w.get(word, 0)
    x = X_test[6][word]

    misclassified_list_idx6.append((word, abs(w*x), w, x, w*x))

misclassified_list_idx6_sorted = sorted(misclassified_list_idx6, key=lambda x: x[1], reverse=True)

misclassified_list_idx6_sorted[:7]
```

```
[('and', 2.153464108931532, 0.036499391676805625, 59, 2.153464108931532),
 ('to', 1.661305644905903, -0.029666172230462553, 56, -1.661305644905903),
 ('a', 1.3866435559407309, 0.021332977783703553, 65, 1.3866435559407309),
 ('the', 1.2389793503441549, 0.006999883335277711, 177, 1.2389793503441549),
 ('only', 0.9559840669322085, -0.11949800836652606, 8, -0.9559840669322085),
 ('as', 0.6969883835269407, 0.040999316678055335, 17, 0.6969883835269407),
 ('even', 0.592990116831383, -0.09883168613856384, 6, -0.592990116831383)]
```

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

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

$= w^T X^T X w - w^T X^T y - y^T X w + y^T y + \lambda w^T w$

$= w^T X^T X w - 2y^T X w + y^T y + \lambda w^T w$

$\nabla J(w) = 2X^T X w - 2X^T y + 0 + 2\lambda I w = 0$

① $\rightarrow$ $\boxed{X^T X w + \lambda I w = X^T y}$

$\rightarrow (X^T X + \lambda I) w = X^T y$, $\boxed{w = (X^T X + \lambda I)^{-1} X^T y}$ ②

For any vector $b \neq 0$, $b^T X^T X b = (Xb)^T(Xb) \geq 0$, $b^T \lambda I b > 0$ $(\lambda > 0)$

So, $b^T (X^T X + \lambda I) b = \underbrace{b^T X^T X b}_{\geq 0} + \underbrace{b^T \lambda I b}_{> 0} > 0$.

Given Appendix's definition of spd,

$\boxed{X^T X + \lambda I \text{ is a spd, and spd is invertible.}}$ ③

16. As mentioned, $w = \frac{1}{\lambda}(X^T y - X^T X w)$

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

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

17. $w = X^T \alpha$, w is the linear combination of vector x_i for each i,

which means that w is in the span of x_i for each i

18. As mentioned in Q16, $\alpha = \frac{1}{\lambda}(y - Xw)$, $w = X^T \alpha$.

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

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

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

$\boxed{\therefore \alpha = (\lambda I + XX^T)^{-1} y}$ (As mentioned in Q15, $\lambda I + XX^T$ is invertible)

19. As mentioned in Q18, $\alpha = (\lambda I + XX^T)^{-1} y$, $w = X^T \alpha$

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

20. $\boxed{f(x) = x^T w^* = x^T X^T (\lambda I + XX^T)^{-1} y}$

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

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

Q21.

- Write functions that compute the RBF kernel kRBF(σ)(x,x′) = exp□−ɪx−x′ɪ2/□2σ2□□ and the polynomial kernel kpoly(a,d)(x, x′) = (a + ⟨x, x′⟩)d. The linear kernel klinear(x, x′) = ⟨x,x′⟩, has been done for you in the support code. Your functions should take as input two matrices W ∈ Rn1×d and X ∈ Rn2×d and should return a matrix M ∈ Rn1×n2 where Mij = k(Wi·, Xj·). In words, the (i, j)'th entry of M should be kernel evaluation between wi (the ith row of W) and xj (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 import distance
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
    dis = distance.cdist(X1, X2, 'euclidean')
    return np.exp(-dis ** 2 / (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
```

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)
```

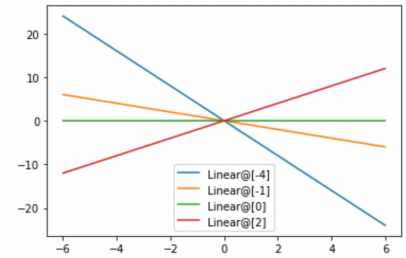
```
[[16.  4.  0. -8.]
 [ 4.  1.  0. -2.]
 [ 0.  0.  0.  0.]
 [-8. -2.  0.  4.]]
```

Q23.

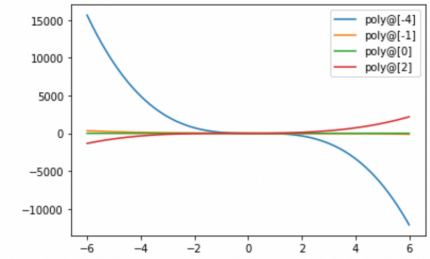
- Suppose we have the data set DX,y = {(−4, 2), (−1, 0), (0, 3), (2, 5)} (in each set of paren- theses, the first number is the value of xi and the second number the corresponding value of the target yi). Then by the representer theorem, the final prediction function will be in the span of the functions x □→ k(x0, x) for x0 ∈ DX = {−4, −1, 0, 2}. This set of functions will look quite different depending on the kernel function we use. The set of functions x □→ klinear(x0, 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].
(b) Plot the set of functions x □→ kRBF(1)(x0, x) for x0 ∈ X and for x ∈ [−6, 6].

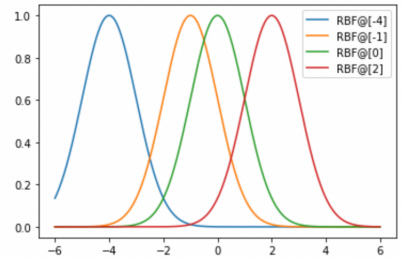
```
# PLot kernel machine functions

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()
```




```
# a) PLot poly machine functions

plot_step = .01
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

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




```
# b) PLot RBF machine functions

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)

# Plot 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()
```

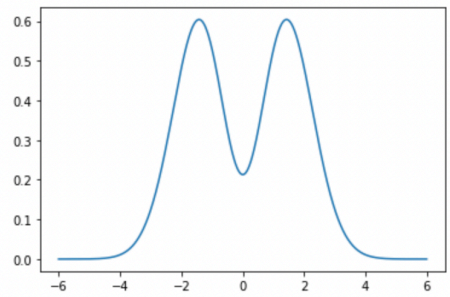


Q24.

- By the representer theorem, the final prediction function will be of the form f(x) = ∑ni=1 αik(xi,x), where x1,...,xn ∈ 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 correspond- ing 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
        preds = np.dot(self.weights.T, self.kernel(self.training_points, X))

        return preds.T
```

```
# Plotting the result function

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)

k = functools.partial(RBF_kernel, sigma=sigma)
k_machine = Kernel_Machine(k, prototypes, weights)

plt.plot(xpts, k_machine.predict(xpts))
plt.show()
```



Q25.

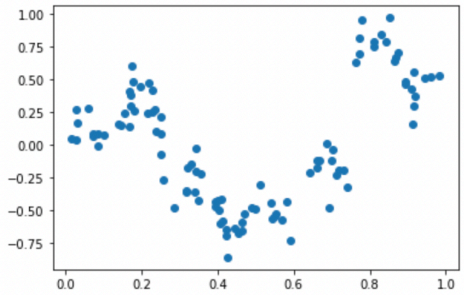
- Plot the training data. You should note that while there is a clear relationship between x and y, the relationship is not linear.

Ans) Plotting the train data below. We can see that while there is a clear relationship between x and y, the relationship is not linear.

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

```
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')
plt.show()
```



Q26.

- In a previous problem, we showed that in kernelized ridge regression, the final prediction function is f(x) = ∑ni=1 αik(xi, x), where α = (λI + K)−1y and K ∈ Rn×n is the kernel matrix of the training data: Kij = k(xi,xj), for x1,...,xn. In terms of kernel machines, αi is the weight on the kernel function evaluated at the training point xi. 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
    alpha = np.dot(np.linalg.inv(l2reg * np.identity(X.shape[0]) + kernel(X, X)), y)
    return Kernel_Machine(kernel, X, alpha)
```

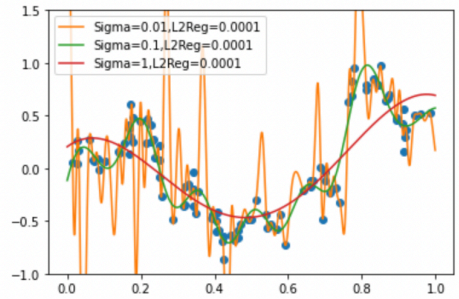
Q27.

- 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?

Ans)

- Sigma = 0.01 would be most likely to overfit
- Sigma = 1 would be less likely to overfit

```
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()
```

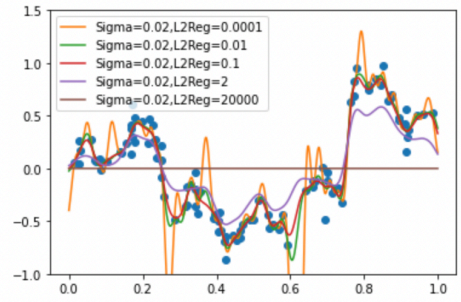


Q28.

- 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 λ → ∞?

Ans) When λ → ∞ (In my example, λ = 20000), the prediction function is a straight line ovelapped with x axis

```
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, 20000]:
    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()
```



Q29.

- {BONUS} 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.

```
from sklearn.base import BaseEstimator, RegressorMixin, ClassifierMixin

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

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

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

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

        return self

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

        return(self.kernel_machine_.predict(X))

    def score(self, X, y=None):
        return(((self.predict(X)-y)**2).mean())
```

```
param_grid = [{'kernel': ['RBF'],'sigma':[0.06, 0.061, 0.062, 0.063, 0.064, 0.065, 0.066, 0.067, 0.068, 0.069, 0.070
               'l2reg': [0.115, 0.117, 0.119, 0.121, 0.123, 0.125, 0.127, 0.129, 0.131, 0.133, 0.135]},
              {'kernel':['polynomial'],'offset':[-5, -4, -3, -2, -1, 0, 1, 0, 1, 2, 3, 4, 5], \
               'degree':[1,2,3,4,5,6,7,8,9,10],'l2reg':[0.005, 0.006, 0.007, 0.008, 0.009, 0.01] },
              {'kernel':['linear'],'l2reg': [10,9,8,7,6,5,4,3,2,1,0.1,.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': ['RBF'],
                          'l2reg': [0.115, 0.117, 0.119, 0.121, 0.123, 0.125,
                                    0.127, 0.129, 0.131, 0.133, 0.135],
                          'sigma': [0.06, 0.061, 0.062, 0.063, 0.064, 0.065,
                                    0.066, 0.067, 0.068, 0.069, 0.07]},
                         {'degree': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
                          'kernel': ['polynomial'],
                          'l2reg': [0.005, 0.006, 0.007, 0.008, 0.009, 0.01],
                          'offset': [-5, -4, -3, -2, -1, 0, 1, 0, 1, 2, 3, 4,
                                     5]},
                         {'kernel': ['linear'],
                          'l2reg': [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0.1, 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_degree", "param_kernel","param_l2reg" ,"param_offset","param_sigma",
        "mean_test_score","mean_train_score"]
df_toshow = df[cols_to_keep].fillna('-')
df_toshow.sort_values(by=["mean_test_score"])
```

	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_train_score
80	-	RBF	0.129	-	0.063	0.013808	0.014452
91	-	RBF	0.131	-	0.063	0.013808	0.014474
69	-	RBF	0.127	-	0.063	0.013808	0.014429
102	-	RBF	0.133	-	0.063	0.013808	0.014497
58	-	RBF	0.125	-	0.063	0.013809	0.014407
...	...	...	...	...	...	...	...
710	8	polynomial	0.008	-1	-	0.471731	0.337790
892	10	polynomial	0.010	-1	-	0.516987	0.353146
671	8	polynomial	0.005	-1	-	1.141505	0.689108
697	8	polynomial	0.007	-1	-	1.296353	0.866005
684	8	polynomial	0.006	-1	-	103.227772	64.367345

913 rows × 7 columns

Linear kernel: The best hyperparameter for linear kernel is param_l2reg = 4

- By firstly checking basic results, param_l2reg = 1 had good result.
- When I added more parameters to check the best hyperparameter(1~10), I found param_l2reg = 4 as an optimal value.
- We can also see that making small change in any one of the hyperparameters in either direction may cause the performance to get worse.

```
df_toshow[df_toshow['param_kernel'] == 'linear'].sort_values(by='mean_test_score')
```

	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_train_score
907	-	linear	4.00	-	-	0.164510	0.206563
908	-	linear	3.00	-	-	0.164512	0.206538
906	-	linear	5.00	-	-	0.164513	0.206592
905	-	linear	6.00	-	-	0.164521	0.206625
909	-	linear	2.00	-	-	0.164522	0.206518
904	-	linear	7.00	-	-	0.164534	0.206661
910	-	linear	1.00	-	-	0.164540	0.206506
903	-	linear	8.00	-	-	0.164550	0.206699
911	-	linear	0.10	-	-	0.164565	0.206501
912	-	linear	0.01	-	-	0.164569	0.206501
902	-	linear	9.00	-	-	0.164569	0.206739
901	-	linear	10.00	-	-	0.164591	0.206780

RBF kernel: The best hyperparameter for RBF kernel: param_sigma = 0.063, param_l2reg = 0.129

- By firstly checking basic results, param_l2reg = 0.0625, param_sigma = 0.1 had good result.

- Param_sigma: When I applied 0.01 ~ 0.1 (+0.01), I decided to check close range of 0.06~0.07
- By checking more close range, I found 0.062 ~ 0.064 as a good result range

- Param_l2reg: When I applied 2 ** -5 ~ 2 ** 5 (+1), I found 0.1250 had good results.
- When applying more close range (0.115 ~ 0.135 (+0.002)), I found 0.127 ~ 0.133 as a good result range

- By combining both, I found param_sigma = 0.063, param_l2reg = 0.129 as an optimal value.
- We can also see that making small change in any one of the hyperparameters in either direction may cause the performance to get worse.

```
df_toshow[df_toshow['param_kernel'] == 'RBF'].sort_values(by='mean_test_score')
```

	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_train_score
80	-	RBF	0.129	-	0.063	0.013808	0.014452
91	-	RBF	0.131	-	0.063	0.013808	0.014474
69	-	RBF	0.127	-	0.063	0.013808	0.014429
102	-	RBF	0.133	-	0.063	0.013808	0.014497
58	-	RBF	0.125	-	0.063	0.013809	0.014407
...	...	...	...	...	...	...	...
76	-	RBF	0.127	-	0.07	0.014306	0.015795
87	-	RBF	0.129	-	0.07	0.014326	0.015830
98	-	RBF	0.131	-	0.07	0.014347	0.015865
109	-	RBF	0.133	-	0.07	0.014367	0.015899
120	-	RBF	0.135	-	0.07	0.014387	0.015933

Polynomial kernel: The best hyperparameter for Polynomial kernel:

param_degree = 9, param_l2reg = 0.008, param_offset = -3 ¶

- param_degree: seeking 1 ~ 10, I found 8 ~ 10 as a good parameter range
- param_l2reg: seeking 0.005 ~ 0.010, I found 0.007~0.009 as a good parameter range
- param_offset: seeking -5 ~ 5, I found -3 ~ -1 as a good parameter range

- By combining all, I found param_degree = 9, param_l2reg = 0.008, param_offset = -3 as an optimal value.
- We can also see that making small change in any one of the hyperparameters in either direction may cause the performance to get worse.

```
df_toshow[df_toshow['param_kernel'] == 'polynomial'].sort_values(by='mean_test_score')
```

	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_train_score
786	9	polynomial	0.008	-3	-	0.030211	0.040353
773	9	polynomial	0.007	-3	-	0.030243	0.040048
799	9	polynomial	0.009	-3	-	0.030323	0.040800
760	9	polynomial	0.006	-3	-	0.030361	0.039838
747	9	polynomial	0.005	-3	-	0.030531	0.039695
...	...	...	...	...	...	...	...
710	8	polynomial	0.008	-1	-	0.471731	0.337790
892	10	polynomial	0.010	-1	-	0.516987	0.353146
671	8	polynomial	0.005	-1	-	1.141505	0.689108
697	8	polynomial	0.007	-1	-	1.296353	0.866005
684	8	polynomial	0.006	-1	-	103.227772	64.367345

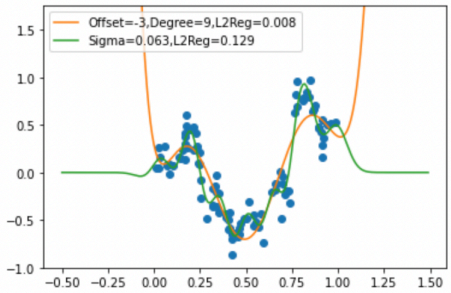
Q30.

- {BONUS} Plot your best fitting prediction functions using the polynomial kernel and the RBF kernel. Use the domain x ∈ (−0.5, 1.5). Comment on the results.

Ans)

- Prediction functions using the RBF kernel better fits the data than the polynomial kernel
- For outside of the training points, the polynomial kernel acts very unreasonable.

```
## 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= -3
degree = 9
l2reg = 0.008
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 = 0.063
l2reg= 0.129
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()
```



31. Bayes decision function

Let Bayes decision function : $f^*(x)$

① $\boxed{f^*(x) = E(f(x)+\epsilon) = f(x)}$

② Bayes Risk: $E(l(\hat{y},y)) = E((\hat{y}-y)^2) = E(\hat{y}^2+y^2-2\hat{y}y)$

$= E(\hat{y}^2) + E(y^2) - 2E(\hat{y}y)$

$= Var(\hat{y}) + \{E(\hat{y})\}^2 + Var(y) + \{E(y)\}^2 - 2(cov(\hat{y},y) + E(\hat{y})E(y))$

As we sampled x uniformly from (0,1),

$E(y)=1$, $Var(y)=Var(\epsilon)=0.01$, $cov(\hat{y},y)=0$.

$E(\hat{y})=1$ $Var(\hat{y})=0$

$= 0 + 1 + Var(\epsilon) + 1 - 2(0+1)$

$\boxed{= Var(\epsilon) = 0.01}$

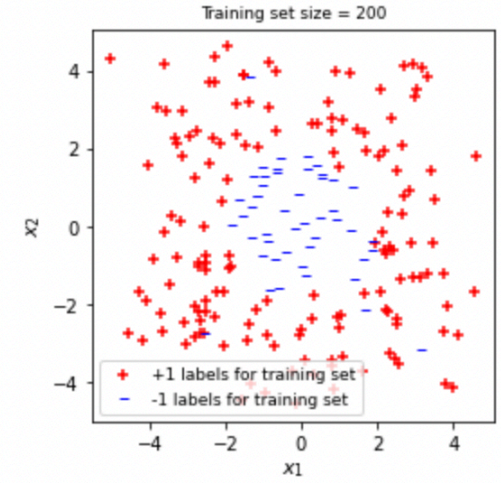
Q32.

- Load the SVM training svm-train.txt and svm-test.txt test data from the zip file. Plot the training data using the code supplied. Are the data linearly separable? Quadratically separable? What if we used some RBF kernel?

Ans)

- Above data may not be linearly seperable and quadratically separable.
- It might get much better solution through RBF kernel

```
# Load and plot the SVM data
#load the training and test sets
data_train,data_test = np.loadtxt("svm-train.txt"),np.loadtxt("svm-test.txt")
x_train, y_train = data_train[:,0:2], data_train[:,2].reshape(-1,1)
x_test, y_test = data_test[:,0:2], data_test[:,2].reshape(-1,1)

#determine predictions for the training set
yplus = np.ma.masked_where(y_train[:,0]<=0, y_train[:,0])
xplus = x_train[~np.array(yplus.mask)]
yminus = np.ma.masked_where(y_train[:,0]>0, y_train[:,0])
xminus = x_train[~np.array(yminus.mask)]

#plot the predictions for the training set
figsize = plt.figaspect(1)
f, (ax) = plt.subplots(1, 1, figsize=figsize)

pluses = ax.scatter (xplus[:,0], xplus[:,1], marker='+', c='r', label = '+1 labels for training set')
minuses = ax.scatter (xminus[:,0], xminus[:,1], marker=r'$-$', c='b', label = '-1 labels for training set')

ax.set_ylabel(r"$x_2$", fontsize=11)
ax.set_xlabel(r"$x_1$", fontsize=11)
ax.set_title('Training set size = %s'% len(data_train), fontsize=9)
ax.axis('tight')
ax.legend(handles=[pluses, minuses], fontsize=9)
plt.show()
```



Q33.

- Unlike for kernel ridge regression, there is no closed-form solution for SVM classification (kernelized or not). Implement kernelized Pegasos. Because we are not using a sparse representation for this data, you will probably not see much gain by implementing the “optimized” versions described in the problems above.

```
def train_soft_svm(x_train, y_train, k, l2_reg=0.01):
    alpha = np.zeros(len(x_train))
    kernel = k(x_train, x_train)
    t = 0 # step number
    max_step = 1000

    while t < max_step:
        t += 1
        eta = 1 / (t*l2_reg)

        j = np.random.randint(1, len(x_train))
        kernel_j = kernel[j, :]

        if y_train[j] * (kernel_j.T @ alpha) < 1:
            alpha[j] *= (1 - eta * l2_reg)
            alpha[j] += eta * y_train[j]
        else:
            alpha[j] *= (1 - eta * l2_reg)

    return Kernel_Machine(k, x_train, alpha)
```

Q34.

- Find the best hyperparameter settings (including kernel parameters and the regularization parameter) for each of the kernel types. Summarize your results in a table, which gives training error and test error (i.e. average 0/1 loss) for each setting. Include in your table the best settings for each kernel type, as well as nearby settings that show that making small change in any one of the hyperparameters in either direction will cause the performance to get worse. You should use the 0/1 loss on the test set to rank the parameter settings.

```
from sklearn.base import BaseEstimator, RegressorMixin, ClassifierMixin

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

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

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

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

        return self

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

        return(self.kernel_machine_.predict(X))

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

```
# converting 1,-1 values into 1, 0
Y_train = np.zeros(len(y_train)).reshape(-1,1)
Y_test = np.zeros(len(y_test)).reshape(-1,1)

idx = 0
for y in y_train:
    if y > 0: Y_train[idx] = 1
    else: Y_train[idx] = 0
    idx += 1

idx = 0
for y in y_test:
    if y > 0: Y_test[idx] = 1
    else: Y_test[idx] = 0
    idx += 1
```

```
from sklearn.model_selection import GridSearchCV,PredefinedSplit
from sklearn.model_selection import ParameterGrid
from sklearn.metrics import mean_squared_error,make_scorer
import pandas as pd

test_fold = [-1]*len(x_train) + [0]*len(x_test)   #0 corresponds to test, -1 to train
predefined_split = PredefinedSplit(test_fold=test_fold)
```

```
param_grid = [{'kernel': ['RBF'],'sigma':[1.116, 1.118, 1.2, 1.202, 1.204, 1.206, 1.208], \
               'l2reg': [0.109, 0.111, 0.113, 0.115, 0.117, 0.119, 0.121, 0.123, 0.125]},
              {'kernel':['polynomial'],'offset':[-5,-4,-3,-2,-1,0,1,2,3,4,5], 'degree':[2,3,4,5,6],\
               'l2reg':[500, 600, 700, 800, 900, 1000, 1100, 1200, 1300, 1400, 1500] },
              {'kernel':['linear'],'l2reg': [3400, 3450, 3500, 3550, 3600]}]

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': [0.109, 0.111, 0.113, 0.115, 0.117, 0.119,
                                    0.121, 0.123, 0.125],
                          'sigma': [1.116, 1.118, 1.2, 1.202, 1.204, 1.206,
                                    1.208]},
                         {'degree': [2, 3, 4, 5, 6], 'kernel': ['polynomial'],
                          'l2reg': [500, 600, 700, 800, 900, 1000, 1100, 1200,
                                    1300, 1400, 1500],
                          'offset': [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]},
                         {'kernel': ['linear'],
                          'l2reg': [3400, 3450, 3500, 3550, 3600]}],
             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_degree", "param_kernel","param_l2reg" ,"param_offset","param_sigma",
        "mean_test_score","mean_train_score"]
df_toshow = df[cols_to_keep].fillna('-')
df_toshow.sort_values(by=["mean_test_score"])
```

	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_train_score
4	-	RBF	0.109	-	1.204	0.046384	0.034200
5	-	RBF	0.109	-	1.206	0.046385	0.034261
3	-	RBF	0.109	-	1.202	0.046385	0.034139
6	-	RBF	0.109	-	1.208	0.046385	0.034323
2	-	RBF	0.109	-	1.2	0.046385	0.034077
...	...	...	...	...	...	...	...
285	3	polynomial	1400.000	-3	-	0.895573	0.942424
209	3	polynomial	700.000	-2	-	1.088235	1.025443
296	3	polynomial	1500.000	-3	-	1.880582	1.949127
198	3	polynomial	600.000	-2	-	3.928099	3.796035

Linear kernel: The best hyperparameter for linear kernel is param_l2reg = 3450

- When I firstly checked param_l2reg, it seemed to require large number to
- We can also see that making small change in any one of the hyperparameters in either direction may cause the performance to get worse.

```
df_toshow[df_toshow['param_kernel'] == 'linear'].sort_values(by='mean_test_score')
```

	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_train_score
669	-	linear	3450.0	-	-	0.779614	0.757294
668	-	linear	3400.0	-	-	0.779614	0.757267
670	-	linear	3500.0	-	-	0.779614	0.757320
671	-	linear	3550.0	-	-	0.779614	0.757346
672	-	linear	3600.0	-	-	0.779614	0.757371

RBF kernel: The best hyperparameter for RBF kernel: param_sigma = 1.204, param_l2reg = 0.109

- By firstly checking basic results, param_l2reg = 0.0625, param_sigma = 0.1 had good result.
- Param_sigma: By checking more close range as we did on previous questions, I found 0.107 ~ 0.111 as a good result range
- Param_l2reg: By checking more close range as we did on previous questions, I found 1.118 ~ 1.208 as a good result range
- By combining both, I found param_sigma = 1.204, param_l2reg = 0.109 as an optimal value.
- We can also see that making small change in any one of the hyperparameters in either direction may cause the performance to get worse.

```
df_toshow[df_toshow['param_kernel'] == 'RBF'].sort_values(by='mean_test_score')
```

	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_train_score
4	-	RBF	0.109	-	1.204	0.046384	0.034200
5	-	RBF	0.109	-	1.206	0.046385	0.034261
3	-	RBF	0.109	-	1.202	0.046385	0.034139
6	-	RBF	0.109	-	1.208	0.046385	0.034323
2	-	RBF	0.109	-	1.2	0.046385	0.034077
...	...	...	...	...	...	...	...
42	-	RBF	0.121	-	1.116	0.046909	0.031993

Polynomial kernel: The best hyperparameter for Polynomial kernel:

param_degree = 6, param_l2reg = 800, param_offset = -1

- param_degree: seeking 1 ~ 10, I found 4 ~ 6 as a good parameter range
- param_l2reg: seeking 500~1500, I found 700 ~ 1000 as a good parameter range
- param_offset: seeking -5 ~ 5, I found -1 ~ 1 as a good parameter range

- By combining all, I found param_degree = 6, param_l2reg = 800, param_offset = -1 as an optimal value.
- We can also see that making small change in any one of the hyperparameters in either direction may cause the performance to get worse.

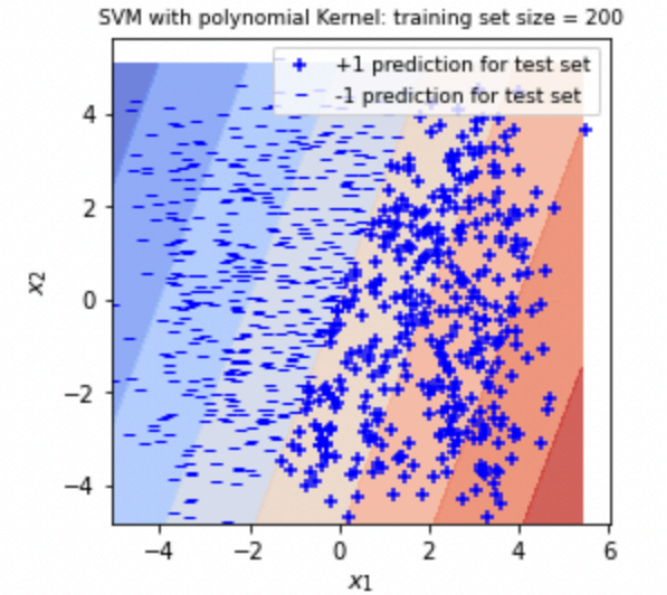
```
df_toshow[df_toshow['param_kernel'] == 'polynomial'].sort_values(by='mean_test_score')
```

	param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_train_score
584	6	polynomial	800.0	-1	-	0.066145	0.065034
595	6	polynomial	900.0	-1	-	0.066289	0.064769
573	6	polynomial	700.0	-1	-	0.066358	0.065856
597	6	polynomial	900.0	1	-	0.066606	0.060945
606	6	polynomial	1000.0	-1	-	0.066671	0.064797
...	...	...	...	...	...	...	...
285	3	polynomial	1400.0	-3	-	0.895573	0.942424

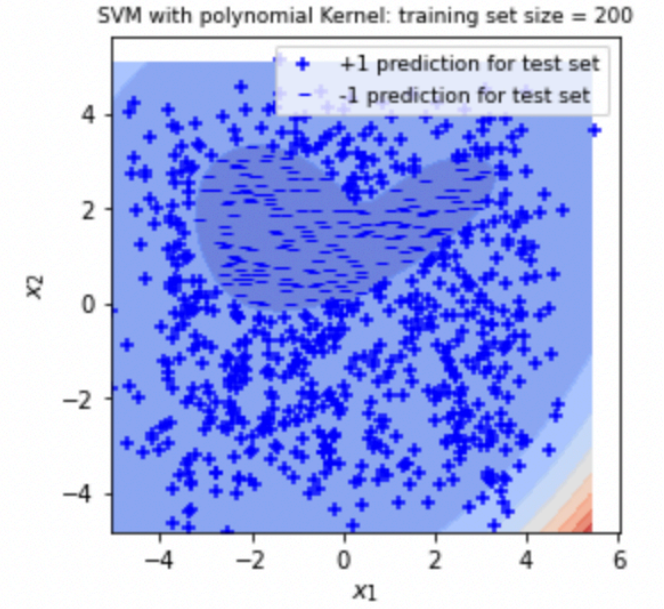
Q35.

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

```
# Linear kernel
# Code to help plot the decision regions
# (Note: This ode isn't necessarily entirely appropriate for the questions asked. So think about what you are doing.

k = functools.partial(linear_kernel)
f = train_soft_svm(x_train, y_train, k, l2_reg=3450)

#determine the decision regions for the predictions
x1_min = min(x_test[:,0])
x1_max= max(x_test[:,0])
x2_min = min(x_test[:,1])
x2_max= max(x_test[:,1])
h=0.1
xx, yy = np.meshgrid(np.arange(x1_min, x1_max, h),
                     np.arange(x2_min, x2_max, h))

Z = f.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

#determine the predictions for the test set
y_bar = f.predict (x_test)
yplus = np.ma.masked_where(y_bar<=0, y_bar)
xplus = x_test[~np.array(yplus.mask)]
yminus = np.ma.masked_where(y_bar>0, y_bar)
xminus = x_test[~np.array(yminus.mask)]

#plot the learned boundary and the predictions for the test set
figsize = plt.figaspect(1)
f, (ax) = plt.subplots(1, 1, figsize=figsize)
decision =ax.contourf(xx, yy, Z, cmap=plt.cm.coolwarm, alpha=0.8)
pluses = ax.scatter (xplus[:,0], xplus[:,1], marker='+', c='b', label = '+1 prediction for test set')
minuses = ax.scatter (xminus[:,0], xminus[:,1], marker=r'$-$', c='b', label = '-1 prediction for test set')
ax.set_ylabel(r"$x_2$", fontsize=11)
ax.set_xlabel(r"$x_1$", fontsize=11)
ax.set_title('SVM with polynomial Kernel: training set size = %s'% len(data_train), fontsize=9)
ax.axis('tight')
ax.legend(handles=[pluses, minuses], fontsize=9)
plt.show()
```



```
# polynomial kernel

k = functools.partial(polynomial_kernel, offset=-1, degree=6)
f = train_soft_svm(x_train, y_train, k, l2_reg=800)

#determine the decision regions for the predictions
x1_min = min(x_test[:,0])
x1_max= max(x_test[:,0])
x2_min = min(x_test[:,1])
x2_max= max(x_test[:,1])
h=0.1
xx, yy = np.meshgrid(np.arange(x1_min, x1_max, h),
                     np.arange(x2_min, x2_max, h))

Z = f.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

#determine the predictions for the test set
y_bar = f.predict (x_test)
yplus = np.ma.masked_where(y_bar<=0, y_bar)
xplus = x_test[~np.array(yplus.mask)]
yminus = np.ma.masked_where(y_bar>0, y_bar)
xminus = x_test[~np.array(yminus.mask)]

#plot the learned boundary and the predictions for the test set
figsize = plt.figaspect(1)
f, (ax) = plt.subplots(1, 1, figsize=figsize)
decision =ax.contourf(xx, yy, Z, cmap=plt.cm.coolwarm, alpha=0.8)
pluses = ax.scatter (xplus[:,0], xplus[:,1], marker='+', c='b', label = '+1 prediction for test set')
minuses = ax.scatter (xminus[:,0], xminus[:,1], marker=r'$-$', c='b', label = '-1 prediction for test set')
ax.set_ylabel(r"$x_2$", fontsize=11)
ax.set_xlabel(r"$x_1$", fontsize=11)
ax.set_title('SVM with polynomial Kernel: training set size = %s'% len(data_train), fontsize=9)
ax.axis('tight')
ax.legend(handles=[pluses, minuses], fontsize=9)
plt.show()
```



```
# RBF kernel
k = functools.partial(RBF_kernel, sigma=1.204)
f = train_soft_svm(x_train, y_train, k, l2_reg=0.109)

#determine the decision regions for the predictions
x1_min = min(x_test[:,0])
x1_max= max(x_test[:,0])
x2_min = min(x_test[:,1])
x2_max= max(x_test[:,1])
h=0.1
xx, yy = np.meshgrid(np.arange(x1_min, x1_max, h),
                     np.arange(x2_min, x2_max, h))

Z = f.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

#determine the predictions for the test set
y_bar = f.predict (x_test)
yplus = np.ma.masked_where(y_bar<=0, y_bar)
xplus = x_test[~np.array(yplus.mask)]
yminus = np.ma.masked_where(y_bar>0, y_bar)
xminus = x_test[~np.array(yminus.mask)]

#plot the learned boundary and the predictions for the test set
figsize = plt.figaspect(1)
f, (ax) = plt.subplots(1, 1, figsize=figsize)
decision =ax.contourf(xx, yy, Z, cmap=plt.cm.coolwarm, alpha=0.8)
pluses = ax.scatter (xplus[:,0], xplus[:,1], marker='+', c='b', label = '+1 prediction for test set')
minuses = ax.scatter (xminus[:,0], xminus[:,1], marker=r'$-$', c='b', label = '-1 prediction for test set')
ax.set_ylabel(r"$x_2$", fontsize=11)
ax.set_xlabel(r"$x_1$", fontsize=11)
ax.set_title('SVM with RBF Kernel: training set size = %s'% len(data_train), fontsize=9)
ax.axis('tight')
ax.legend(handles=[pluses, minuses], fontsize=9)
plt.show()
```