1003 HW3

Long Chen
lc3424@nyu.edu

March 14th, 2021

Q1

For $\forall g \in \partial f_k(x)$,

$$f_k(z) \geq f_k(x) + g^T(z - x) \tag{1}$$

Since $f_k(z) <= f(z)$ for $\forall k$ and $f_k(x) = f(x)$,

$$f(z) \geq f(x) + g^T(z - x) \tag{2}$$

Q2

$$\partial J(w) = \begin{cases} 0 & \text{if } yw^T x \geq 1 \\ \text{-yx} & \text{otherwise.} \end{cases} \tag{3}$$

Q3

Gradient undefined when $y_i w^T x_i = 1$.
For $y_i w^T x_i < 1$,

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

$$\Delta J_i(w) = \lambda w - y_i x_i \tag{5}$$

For $y_i w^T x_i > 1$,

$$J_i(w) = \frac{\lambda}{2}\|w\|^2 \tag{6}$$

$$\Delta J_i(w) = \lambda w \tag{7}$$

Q4

$J_i(w)$ is convex and continuous for $y_i w^T x_i \neq 1$. So the gradients derived from Q3 for the two domain are also subgradients for $J_i(w)$.

Now consider $y_i w^T x_i = 1$, for $\forall \alpha \in [0, 1]$,

$$\begin{aligned} \partial J_i(w) &= \lambda w + \partial \left[\alpha 0 + (1-\alpha)(1 - y_i w^T x_i)\right] && (8) \\ &= \lambda w + \partial \left[(1-\alpha)(1 - y_i w^T x_i)\right] && (9) \\ &= \lambda w + (1-\alpha)(y_i x_i) && (10) \end{aligned}$$

Setting $\alpha = 1$,

$$\partial J_i(w) = \lambda w \tag{11}$$

Therefore, we conclude that:

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

Q5

```
def sparse_representation(list_of_words):
    return Counter(list_of_words)
```

Q6

```
def train_test_split():
    data = load_and_shuffle_data()
    X_train = data[:1500]
    y_train = [l[-1] for l in X_train]
    X_train = [sparse_representation(l[:-1]) for l in X_train]

    X_test = data[1500:]
    y_test = [l[-1] for l in X_test]
    X_test = [sparse_representation(l[:-1]) for l in X_test]

    return X_train, y_train, X_test, y_test


X_train, y_train, X_test, y_test = train_test_split()
```

Q7

```python
def Q7_pegasos(X_train, y_train, lamb, epoch=10, shuffle=False, verbose=False, eval_step=10):
    assert lamb > 0

    X, Y = X_train, y_train
    w, t, dim = {}, 1, len(Y)

    for i in range(epoch):
        if shuffle:
            temp = list(zip(X, Y))
            random.shuffle(temp)
            X, Y = zip(*temp)
            del temp

        for j in range(dim):
            x, y = X[j], Y[j]
            t += 1
            eta = 1 / (t * lamb)
            coef = 1 - eta * lamb

            for k, v in w.items():
                w[k] = coef * v

            if y * dotProduct(w, x) < 1:
                for k, v in x.items():
                    w[k] = w.get(k, 0) + v * eta * y

        if verbose and not i % eval_step:
            print('epoch: {} with classification error: {}'.format(i, classification_error(w, X, Y)))

    return w
```

Q8

$$W_{t+1} = \frac{w_{t+1}}{s_{t+1}} \tag{12}$$

$$= \frac{(1-\eta_t\lambda)w_t + \eta_t y_j x_j}{s_{t+1}} \tag{13}$$

Since $s_t = s_t W_t$ and $s_{t+1} = (1-\eta_t\lambda)s_t$,

$$(9) = \frac{(1-\eta_t\lambda)s_t W_t + \eta_t y_j x_j}{s_{t+1}} \tag{14}$$

$$= W_t + \frac{1}{s_{t+1}}\eta_t y_j x_j. \tag{15}$$

Q9

```
def Q8_pegasos(X_train, y_train, lamb, epoch=10, shuffle=False,
    verbose=False, eval_step = 10):
    assert lamb > 0

    X, Y = X_train, y_train
    w, s, t, dim = {}, 1, 1, len(Y)

    for i in range(epoch):
        if shuffle:
            temp = list(zip(X, Y))
            random.shuffle(temp)
            X, Y = zip(*temp)
            del temp

        for j in range(dim):
            x, y = X[j], Y[j]
            t += 1
            eta = 1 / (t * lamb)
            s *= 1 - eta * lamb

            if y * dotProduct(w, x) < 1:
                for k, v in x.items():
                    w[k] = w.get(k, 0) + v * eta * y / s

        if verbose and not i % eval_step:
            res = {}
            increment(res, s, w)
            print('epoch: {} with classification error: {}'.format(
    i, classification_error(res, X, Y)))

    res = {}
    increment(res, s, w)

    return res
```

Pegasos before optimization with epoch=20: 95.60s
Pegasos after optimization with epoch=20: 1.87s

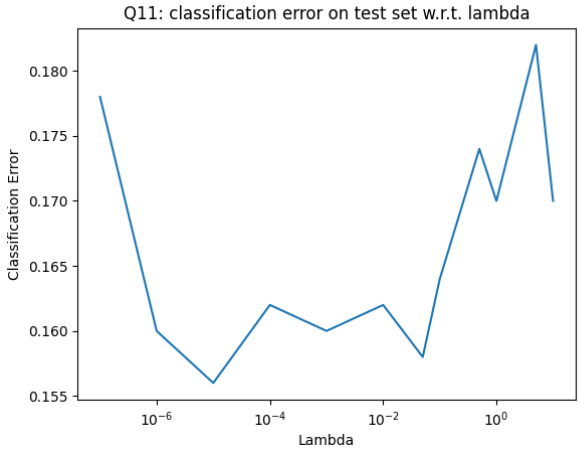
Note: For the optimized pegasos algorithm, w is not calculated for every epoch. Instead, it is calculated at the end of training process, or every eval_step epoch (default=10) if verbose=True.

Q10

```
def classification_error(w, X, y):
    res = 0
    for i in range(len(y)):
        if y[i] * dotProduct(X[i], w) > 0:
            res += 1
    return 1 - res / len(y)
```

Q11

```
lambdas = [1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 5e-2, 1e-1, 5e-1, 1,
     5, 10]
model_errors = []
for lamb in lambdas:
    w = Q8_pegasos(X_train=X_train, y_train=y_train, lamb=lamb,
    epoch=20, shuffle=True)
    model_errors.append(classification_error(w, X_test, y_test))
print(model_errors)

print('Best lambda = {}, with classification error = {}'.format(
    lambdas[model_errors.index(min(model_errors))], min(
    model_errors)))

plt.xscale('log')
plt.plot(lambdas, model_errors)
plt.ylabel('Classification Error')
plt.xlabel('Lambda')
plt.title('Q11: classification error on test set w.r.t. lambda')
plt.savefig('figures/Q11')
plt.show()
```



Best lambda = 1e-05, with classification error = 0.1560

Q12

Comment: when the predicted value are of higher magnitude, the classification is highly accurate. Those who have low magnitude of predicted value tend to predict wrongly.

```
w = Q8_pegasos(X_train=X_train, y_train=y_train, lamb=1e-5, epoch
    =20, shuffle=True)
y_pred = [dotProduct(w, xi) for xi in X_test]
y_ypred = [(y_test[i], y_pred[i]) for i in range(len(y_pred))]
y_ypred.sort(key=lambda x:x[1])
```

Min	Max	False	Observations	Classification Error
$-\infty$	-10000	3	63	0.048
-10000	-1000	27	165	0.164
-1000	-100	13	21	0.619
-100	0	2	2	1
0	100	4	4	1
100	1000	9	13	0.692
1000	10000	31	159	0.195
10000	∞	0	73	0

Table 1: Classification error by binning observation predictions into groups.

```
bins = [float('-inf'), -10000, -1000, -100, 0, 100, 1000, 10000,
    float('inf')]
stat_false, stat_count = [0 for _ in range(len(bins))], [0 for _ in
     range(len(bins))]

i = 0

for y, yp in y_ypred:
    if yp > bins[i+1]:
        print(yp)
        i += 1
    if not y * yp > 0: # False prediction
        stat_false[i] += 1
    stat_count[i] += 1

for i in range(len(stat_count)-1):
    print('Point of score between {} and {} has classification
    error={:.3f} with num_of_sample={}, false={}'.format(bins[i],
    bins[i+1], stat_false[i] / stat_count[i], stat_count[i],
    stat_false[i]))
```

Q13*

See the following two examples. A great amount of feature with high weights (or high $w_i x_i$) are frequent words (or stopwords) with very neutral meanings. A new feature could be the frequency of the feature (word). Or we can use TF-IDF enbedding so that stopwords are of less importance.

	feature_name	feature_value	feature_weight	product
15	and	12	2.766574e+02	3.319889e+03
91	this	9	-2.066598e+02	-1.859938e+03
28	the	25	6.666444e+01	1.666611e+03
71	on	5	-2.999900e+02	-1.499950e+03
168	bad	1	-1.499950e+03	-1.499950e+03
92	will	4	3.666544e+02	1.466618e+03
21	even	3	-4.399853e+02	-1.319956e+03
31	who	6	2.133262e+02	1.279957e+03
136	movie	5	-2.499917e+02	-1.249958e+03
244	if	2	-4.333189e+02	-8.666378e+02
193	plot	2	-4.033199e+02	-8.066398e+02
73	in	13	5.999800e+01	7.799740e+02
109	though	2	3.866538e+02	7.733076e+02
51	is	10	7.666411e+01	7.666411e+02
174	good	2	3.233226e+02	6.466451e+02
238	great	1	6.433119e+02	6.433119e+02
29	two	2	-3.199893e+02	-6.399787e+02
45	most	1	6.399787e+02	6.399787e+02
199	nothing	1	-6.133129e+02	-6.133129e+02
202	character	2	-3.033232e+02	-6.066464e+02
24	he	2	2.999900e+02	5.999800e+02
101	an	2	-2.999900e+02	-5.999800e+02
11	people	2	2.966568e+02	5.933136e+02
69	her	5	-1.133296e+02	-5.666478e+02
200	about	3	-1.799940e+02	-5.399820e+02
97	i	5	-1.066631e+02	-5.333156e+02
50	rush	4	1.299957e+02	5.199827e+02
246	have	2	-2.566581e+02	-5.133162e+02

	feature_name	feature_value	feature_weight	product
45	and	9	2.766574e+02	2.489917e+03
46	the	29	6.666444e+01	1.933269e+03
17	see	3	5.499817e+02	1.649945e+03
1	is	20	7.666411e+01	1.533282e+03
287	bad	1	-1.499950e+03	-1.499950e+03
47	who	7	2.133262e+02	1.493284e+03
42	this	7	-2.066598e+02	-1.446618e+03
66	most	2	6.399787e+02	1.279957e+03
15	you	3	4.199860e+02	1.259958e+03
14	on	4	-2.999900e+02	-1.199960e+03
56	or	5	-2.399920e+02	-1.199960e+03
260	actually	3	-3.499883e+02	-1.049965e+03
103	her	8	-1.133296e+02	-9.066364e+02
113	to	19	-4.666511e+01	-8.866371e+02
8	as	15	5.666478e+01	8.499717e+02
276	well	2	4.066531e+02	8.133062e+02
19	plot	2	-4.033199e+02	-8.066398e+02
131	at	3	-2.433252e+02	-7.299757e+02
36	in	12	5.999800e+01	7.199760e+02
135	could	1	-6.933102e+02	-6.933102e+02
94	than	3	2.299923e+02	6.899770e+02
67	their	4	1.633279e+02	6.533116e+02
76	done	2	3.266558e+02	6.533116e+02
139	director	1	-6.266458e+02	-6.266458e+02
71	nothing	1	-6.133129e+02	-6.133129e+02
183	some	2	-3.033232e+02	-6.066464e+02
174	he	2	2.999900e+02	5.999800e+02
26	into	3	-1.966601e+02	-5.899803e+02
78	better	2	-2.833239e+02	-5.666478e+02

Q14

$$\Delta J(w) = 2X^T(Xw - y) + 2\lambda Iw = 0 \tag{16}$$

$$X^T Xw + \lambda Iw = X^T y \tag{17}$$

Thus,

$$(X^T X + \lambda I)w = X^T y \tag{18}$$

$$w^* = (X^T X + \lambda I)^{-1} X^T y \tag{19}$$

For any non-zero $v \in \mathbb{R}^d$,

$$v^T(X^T X + \lambda I)v = v^T X^T Xv + \lambda v^T v \tag{20}$$

$$= \langle Xv, Xv\rangle + \lambda\langle v, v\rangle \tag{21}$$

Since v is non-zero, we have $\langle v, v\rangle > 0$. Also $\langle Xv, Xv\rangle \geq 0$. Thus for any $\lambda > 0$,

$$(17) > 0 \tag{22}$$

That is, $(X^T X + \lambda I)$ is positive definite and thus $(X^T X + \lambda I)$ is invertible.

Q15

$$w = X^T \left(\frac{1}{\lambda}(y - Xw)\right) \tag{23}$$

Thus,

$$\alpha = \frac{1}{\lambda}(y - Xw) \tag{24}$$

Q16

Since,

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

$$w = X^T \alpha \tag{25}$$

$$= \sum_{i=1}^{n} x_i \alpha_i \tag{26}$$

That is, w is a linear combination of X^T's columns. Therefore we say w is "in the span of the data".

Q17

Since $w = X^T\alpha$,

$$\alpha = \frac{1}{\lambda}(y - Xw) \tag{27}$$
$$= \frac{1}{\lambda}(y - XX^T\alpha) \tag{28}$$

$$\lambda\alpha I = y - XX^T\alpha \tag{29}$$

$$(\lambda I + XX^T)\alpha = y \tag{30}$$

$$\alpha = (\lambda I + XX^T)^{-1}y \tag{31}$$

Q18

Let $K = XX^T$. Thus,

$$Xw = XX^T(\lambda I + XX^T)^{-1}y \tag{32}$$
$$= K(\lambda I + K)^{-1}y \tag{33}$$

Q19

Let,

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

Thus,

$$f(x) = x^T w^* \tag{34}$$
$$= x^T X^T \alpha^* \tag{35}$$
$$= k_x^T \alpha^* \tag{36}$$

Q20

```
def RBF_kernel(X1,X2,sigma):
    return np.exp((-1/(2 * np.power(sigma, 2))) * scipy.spatial.
    distance.cdist(X1, X2, 'sqeuclidean'))

def polynomial_kernel(X1, X2, offset, degree):
    return np.power((offset + np.dot(X1, X2.T)), degree)
```

Q21

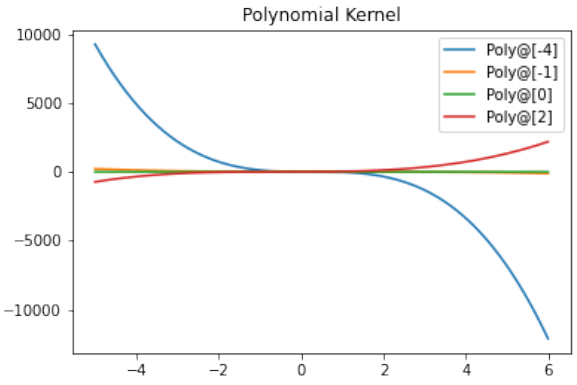
```
prototypes = np.array([-4,-1,0,2]).reshape(-1,1)
# Linear kernel
y = linear_kernel(prototypes, prototypes)

print(y)
```

Output:

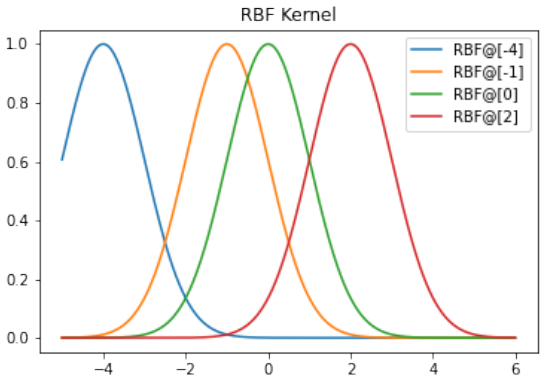
[[16 4 0 -8]
[4 1 0 -2]
[0 0 0 0]
[-8 -2 0 4]]

Q22

a)



b)



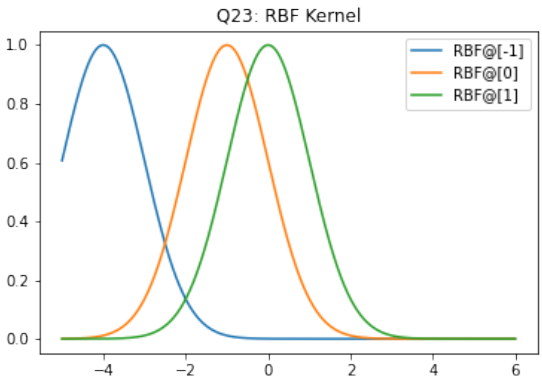
Code

```
plot_step = .01
xpts = np.arange(-5.0, 6, plot_step).reshape(-1,1)
prototypes = np.array([-4,-1,0,2]).reshape(-1,1)

# Poly kernel
y = polynomial_kernel(prototypes, xpts, 1, 3)
for i in range(len(prototypes)):
    label = "Poly@"+str(prototypes[i,:])
    plt.plot(xpts, y[i,:], label=label)
plt.legend(loc = 'best')
plt.title('Polynomial Kernel')
plt.savefig('figures/Q22_poly.png')
plt.show()


# RBF kernel
y = RBF_kernel(prototypes, xpts, 1)
for i in range(len(prototypes)):
    label = "RBF@"+str(prototypes[i,:])
    plt.plot(xpts, y[i,:], label=label)
plt.legend(loc = 'best')
plt.title('RBF Kernel')
plt.savefig('figures/Q22_rbf.png')
plt.show()
```

Q23

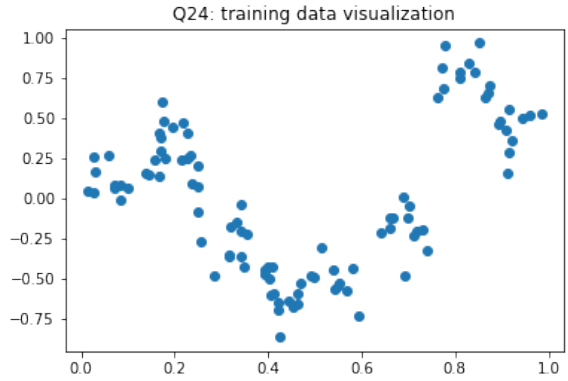
```
class Kernel_Machine(object):
    def __init__(self, kernel, training_points, weights):
        self.kernel = kernel
        self.training_points = training_points
        self.weights = weights

```

```
    def predict(self, X):
        Ker = self.kernel(self.training_points, X)

        return np.matmul(Ker.T, self.weights)

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

Q23_res = km.predict(X=np.arange(-5.0, 6, .01).reshape(-1,1))

for i in range(len(np.array([-1, 0, 1]).reshape(-1,1))):
    label = "RBF@"+str(np.array([-1, 0, 1]).reshape(-1,1)[i,:])
    plt.plot(xpts, y[i,:], label=label)
plt.legend(loc = 'best')
plt.title('RBF Kernel')
plt.savefig('figures/Q23.png')
plt.show()
```



Q24

```
plt.scatter(x_train, y_train)
plt.legend(loc = 'best')
plt.title('Q24: training data visualization')
plt.savefig('figures/Q24png')
plt.show()
```

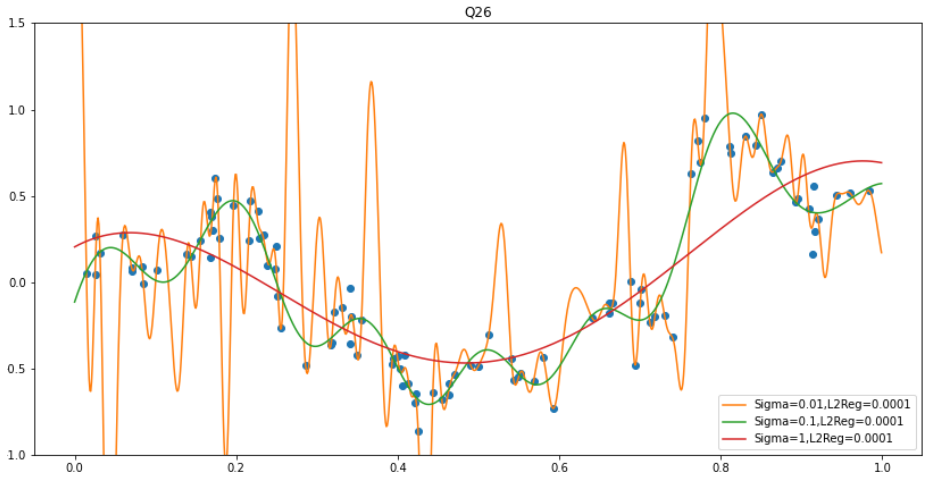


Q25

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

    return Kernel_Machine(kernel, X, alpha)
```

Q26

```
plt.figure(figsize=(20, 10))
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.savefig('figures/Q26.png')
plt.show()
```

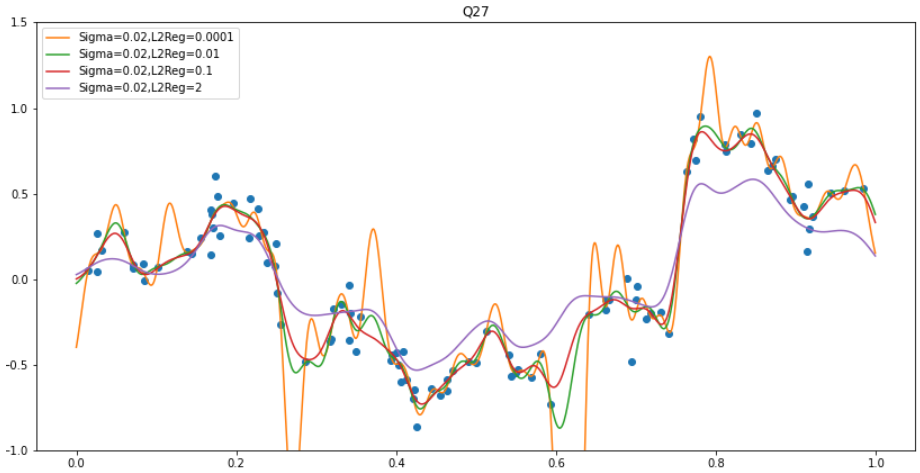


Observation

Small sigmas (e.g. $\sigma = 0.01$) tend to overfit, while large sigmas (e.g. $\sigma = 1$) are less likely to overfit (or in other words, more likely to underfit).

Q27

```
plt.figure(figsize=(14, 7))
plot_step = .001
xpts = np.arange(0 , 1, plot_step).reshape(-1,1)
plt.plot(x_train,y_train,'o')
sigma= .02
for l2reg in [.0001,.01,.1,2]:
    k = functools.partial(RBF_kernel, sigma=sigma)
    f = train_kernel_ridge_regression(x_train, y_train, k, l2reg=
    l2reg)
    label = "Sigma="+str(sigma)+",L2Reg="+str(l2reg)
    plt.plot(xpts, f.predict(xpts), label=label)
plt.legend(loc = 'best')
plt.ylim(-1,1.5)
plt.title('Q27')
plt.savefig('figures/Q27.png')
plt.show()
```



Observation

As $\lambda \longrightarrow \infty$, the prediction function tends to be smoother, leading to less overfitting.

Q28

Best performance for each kernel is reported in Table. 2.

Kernel	deg	l2_reg	offset	sigma	Mean test score	Mean train score
Poly	4	0.01	-1	-	0.0435	0.0601
RBF	**-**	**0.0625**	**-**	**0.1**	**0.0212**	**0.0232**
Linear	-	1	-	-	0.165	0.207

Table 2: Best performance for each kernel. Overall RBF kernel achieves best test score with 0.021270.

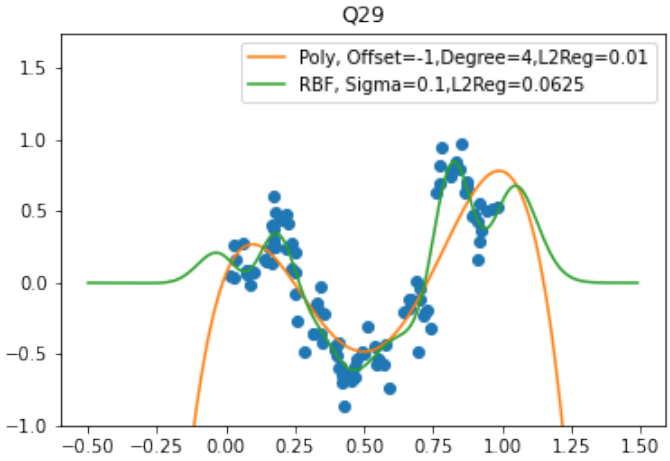
From the three screenshots, one could observe deterioration of performance with change from best hyperparameters.

param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_train_score
-	linear	1.00	-	-	0.164540	0.206506
-	linear	0.01	-	-	0.164569	0.206501
-	linear	10.00	-	-	0.164591	0.206780

param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_train_score
4	polynomial	0.01	-1	-	0.043454	0.060135
4	polynomial	0.01	1	-	0.060262	0.088844
2	polynomial	0.10	-1	-	0.065554	0.098913
2	polynomial	0.01	1	-	0.066532	0.097785
2	polynomial	0.01	-1	-	0.066915	0.097706
2	polynomial	0.10	1	-	0.067454	0.103356
3	polynomial	0.10	1	-	0.067508	0.097546
3	polynomial	0.01	-1	-	0.068156	0.097027
4	polynomial	0.10	1	-	0.068353	0.096693
3	polynomial	0.10	-1	-	0.068397	0.099992
3	polynomial	0.01	1	-	0.068927	0.096913
4	polynomial	10.00	1	-	0.107039	0.147362
3	polynomial	10.00	1	-	0.126708	0.168293
4	polynomial	0.10	0	-	0.130804	0.168819
4	polynomial	0.01	0	-	0.130907	0.168813
4	polynomial	10.00	0	-	0.138611	0.180633
3	polynomial	0.10	0	-	0.138927	0.178269
3	polynomial	0.01	0	-	0.138973	0.178267
4	polynomial	10.00	-1	-	0.139007	0.182572
3	polynomial	10.00	0	-	0.143452	0.185106
2	polynomial	10.00	1	-	0.146247	0.188364
2	polynomial	0.10	0	-	0.151723	0.192543
2	polynomial	0.01	0	-	0.151737	0.192542
2	polynomial	10.00	0	-	0.153268	0.195033
4	polynomial	0.10	-1	-	0.188090	0.148359
2	polynomial	10.00	-1	-	0.202746	0.239150
3	polynomial	10.00	-1	-	0.214186	0.256200

param_degree	param_kernel	param_l2reg	param_offset	param_sigma	mean_test_score	mean_train_score
-	RBF	0.0625	-	0.1	0.021270	0.023245
-	RBF	0.1250	-	0.1	0.022885	0.024608
-	RBF	0.2500	-	0.1	0.024845	0.026226
-	RBF	0.5000	-	0.1	0.026609	0.028110
-	RBF	1.0000	-	0.1	0.027562	0.030319
-	RBF	2.0000	-	0.1	0.028041	0.033539
-	RBF	4.0000	-	0.1	0.030082	0.039685
-	RBF	8.0000	-	0.1	0.037650	0.052373
-	RBF	16.0000	-	0.1	0.055006	0.075591
-	RBF	0.0625	-	1	0.063632	0.098843
-	RBF	0.1250	-	1	0.070252	0.108166
-	RBF	32.0000	-	0.1	0.081715	0.108504
-	RBF	0.2500	-	1	0.084526	0.124746
-	RBF	0.5000	-	1	0.104339	0.145914
-	RBF	1.0000	-	1	0.123539	0.165664
-	RBF	2.0000	-	1	0.137968	0.180304
-	RBF	4.0000	-	1	0.147609	0.190103
-	RBF	8.0000	-	1	0.154256	0.196912
-	RBF	0.0625	-	10	0.158334	0.200690
-	RBF	16.0000	-	1	0.159205	0.201991
-	RBF	0.1250	-	10	0.160477	0.203022
-	RBF	32.0000	-	1	0.162793	0.205660
-	RBF	0.2500	-	10	0.162921	0.205584
-	RBF	0.5000	-	10	0.164975	0.207698
-	RBF	1.0000	-	10	0.166360	0.209110
-	RBF	2.0000	-	10	0.167174	0.209937
-	RBF	4.0000	-	10	0.167614	0.210387
-	RBF	8.0000	-	10	0.167840	0.210621
-	RBF	16.0000	-	10	0.167949	0.210743
-	RBF	32.0000	-	10	0.167995	0.210809

Q29



Comment: RBF kernel fits better the regional variation (top-right and top-left), while polynomial kernel tends to generalize in the areas. Moreover, RBF kernel is a more "aggressive": fit while polynomial kernel seems to be more neutral (less variance). Also different with extreme X values, where RBF approaches 0 while polynomial fits to infinities.

Q30

$$f^* \in \arg\min_f R(f)$$

Since $\epsilon \sim \mathcal{N}(0, 0.1^2)$, the expected value of ϵ is 0. So,

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

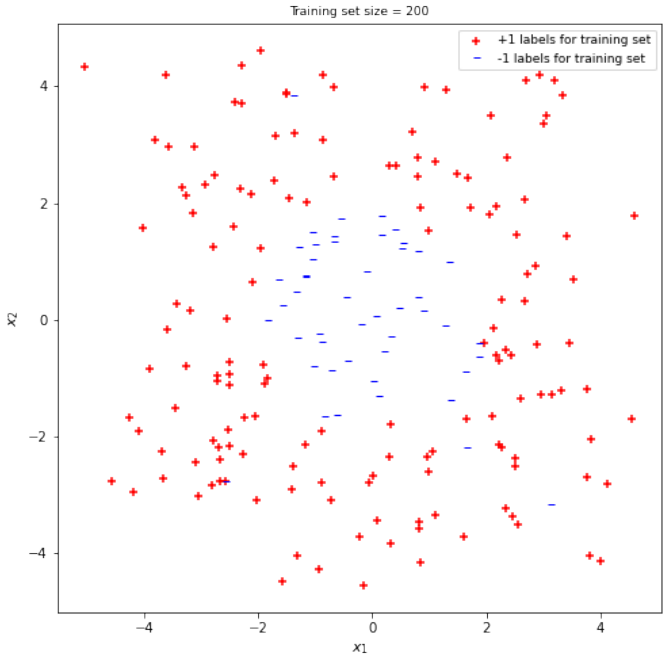
Consider,

$$\ell(\hat{y} - y)^2 = (f(x) + \epsilon - f(x))^2 \tag{37}$$
$$= \epsilon^2 \tag{38}$$

Since $\epsilon \sim \mathcal{N}(0, 0.1^2)$, $\mathbb{E}[\epsilon] = 0$, $Var[\epsilon] = 0.1^2$,

$$\mathbb{E}\left[\ell(\hat{y} - y)^2\right] = \mathbb{E}\left[\epsilon^2\right] \tag{39}$$
$$= \left(\mathbb{E}[\epsilon]\right)^2 + Var[\epsilon] \tag{40}$$
$$= Var[\epsilon] \tag{41}$$
$$= 0.1^2 \tag{42}$$

Q31*



Comment: The dataset is not linearly separable. It also seems that it's not quadratically separable. However, it can be gaussian-seperable using RBF kernel (unlikely, considering some outliers of negative labels).

Q32*

```
def train_soft_svm(X_train, y_train, kernel, lamb, epoch=20):
    assert lamb > 0

    X, Y = X_train, y_train
    K = kernel(X, X)
    t, dim = 0, K.shape[0]
    alpha = np.zeros(dim)

    for i in range(epoch):
        for j in range(dim):
            t += 1
            eta = 1 / (t * lamb)
            y_pred = np.matmul(K[i], alpha)

            alpha -= lamb * eta * np.matmul(K , alpha)
```

```
16
17              if Y[i][0] * y_pred < 1:
18                  alpha += Y[i][0] * eta * K[i]
19
20      return Kernel_Machine(kernel, X, alpha)
```

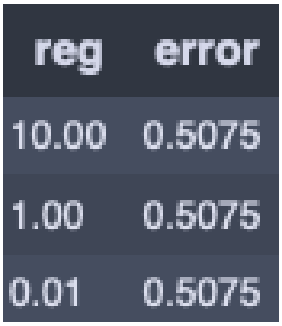
Q33*

Best performance for each kernel is reported in Table. 3.

Kernel	deg	l2_reg	offset	sigma	Error
Poly	2	0.01	0	-	0.0.220
RBF	-	0.0625	-	1	0.2550
Linear	-	1	-	-	0.5075

Table 3: Best performance for each kernel. Overall polynomial kernel achieves best test score with 0.220. No significant difference with changing regularization coefficient in polynomial kernel and in linear kernel.

From the three screenshots, one could observe deterioration of performance with change from best hyperparameters. No significant difference with changing regularization coefficient in polynomial kernel and in linear kernel.



Linear

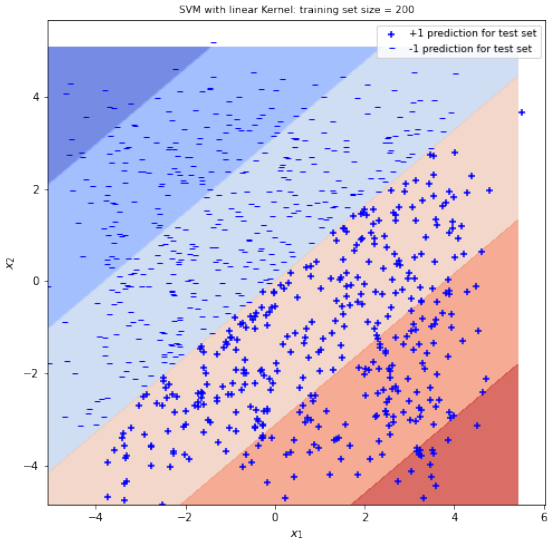
offset	deg	reg	error
0	2	10.00	0.22000
0	4	0.10	0.22000
0	4	10.00	0.22000
0	4	0.01	0.22000
0	2	0.10	0.22000
0	2	0.01	0.22000
-1	2	0.10	0.49000
1	3	0.01	0.49125
1	2	0.10	0.49125
-1	4	0.10	0.49250
-1	2	0.01	0.49375
1	3	0.10	0.49375
-1	4	10.00	0.49375
1	2	10.00	0.50125
-1	4	0.01	0.50125
1	4	10.00	0.50375
1	4	0.10	0.50625
1	3	10.00	0.50750
-1	2	10.00	0.50750
0	3	0.10	0.50750
0	3	10.00	0.50750
0	3	0.01	0.50750
1	2	0.01	0.50875
1	4	0.01	0.51000
-1	3	0.01	0.78000
-1	3	0.10	0.78000
-1	3	10.00	0.78000

Polynomial

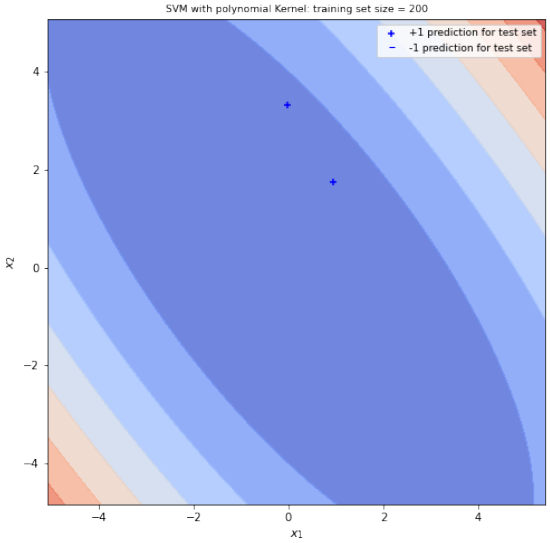
sigma	reg	error
1.0	0.0625	0.25500
1.0	0.1250	0.26250
1.0	0.2500	0.26875
1.0	0.5000	0.28000
1.0	2.0000	0.28375
1.0	1.0000	0.28375
1.0	4.0000	0.28375
1.0	8.0000	0.28375
1.0	16.0000	0.28375
1.0	32.0000	0.28375
0.1	32.0000	0.37750
0.1	2.0000	0.37750
0.1	4.0000	0.37750
0.1	8.0000	0.37750
0.1	16.0000	0.37750
0.1	1.0000	0.37875
0.1	0.5000	0.38000
0.1	0.1250	0.38125
0.1	0.2500	0.38125
0.1	0.0625	0.40625
10.0	0.2500	0.56000
10.0	0.5000	0.56000
10.0	1.0000	0.56000
10.0	2.0000	0.56000
10.0	32.0000	0.56000
10.0	8.0000	0.56000
10.0	16.0000	0.56000
10.0	0.1250	0.56000
10.0	4.0000	0.56000
10.0	0.0625	0.56000

RBF

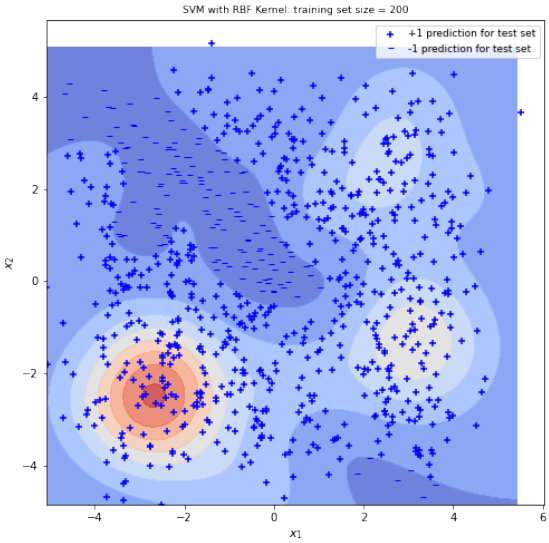
Q34*



Linear



Polynomial


SVM with RBF Kernel: training set size = 200
+1 prediction for test set
-1 prediction for test set
4
2
0
−2
−4
−4
−2
0
2
4
6
x_2
x_1

RBF