Homework 3: DS GA 1003

Selected Solutions

1 Subgradients

Q1. Suppose $f_1, ..., f_m : \mathbb{R}^d \to \mathbb{R}$ are convex functions, and $f(x) = max_{i=1,...,m}\ f_i(x)$. Let k be any index for which $f_k(x) = f(x)$, and choose $g \in \partial f_k(x)$. Show that $g \in \partial f(x)$

For any $z \in \mathbb{R}^d$, we know that $f(z) \geq f_i(z)\ \forall i$. **... (1)**
This is because we select the maximum of all functions f_i at z to be the value of f at z.

From the question, we choose $g \in \partial f_k(x)$.
So from the subgradient rule we know that $f_k(z) \geq f_k(x) + g^T(z - x)$

But at point x, we know that $f(x) = f_k(x)$. So $f_k(z) \geq f(x) + g^T(z - x)$. **...(2)**

Combining the above to statements, we get: $f(z) \geq f_k(z) \geq f(x) + g^T(z - x)$.
Or $f(z) \geq f(x) + g^T(z - x)$

Q2. Give a subgradient of $J(w) = max\{0, 1 - yw^T x\}$.

This can be written as a particular instance of the previous problem where $J_1(w) = 0$ and $J_2(w) = 1 - yw^T x$ and $J(w) = max_{i=1,2} J_i(w)$. So from our obtained result, we can give a subgradient, g, defined as:

$$g = \begin{cases} 0 & \text{when } yw^T x \geq 1 \\ -yx & \text{when } yw^T x < 1 \end{cases}$$

Q3. 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 $\theta \in (0, 1)$,

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

Let $a, b \in \mathbb{R}^n, \theta \in (0, 1)$. Let $x = \theta a + (1 - \theta)b$.

From the subgradient rule: $f(z) \geq f(x) + g^T(z - x)$.

Putting in a, b as z in this equation:

$f(a) \geq f(x) + g^T(a - x)$ ...(1)

$f(b) \geq f(x) + g^T(b - x)$ ...(2)

Now, we also simplify $a - x$ and $b - x$ as:

$a - x = a - \theta a - (1 - \theta)b = (1 - \theta)(a - b)$

$b - x = b - \theta a - (1 - \theta)b = -\theta(a - b)$

Substituting back into $(1), (2)$:

$f(a) \geq f(x) + (1 - \theta)g^T(a - b) \ \times \theta$ ...(3)

$f(b) \geq f(x) - \theta g^T(a - b) \ \times (1 - \theta)$ ...(4)

Now adding $(3), (4)$:

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

Which is the desired relation we have to show.

2 SVM with Pegasos

Q4. Consider the SVM objective function for a single training point $J_i(w) = \frac{\lambda}{2}||w||^2 + \max\{0, 1 - y_i w^T x_i\}$. The function $J_i(w)$ 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.

The function is not differentiable at the point where the the max function switches between 0 and $1 - y_i w^T x_i$ i.e. at $y_i w^T x_i = 1$ i.e. gradient not defined there.

The gradient can be written as: $\frac{dJ_i(w)}{dw} = \begin{cases} \lambda w & \text{when } y_i w^T x_i > 1 \\ \lambda w - y_i x_i & \text{when } y_i w^T x_i < 1 \\ \text{undefined} & \text{when } y_i w^T x_i = 1 \end{cases}$

Q5. Show that the subgradient is equal to the value obtained in the function.

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

We can write this as: $J_i(w) = \begin{cases} \frac{\lambda}{2}||w||^2 + 0 & \text{when } y_i w^T x_i \geq 1 \\ \frac{\lambda}{2}||w||^2 + 1 - y_i w^T x_i & \text{when } y_i w^T x_i < 1 \end{cases}$

In both cases, the function is a sum of 2 convex functions so we can take the subgradient of each individually and take a sum. Additionally at the point where the margin is 1, we can choose either one of the max function arguments since they are equal, so we include it in the first interval. The subgradient can be written as follows:

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

If step size if $\eta_t = \frac{1}{\lambda t}$, then the stochastic subgradient descent update is:

$w_{new} = w - \eta_t g_w$

$= \begin{cases} w - \eta_t \lambda w & \text{when } y_i w^T x_i \geq 1 \\ w - \eta_t(\lambda w - y_i x_i) & \text{when } y_i w^T x_i < 1 \end{cases}$

$= \begin{cases} w(1 - \eta_t \lambda) & \text{when } y_i w^T x_i \geq 1 \\ w(1 - \eta_t \lambda) + \eta_t y_i x_i & \text{when } y_i w^T x_i < 1 \end{cases}$

which is the same as the update given in the algorithm.

3 Dataset and Sparse Representation

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

```
def construct_bow(wordlist):
    counts = Counter()
    for word in wordlist:
        counts[word]+=1
    #print(counts.most_common(3))
    return counts
```

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

The following code block performs the same. Note that here I prefer to use the term validation i.e. X_val and y_val, since we normally don't have access to the test labels. Additionally function load_and_shuffle already shuffles once so we can just split the data by index.

```
from utils_svm_reviews import *

data = load_and_shuffle_data()
print(len(data))

train_data = data[:1500]
val_data = data[1500:]

print(len(train_data), len(val_data))
X_train = []
y_train = []
for item in train_data:
    X_train.append(construct_bow(item[:-1]))
    y_train.append(int(item[-1]))
    assert abs(y_train[-1]) == 1

X_val = []
y_val = []
for item in val_data:
    X_val.append(construct_bow(item[:-1]))
    y_val.append(int(item[-1]))
    assert abs(y_val[-1]) == 1
```

Q8. Implement the Pegasos algorithm to run on a sparse data representation

The following code block performs the same:

```
def naive_pegasos(X, y, lambda_reg, max_epochs = 1, X_val = None, y_val = None):
    w = {}
    for item in X:
        for key in item.keys():
            if key not in w.keys():
                w[key] = 0

    t = 0

    start = time.time()
    if X_val is not None:
        print("Reporting per epoch error")
        print("Epoch", "Val-Error")
    for epoch in range(max_epochs):
        for i in range(len(X)):
            t+=1
            eta_t = 1/(t*lambda_reg)
            #This increment happens regardless of margin:
            #w_t+1 = (1 - (eta_t * lambda)) * w_t

            #This increment happens only if margin<1
            #w_t+1 = w_t+1 + (eta_t * y_train[i] * X_train[i])
            if y[i] * dotProduct(w, X[i]) < 1:
                increment(w, -eta*lambda_reg, w)
                increment(w, eta_t*y[i], X[i])
            else:
                increment(w, -eta*lambda_reg, w)


        if X_val is not None:
            print(epoch, classification_error(X_val, y_val, w))

    end = time.time()
    naive_w = copy.deepcopy(w)
    print("Average epoch time: ", (end-start)/max_epochs)
    return naive_w
```

Q9. Show that the optimized update is equivalent. Implement the Pegasos algorithm with the (s, W) representation described above

$s_{t+1}W_{t+1} = s_{t+1}(W_t + \frac{1}{s_{t+1}}\eta_t y_i x_i) = (1 - \eta_t\lambda)s_t(W_t + \frac{1}{s_{t+1}}\eta_t y_i x_i) = (1 - \eta_t\lambda)s_t W_t - \eta_t y_i x_i$

And we know that $w = sW$, so substituting that at $t + 1$ and t above, we get:

$w_{t+1} = (1 - \eta_t \lambda) w_t - \eta_t y_i x_i$

Which is the same as the original update.

The following function performs the optimized update. Note that I have a line that checks if 3 consecutive epochs have similar validation errors and I use this as a test for convergence in later questions.

```
def fast_pegasos(X, y, lambda_reg, max_epochs = 25, print_time = False,
        X_val = None, y_val = None):
    #print(lambda_reg)
    w = {}
    for item in X:
        for key in item.keys():
            if key not in w.keys():
                w[key] = 0

    t = 1.0
    s = 1.0
    W = copy.deepcopy(w)
    start = time.time()
    val_errors = []
    for epoch in range(max_epochs):
        for i in range(len(X)):
            t+=1
            eta_t = 1.0/(t*lambda_reg)

            if s == 0:
                s = 1
                W = copy.deepcopy(w)

            if y[i]*s * dotProduct(W, X[i]) < 1:
                s = (1 - eta_t*lambda_reg)*s
                increment(W, (1.0/s)*eta_t*y[i], X[i])
            else:
                s = (1 - eta_t*lambda_reg)*s

        if X_val is not None:
            temp_w = copy.deepcopy(w)
            increment(temp_w, s, W)
            val_errors.append(classification_error(X_val, y_val, temp_w))
            if len(val_errors)>2 and np.isclose(val_errors[-1],
                val_errors[-2], atol=1e-2) and np.isclose(val_errors[-1],
                    val_errors[-3], atol=1e-2):
                break
```

```
        increment(w, s, W)
        #w = {}
        #for key in W.keys():
        #    w[key] = W[key]*s

        end = time.time()

        if print_time:
            print("Average epoch time: ", (end-start)/max_epochs)
        if X_val is not None:
            return
```

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.

To check the same, I run both implementations for 2 epochs with $\lambda = 1$ and compare both the validation errors as well as 5 sampled feature values to see that they correspond. In the output that follows the code we can see that the average epoch time difference is 9.01s vs 0.13s.

Output:

```
Average epoch time: Naive:  9.019188165664673
Average epoch time: Fast: 0.13298046588897705
----------------------------------------
Verifying that they give same validation error:
Naive pegasos, 2 epochs, lambda = 1 -  0.33399999999999996
Fast pegasosm, 2 epochs, lambda = 1 -  0.33399999999999996
Some feature values in both:
Word Naive-Value Fast-Value
capsule -0.004666666666666669 -0.004665111629456861
the 0.0006666666666666832 0.0006664445184939927
running -0.00766666666666666 -0.0076641119626791225
gag -0.009666666666666678 -0.009663445518160636
pair -0.003999999999999995 -0.00399866711096302
of 0.015000000000000128 0.014995001666111342
----------------------------------------
```

We see that the values are almost the same so we can confirm that both implmeentations seem fine.

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 y_i using $sign(w^T x_i)$.

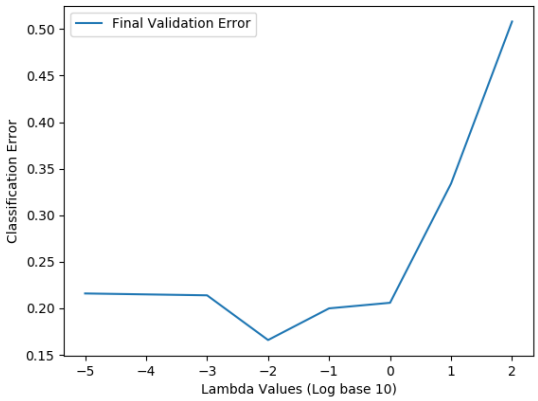

Figure 1: Validation error as a function of log lambda

```
def classification_error(X, y, w):
    preds = []
    for item in X:
        op = dotProduct(w, item)
        if op>0:
            pred = 1
        else:
            pred = -1
        preds.append(pred)
    acc = np.sum([int(preds[i] == y[i]) for i in range(len(preds))])/len(y)
    assert 1-acc == zero_one_loss(y, preds)
    return 1-acc
```

Q12. Search for the regularization parameter that gives the minimal percent error on your test set.

I check values for various λ and evaluate the best validation error and plot the same as a function of λ. Fig. 1 has the required plot. We see the optimal value for validation error at $\lambda = 10^{-2}$

Q13. 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?

The scores are divided on either side of zero so we choose brackets that were diverging from 0 into 10 buckets on either side divided equally from 0 to the maximum in that direction i.e. bucket width was the same in terms of the score, and we selected the width so as to encapsulate all examples into 10 equal width buckets.

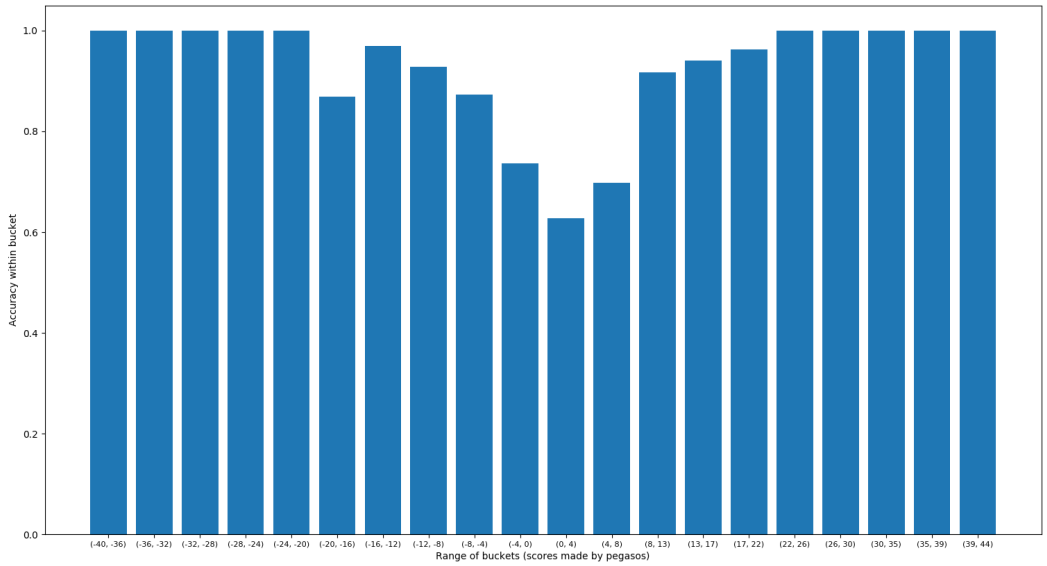

Figure 2: Accuracy of predictions made in each bucket vs bucket limits

In Fig. 2, we see that when scores are very high or very low (high magnitude of score), the accuracy on predictions is nearly perfect (here the y-axis is 1 - classification_error). This is to be expected as the model is essentially very confident about these predictions and it gets them correct almost all the time. Then as we look at predictions closer to zero the model gradually begins to make more errors and the accuracy drops, with the lowest score being in the bucket just on either side of zero or the model is most likely to make an error on an example where it's score has smallest absolute value. In general this seems to be a good trend to see because it tells us that our model is well calibrated and it's sense of score is indicative of a rough confidence in prediction i.e. we can trust a high magnitude score more than a lower maggnitude score. Additionally to note, we also plot Fig. 3 which has the same buckets but instead the y-axis contains the count of predictions in each bucket. We see an opposite trend that fewer examples belong to high magnitude buckets and more examples belong to the buckets close to zero. This is again expected and tells us that there are more examples that the model is less sure about

Q14. Attempt to explain why the model was incorrect. Can you think of a new feature that might be able to fix the issue?

For this, we look at the individual contributions of each feature i.e. word in the bag of words. Attached below is the code and output for 2 examples that the model got wrong. Note that when we sort contributions, we want to see absolute contributions, but because we see the count and weight value we can deduce the sign from the output table.

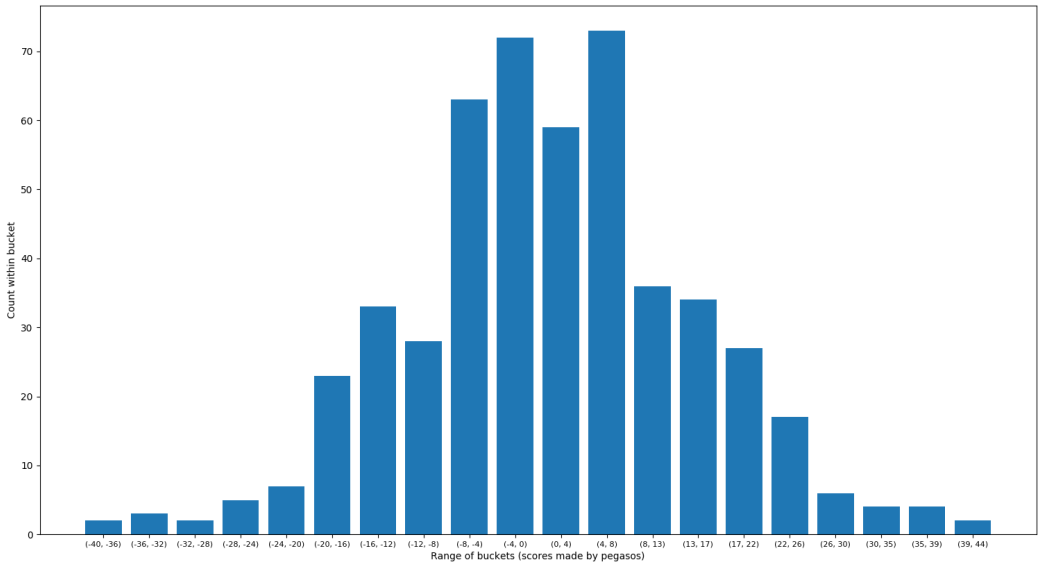

Figure 3: Count of examples in each bucket vs bucket limits

The first thing that we notice from the output is that the heaviest contributions are often from stop words. Perhaps we can exclude these as features since we don't want them to be very indicative of the result. Additionally in the first example, we see that the prediction score was negative but the expected output is positive and the strong negative contributions are coming from words such as "action" and "have". By themselves these words shouldn't perhaps have this connotation since an action movie need not necessarily be bad (that might be something in the dataset). The same goes for the high positive score achieved in the second example. We see that this is because of words like "and" and "the" which shouldn't have so much of an influence. So our first suggestion of not counting stop words is one alternative, or we can look at n-gram features instead where you keep the same bag-of-words model but instead of an individual word, you take each consecutive pair or trigram of words. This might increase your feature size but since we are using a sparse representation anyway this should be ok. The benefit we get is that the contribution from each feature now has context i.e. perhaps "action" can be either "trashy action" or "engrossing action".

Relevant Code:

```
def analyze_example(x_item, w, pred, y_val, k = 5):
    contributions = []
    print("Prediction vs True: ", pred, y_val)
    for xi in x_item.keys():
        if xi not in x_item.keys() or xi not in w.keys():
            continue
        contributions.append((abs(x_item[xi]*w[xi]), xi))
    print("Idx", "Score", "Feature", "Weight", "Count")
```

```
    for i, (score, feature) in enumerate(sorted(contributions, reverse=True)):
        print(i, score, feature, w[feature], x_item[feature])
        if i>k:
            break

assert len(preds) == len(y_val)
tot = 0
for i, pred in enumerate(preds):
    if (pred > 0 and y_val[i] == -1) or (pred<=0 and y_val[i] == 1):
        analyze_example(X_val[i], w_vals[w_idx], pred, y_val[i])
        tot+=1
    if tot == 2:
        break
```

Output of Code as a table:

```
Prediction vs True:  -2.8265410426203226 1
Idx Score Feature Weight Count
0 7.021910137327119 and 0.35109550686635593 20
1 4.639793786942753 action -0.5155326429936392 9
2 4.604239811563945 the 0.12443891382605256 37
3 3.9998222301230166 you 0.44442469223589076 9
4 3.3909604017598656 it's 0.48442291453712366 7
5 3.182080796409003 have -0.7955201991022508 4
6 3.110972845651275 in 0.2222123461179482 14

Prediction vs True:  4.555353095417961 -1
Idx Score Feature Weight Count
0 9.128483178525254 and 0.35109550686635593 26
1 6.595262432780785 the 0.12443891382605256 53
2 5.261988356072976 if -0.657748544509122 8
3 4.888671614594799 you 0.44442469223589076 11
4 4.533131860806086 to -0.15110439536020287 30
5 4.444246922358964 in 0.2222123461179482 20
6 4.266477045464584 he 0.533309630683073 8
```

4 Ridge Regression: Theory

Q15. Show that for w to be a minimizer of $J(w)$, we must have $X^TXw + \lambda Iw = X^Ty$. Show that the minimizer of $J(w)$ is $w = (X^TX + \lambda I)^{-1}X^Ty$. Justify that the matrix $X^TX + \lambda I$ is invertible, for $\lambda > 0$

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

$\frac{dJ(w)}{dw} = 2X^T(Xw - y) + 2\lambda w = 0$ (for minimizer)

$2X^T(Xw - y) + 2\lambda w = 0 \implies 2X^TXw - 2X^Ty + 2\lambda w = 0$

Or, $2X^TXw + 2\lambda w = 2X^Ty \implies X^TXw + \lambda w = X^Ty$

This is the first condition we are required to justify. Additionally in step 2 above, the w we obtain is a minimzer because the second derivative is non-negative $(2X^TX + 2\lambda)$.

Following on, the minimizer of $J(w)$:

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

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

To show that $X^TX + \lambda I$ is invertible, we see if it is a positive definite matrix. Consider a vector $v \in \mathbb{R}^d$ is a non-zero vector:

$v^T(X^TX + \lambda I)v$

$= v^TX^TXv + v^T\lambda Iv$

$= (Xv)^T(Xv) + \lambda v^TIv$

$= ||Xv||^2 + \lambda v^Tv = ||Xv||^2 + \lambda||v||^2 > 0$ if $\lambda > 0$

Q16. Rewrite $X^TXw + \lambda Iw = X^Ty$ as $w = \frac{1}{\lambda}(X^Ty - X^TXw)$. Based on this, show that we can write $w = X^T\alpha$ for some α, and give an expression for α.

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

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

$w = (\frac{1}{\lambda}(X^Ty - X^TXw)) = X^T(\frac{1}{\lambda}(y - Xw))$ (since λ is a scalar and greater than zero)

Therefore the α value which satisfies the equation is $\frac{1}{\lambda}(y - Xw)$

Q17. Based on the fact that $w = X^T\alpha$, explain why we say w is "in the span of the data."

$w = X^T\alpha$, so in the matrix form, each column of $X^T \in \mathbb{R}^{d\times n}$ now corresponds to an example in the data i.e. column 1 is x_1 from the data and so on. We additionally look at α as a single column vector of dimensionality n.
In the matrix multiplication of $X^T\alpha$, then every entry in a particular column of X^T is multiplied by the same index of entry in α or every entry in column 1 is multiplied by α_1 and every entry in column i is multiplied by α_i. As a result every element in x_i is multiplied with α_i and we can express $w = \alpha_1x_1 + \alpha_2x_2 \ldots \alpha_nx_n$

Q18. Show that $\alpha = (\lambda I + XX^T)^{-1}y$. Note that XX^T is the kernel matrix for the standard vector dot product.

$\alpha = \frac{1}{\lambda}(y - Xw)$ (from Q15)

$\lambda\alpha = y - Xw$

$y = \lambda\alpha + Xw = \lambda\alpha + XX^T\alpha$ (substituting from Q15-16)

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

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

Q19. Give a kernelized expression for the Xw, the predicted values on the training points.

Replacing w with $X^T\alpha$ and putting in the formula from Q17:

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

XX^T is the kernel matrix and the above expression only has terms with XX^T, we can replace this with kernel matrix, K as $K(\lambda I + K)^{-1}y$.

Q20. Give an expression for the prediction $f(x) = x^T w^*$ for a new point x, not in the training set

$f(x) = x^T w^* = x^T X^T \alpha^*$ (where X is the training data and x is the new point)

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

If $X \in \mathbb{R}^{n\times d}$ is multiplied by $x \in \mathbb{R}^{d\times 1}$ to give an answer $\in \mathbb{R}^{n\times 1}$. Here each entry of this answer is x is multiplied by each row of X and we know that each such row corresponds to one example x_i from the training data.

So we can call this as $k_x = Xx = \begin{pmatrix} x^T x_1 \\ x^T x_2 \\ \vdots \\ x^T x_n \end{pmatrix} \in \mathbb{R}^{n\times 1}$

Therefore putting this in, we get:

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

5 Kernels and Kernel Machines

Q21. Write functions that compute the RBF kernel $k_{RBF(\sigma)}(x, x') = exp(-||x-x'||^2/2\sigma^2)$ and the polynomial kernel $k_{poly(a,d)}(x, x') = (a+ < x, x' >)^d$. The linear kernel $k_{linear}(x, x') =< x, x' >$, 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 $M_{ij} = k(W_i, Xj)$. 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.

```
def RBF_kernel(X1,X2,sigma):
    matrix = scipy.spatial.distance.cdist(X1, X2, 'sqeuclidean')
    matrix = -matrix/(2*(sigma**2))
    matrix = np.exp(matrix)
    return matrix
    #TODO

def polynomial_kernel(X1, X2, offset, degree):
    matrix = linear_kernel(X1, X2)
    matrix = offset + matrix
    matrix = matrix**degree
    return matrix
```

Q22. Use the linear kernel function defined in the code to compute the kernel matrix on the set of points $x_0 \in D_X = -4, -1, 0, 2$. Include both the code and the output

```
def linear_kernel(X1, X2):
    return np.dot(X1,np.transpose(X2))

ip = np.asarray([[-4], [-1], [0], [2]])
print(ip)
op = linear_kernel(ip, ip)
print(op)
```

Output for above code snippet

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

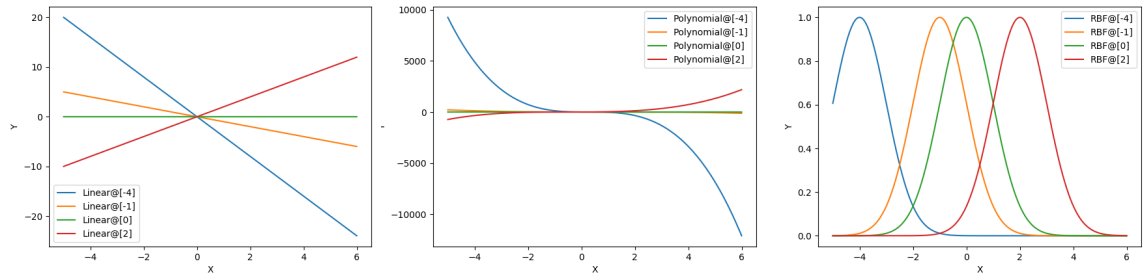

Figure 4: Linear, Polynomial and RBF kernels plotted for $x \in [-6, 6]$

Q23. (a) Plot the set of functions $x \to k_{poly(1,3)}(x_0, x)$ for $x_0 \in D_X$ and for $x \in [-6, 6]$. (b) Plot the set of functions $x \to k_{RBF(1)}(x_0, x)$ for $x_0 \in D_X$ and for $x \in [-6, 6]$.

Fig. 4 has the plots for all three kinds of kernels in the required domain of x. The answer to (a) is the middle figure and (b) is the one on the right.

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 $\alpha_i 1, -1, 1$. Plot the resulting function.

Fig. 5 is the required plot and the following code snippet is what was used to instantiate the class and create the plot using the partial function.

Relevant Code:

```
class Kernel_Machine(object):
    def __init__(self, kernel, training_points, weights):
        self.kernel = kernel
        self.training_points = training_points
        self.weights = weights

    def predict(self, X):
        matrix = self.kernel(X, self.training_points)
        ans = matrix @ self.weights
        return ans

training_points = np.asarray([[-1], [0], [1]])
weights = np.asarray([[1], [-1], [1]])
kernel = functools.partial(RBF_kernel, sigma=1.0)
km = Kernel_Machine(kernel, training_points, weights)

plt.plot(xpts, km.predict(xpts))
plt.xlabel("X")
plt.ylabel("Y")
```

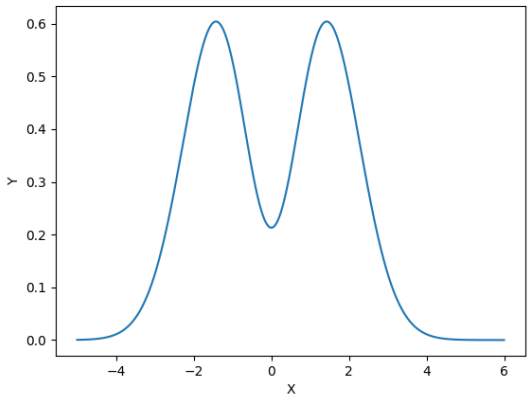

Figure 5: RBF kernel with the given prototype points and corresponding weights for Q23.

```
plt.show()
```

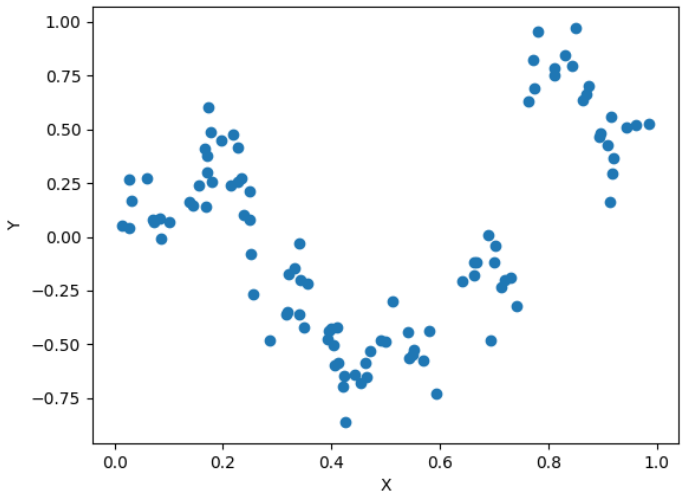

Figure 6: Scatter plot of train points from Q24.

6 Kernel Ridge Regression: In Practise

Q25. Plot the training data. You should note that while there is a clear relationship between x and y, the relationship is not linear.

Fig. 6 is the required scatter plot. The relationship between x and y looks polynomial. Clearly not linear because it seems to have a minimum y-value at around $x \in (0.4, 0.6)$ with the y-value increasing either side of this.

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):
    K = kernel(X, X)
    lambda_I = np.identity(X.shape[0])*l2reg
    inv = np.linalg.inv(K + lambda_I)
    alpha = inv @ 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?

The values of sigma for which we are more likely to see overfitting are the smaller value of 0.01 and you're less likely to see overfitting at 1.0. The sigma parameter controls the width of the gaussian distribution so if we allow for more width then we are reducing the sensitivity to the observed individual data points and hence you're less likely to overfit. Fig. 7 is the required plot that shows us this trend.

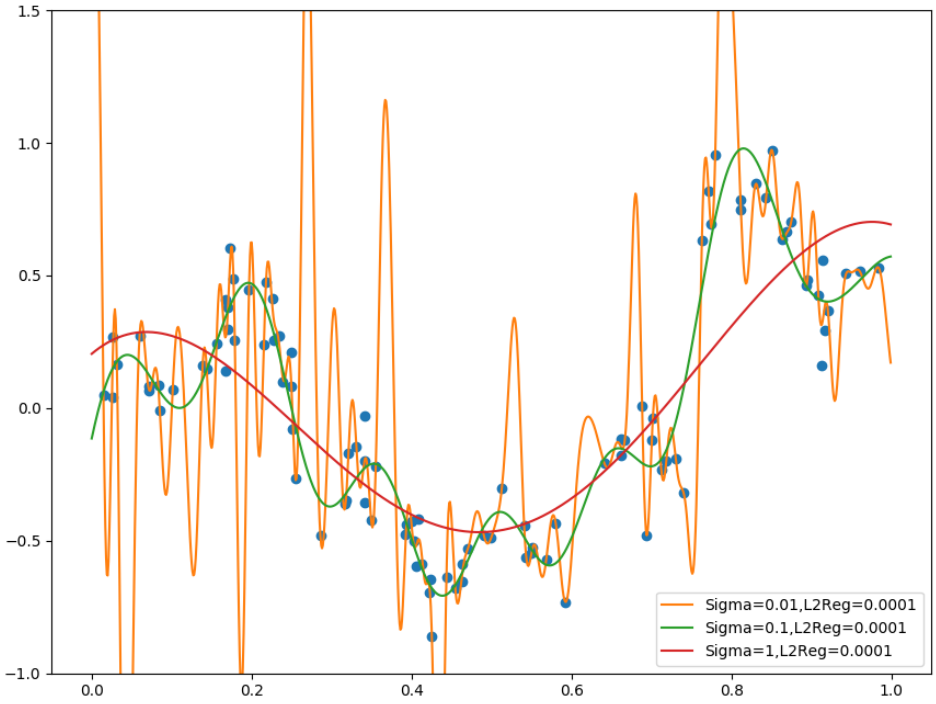

Figure 7: Plot of RBF kernel fits with fixed regularization parameter but varying σ parameter

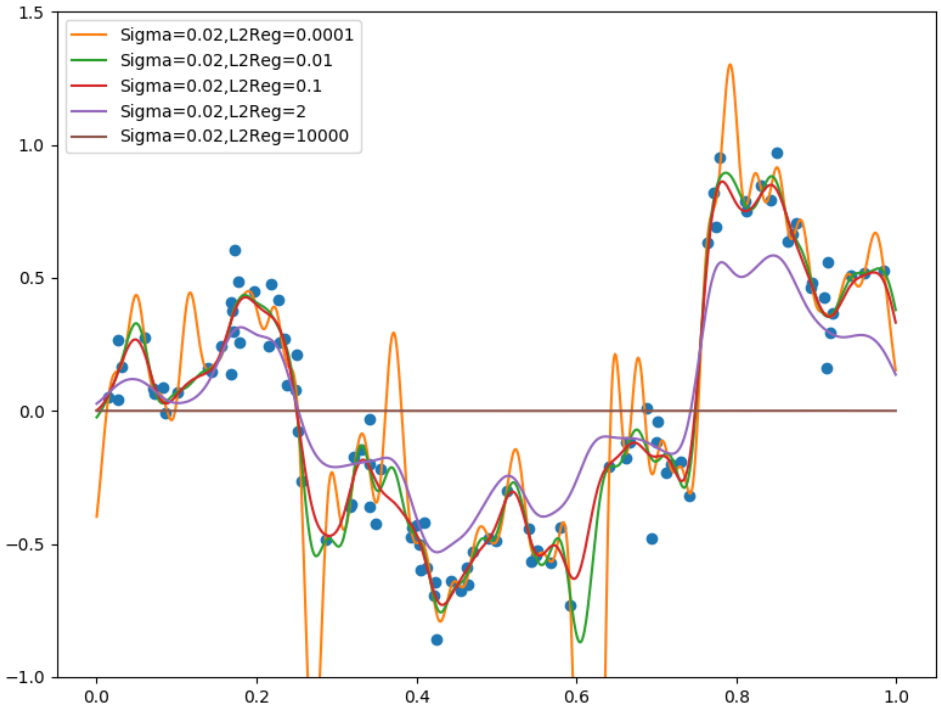

Figure 8: Plot of RBF kernel fits with fixed σ but varying λ regularization parameter

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 : $0.0001, 0.01, 0.1, 2.0$. What happens to the prediction function as $\lambda \to \infty$?

Fig. 8 is the required plot. We see that as we decrease λ, the model expectedly overfits to the data. The sensitivity to the data becomes less and less as we increase the regularization value as we see it's pretty much a straight line for the largest λ. As we increase the lambda parameter to tend to infinity, we can expect a prediction function that is like a straight line at y=0 because you penalize the model very heavily for assigning any weights learnt from the data.

Q29. 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.

We use a grid search varying σ and λ for RBF in the range from (exp(-5.0), exp(5.0)). That same range is also used for λ in the polynomial case, checking degree from 2 to 10, varying offset from -10 to 10. The linear kernel has only λ to vary so we keep the same setting as the other two kernels. The results are plotted in the table below which is sorted by test score (a lower score is more desirable here). We search the dataframe for the rows that are the best

RBF result, the best linear result and the best polynomial result.

Relevant code:

```
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)


lims = [np.exp(i/10.0) for i in range(-50, 50, 5)]
param_grid = [{'kernel': ['RBF'],'sigma':lims, 'l2reg': lims},
              {'kernel':['polynomial'],'offset': [i for i in range(-10, 10)],
                'degree':[i for i in range(2, 10)],'l2reg':lims},
              {'kernel':['linear'],'l2reg': lims}]

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)))

pd.set_option('display.max_rows', 20)
df = pd.DataFrame(grid.cv_results_)
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"])

print(df_toshow)

print("-"*40)
print("Best Linear")
df_polynomial = df_toshow[df_toshow['param_kernel']=='linear']
print(df_polynomial.iloc[0])
## 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
print("-"*40)
print("Best Polynomial")
df_polynomial = df_toshow[df_toshow['param_kernel']=='polynomial']
df_polynomial = df_polynomial.sort_values(by=["mean_test_score"])
print(df_polynomial.iloc[0])
best_l2reg = df_polynomial.iloc[0]['param_l2reg']
best_degree = df_polynomial.iloc[0]['param_degree']
best_offset = df_polynomial.iloc[0]['param_offset']
#offset= 1
#degree = 1
#l2reg = 1
k = functools.partial(polynomial_kernel, offset=best_offset, degree=best_degree)
f = train_kernel_ridge_regression(x_train, y_train, k, l2reg=best_l2reg)
    label = "Polynomial: Offset="+str(best_offset)+",Degree="+str(best_degree)+",
        L2Reg="+str(round(best_l2reg, 4))
plt.plot(xpts, f.predict(xpts), label=label)
#Plot best RBF fit
df_RBF = df_toshow[df_toshow['param_kernel']=='RBF']
print("-"*40)
print("Best RBF")
df_RBF = df_RBF.sort_values(by=["mean_test_score"])
print(df_RBF.iloc[0])
best_l2reg = df_RBF.iloc[0]['param_l2reg']
best_sigma = df_RBF.iloc[0]['param_sigma']
#sigma = 1
#l2reg= 1
k = functools.partial(RBF_kernel, sigma=best_sigma)
f = train_kernel_ridge_regression(x_train, y_train, k, l2reg=best_l2reg)
label = "RBF: Sigma="+str(round(best_sigma, 4))+",L2Reg="+str(round(best_l2reg, 4))
plt.plot(xpts, f.predict(xpts), label=label)
plt.legend(loc = 'best')
plt.ylim(-1,1.75)
plt.show()
```

Code output for Q29:

```
     param_degree param_kernel  param_l2reg param_offset param_sigma
        mean_test_score  mean_train_score
0                  -          RBF     0.006738            -  0.00673795
    0.028267          0.002575
1                  -          RBF     0.006738            -    0.011109
    0.022992          0.003850
2                  -          RBF     0.006738            -   0.0183156
```

```
    0.024086          0.006925
3               -          RBF     0.006738           -   0.0301974
    0.024469          0.008273
4               -          RBF     0.006738           -   0.0497871
    0.018196          0.010426
...           ...           ...          ...          ...         ...                ...
3615            -       linear    12.182494           -           -
    0.164645          0.206874
3616            -       linear    20.085537           -           -
    0.164876          0.207220
3617            -       linear    33.115452           -           -
    0.165263          0.207734
3618            -       linear    54.598150           -           -
    0.165774          0.208368
3619            -       linear    90.017131           -           -
    0.166323          0.209024

[3620 rows x 7 columns]
----------------------------------------
Best Linear
param_degree                -
param_kernel           linear
param_l2reg         0.00673795
param_offset                -
param_sigma                 -
mean_test_score       0.164569
mean_train_score      0.206501
Name: 3600, dtype: object
----------------------------------------
Best Polynomial
param_degree                9
param_kernel        polynomial
param_l2reg         0.00673795
param_offset                -3
param_sigma                 -
mean_test_score       0.030267
mean_train_score      0.039985
Name: 3207, dtype: object
----------------------------------------
Best RBF
param_degree                -
param_kernel              RBF
param_l2reg          0.367879
param_offset                -
param_sigma         0.0497871
```

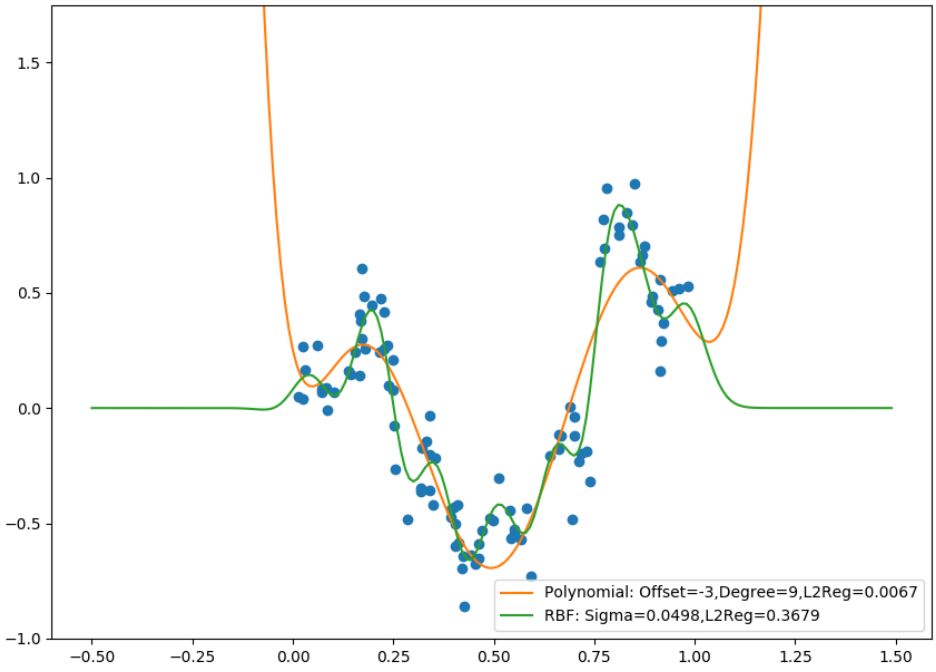

Figure 9: Best fit RBF and Polynomial kernels from the grid search.

```
mean_test_score     0.0138779
mean_train_score    0.0140671
Name: 164, dtype: object
```

Q30. Plot your best fitting prediction functions using the polynomial kernel and the RBF kernel. Comment on the results.

Fig. 9 is the required plot. Both the best polynomial fit and RBF fit seem to fit the data very well but it's interesting to see that just outside the range of [0, 1] the polynomial function values blow up to very high values and the RBF output is basically zero. The training (and test) data is in the [0, 1] range so it is modeled well but our functions are so complicated that they break the second you get a point outside that range. My takeaway is that we've used the kernel methods to find really good fits (very low mean test score) but these become very dataset specific.

Q31. 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 $\text{Var}(\epsilon) = 0.1^2$). The final point is $(x, f(x)+\epsilon)$. What is the Bayes decision function and the Bayes risk for the loss function $l(\hat{y}, y) = (\hat{y} - y)^2$

Bayes Risk, $R(f) = E_{(x,y)\sim P_{\mathcal{X}\times\mathcal{Y}}}[l(f(x), y)] = E[(f(x) - y)^2]$

Bayes Prediction Function, $f^* = \arg\min_f R(f)$

The minimum that the loss term can take is zero i.e. $y = f(x)$, so here the Bayes prediction function can be taken as $E[y|x] = E_x[f(x) + \epsilon] = f(x) + E[\epsilon] = f(x) + 0 = f(x)$

The associated risk is $E[(f(x) - y)^2] = E[(f(x - (f(x) + \epsilon)))^2] = E[\epsilon^2] = Var(\epsilon) - E[\epsilon]^2 = 0.1^2 - 0 = 0.1^2$

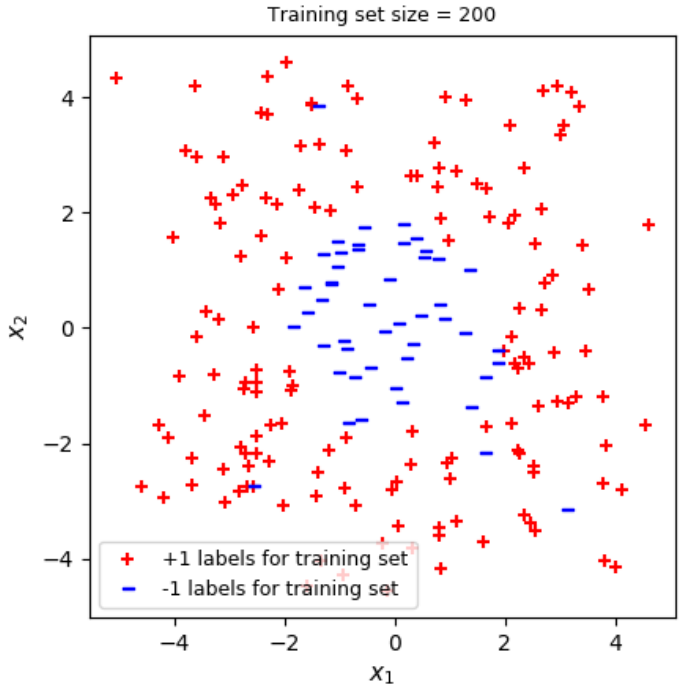

Figure 10: Data for kernelised SVM

7 Kernel SVMs with Kernelised Pegasos

Q31. Are the data linearly separable? Quadratically separable? What if we used some RBF kernel?

Fig. 10 is the plot of the data.

No the data is not linearly separable.

The data seems to resemble a quadratic fit i.e. circle of some radius centered at the origin, but there are some noisy data points (some minuses which are outside this circle) which would make it not quadratically separable. Perhaps if we allow for some softening of the SVM then it would be.

The data should be separable through an RBF kernel.

Q32. Implement kernelized Pegasos.

The relevant code snippet is attached below. We use the same definition of Kernel mahcine and RBF kernels. The output for the following code snippet is the plot Fig. 11.

```
def naive_pegasos(X, y, k, lambda_reg, max_epochs = 3):
    w = np.zeros(len(x_train))
    t = 0

    K = k(x_train, x_train)
```

```
    for epoch in range(max_epochs):
        for i in range(len(K)):
            t+=1
            eta_t = 1/(t*lambda_reg)
            #This increment happens regardless of margin:
            #w_t+1 = (1 - (eta_t * lambda)) * w_t

            #This increment happens only if margin<1
            #w_t+1 = w_t+1 + (eta_t * y_train[i] * X_train[i])
            Ki= K[i, :]
            if y[i] * (Ki @ w) < 1:
                w = w*(1 - eta_t*lambda_reg)
                w[i] += eta_t*y[i]
            else:
                w = w*(1 - eta_t*lambda_reg)

    #end = time.time()
    #naive_w = copy.deepcopy(w)
    #print("Average epoch time: ", (end-start)/max_epochs)
    return Kernel_Machine(k, x_train, w)
    #return naive_w

sigma=1
k = functools.partial(RBF_kernel, sigma=sigma)
f = naive_pegasos(x_train, y_train, k, 0.1)
```

Q33. Find the best hyperparameter settings (including kernel parameters and the regularization parameter) for each of the kernel types.

We use the same search parameters as in Q29. The following code snippet and relevant output

Code for Q33.

```
test_fold = [-1]*len(x_train) + [0]*len(x_test)
predefined_split = PredefinedSplit(test_fold=test_fold)

lims = [np.exp(i/10.0) for i in range(-50, 50, 5)]
#"""
param_grid = [{'kernel': ['RBF'],'sigma':lims, 'l2reg': lims},
              {'kernel':['polynomial'],'offset': lims, 'degree':[i for i in
                range(2, 10)],'l2reg':lims},
              {'kernel':['linear'],'l2reg': lims}]

kernel_svm_estimator = KernelSVM()
grid = GridSearchCV(kernel_svm_estimator,
```

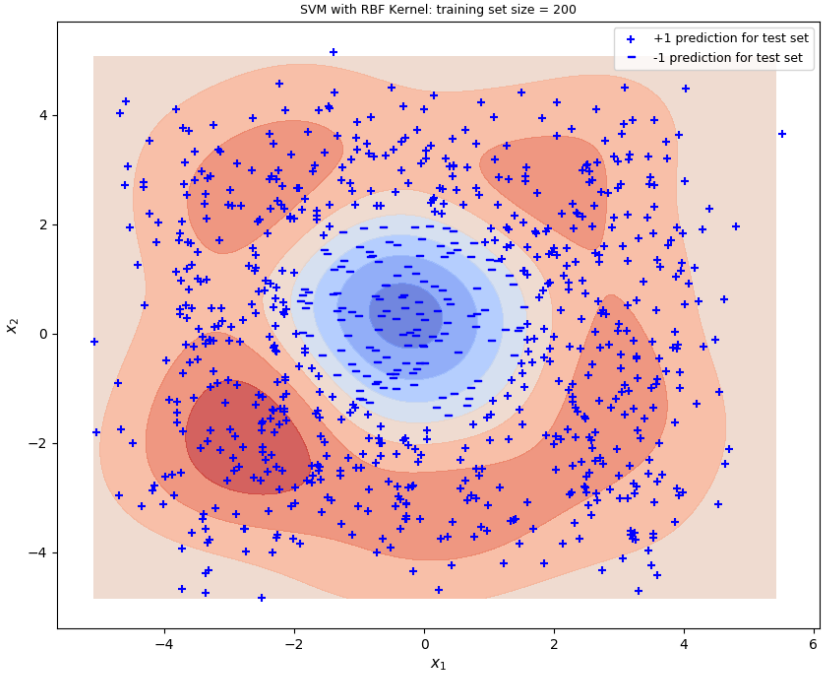

Figure 11: Output of kernelised pegasos

```
                    param_grid,
                    cv = predefined_split,
                    #scoring = make_scorer(zero_one_loss,
                        greater_is_better = True),
                    return_train_score = True
                   )
grid.fit(np.vstack((x_train, x_test)),np.vstack((y_train_01, y_test_01)))

pd.set_option('display.max_rows', 60)
df = pd.DataFrame(grid.cv_results_)
df['mean_test_score'] = df['mean_test_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 = df_toshow.sort_values(by=["mean_test_score"])

print(df_toshow)

############################################################

df_RBF = df_toshow[df_toshow['param_kernel']=='RBF']
```

```
df_RBF = df_RBF.sort_values(by=["mean_test_score"])
print(df_RBF.iloc[0])
df_linear = df_toshow[df_toshow['param_kernel']=='linear']
df_linear = df_linear.sort_values(by=["mean_test_score"])
print(df_linear.iloc[0])
df_polynomial = df_toshow[df_toshow['param_kernel']=='polynomial']
df_polynomial = df_polynomial.sort_values(by=["mean_test_score"])
print(df_polynomial.iloc[0])
```

Output for Q33

```
    param_degree param_kernel  param_l2reg param_offset param_sigma
        mean_test_score  mean_train_score
9               -          RBF     0.006738            -    0.606531
        0.13875            0.110
29              -          RBF     0.011109            -    0.606531
        0.14625            0.100
49              -          RBF     0.018316            -    0.606531
        0.17625            0.100
70              -          RBF     0.030197            -           1
        0.18000            0.190
90              -          RBF     0.049787            -           1
        0.18875            0.190
..            ...          ...          ...          ...         ...
        ...                ...
275             -          RBF     4.481689            -     12.1825
        0.78000            0.760
276             -          RBF     4.481689            -     20.0855
        0.78000            0.760
365             -          RBF    54.598150            -    0.082085
        0.78000            0.760
192             -          RBF     0.606531            -     2.71828
        0.86875            0.850
213             -          RBF     1.000000            -     4.48169
        0.88625            0.865

[3620 rows x 7 columns]
----------------------------------------
Best RBF
param_degree                -
param_kernel              RBF
param_l2reg         0.00673795
param_offset                -
param_sigma          0.606531
mean_test_score       0.13875
mean_train_score          0.11
```

```
Name: 9, dtype: object
800 800 [-1.29354088  0.79411502  1.20351893  1.23467384  1.03818868  1.06192616
  0.76604789 -1.18799504  0.96796686  0.77903898]
624 176
----------------------------------------
Best linear
param_degree                -
param_kernel           linear
param_l2reg          0.367879
param_offset                -
param_sigma                 -
mean_test_score        0.4925
mean_train_score         0.48
Name: 3608, dtype: object
800 800 [-0.071308   -0.48839284 -0.25650981 -0.10003534  0.40858588 -0.15007599
 -0.1296071   0.16179653 -0.35013404 -0.36428446]
624 176
----------------------------------------
Best polynomial
param_degree                  9
param_kernel         polynomial
param_l2reg           0.0497871
param_offset            90.0171
param_sigma                   -
mean_test_score            0.22
mean_train_score           0.24
Name: 3299, dtype: object
624 176
```

Q34. Plot your best fitting prediction functions using the linear, polynomial, and the RBF kernel. The code provided may help.

Fig. 12, Fig. 13 and Fig. 14 are the best RBF, linear and polynomial fits. The relevant parameters are present in the output of Q33.

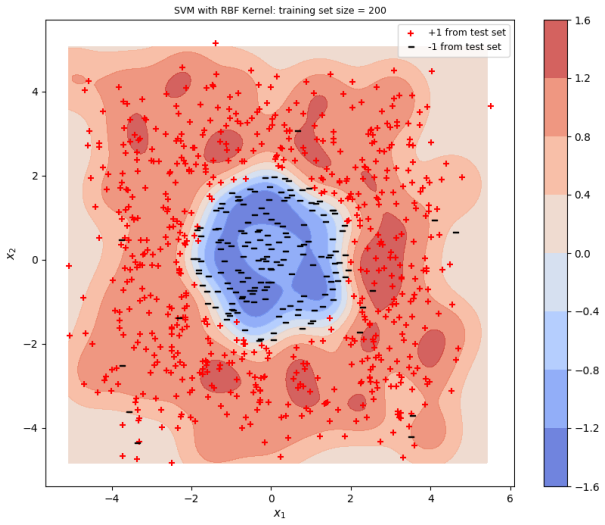

Figure 12: Best RBF fit

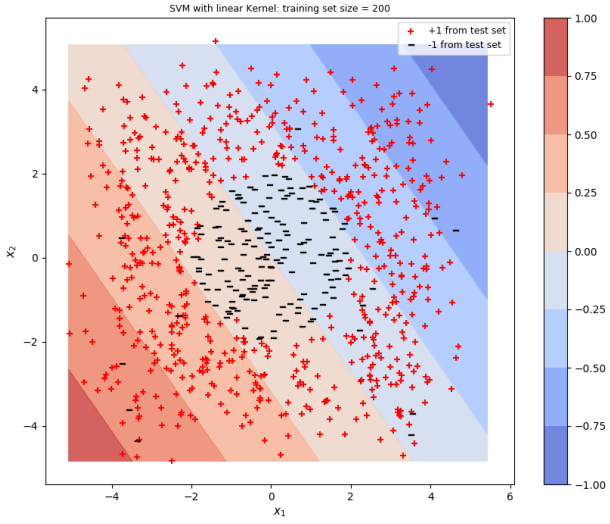

Figure 13: Best linear fit

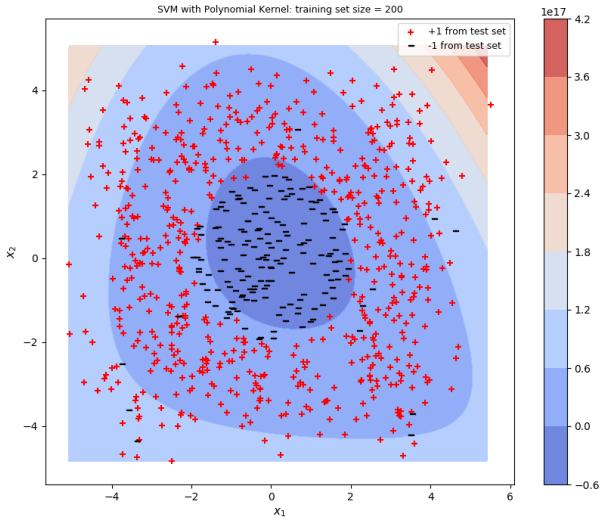

Figure 14: Best polynomial fit