HW3-Coding

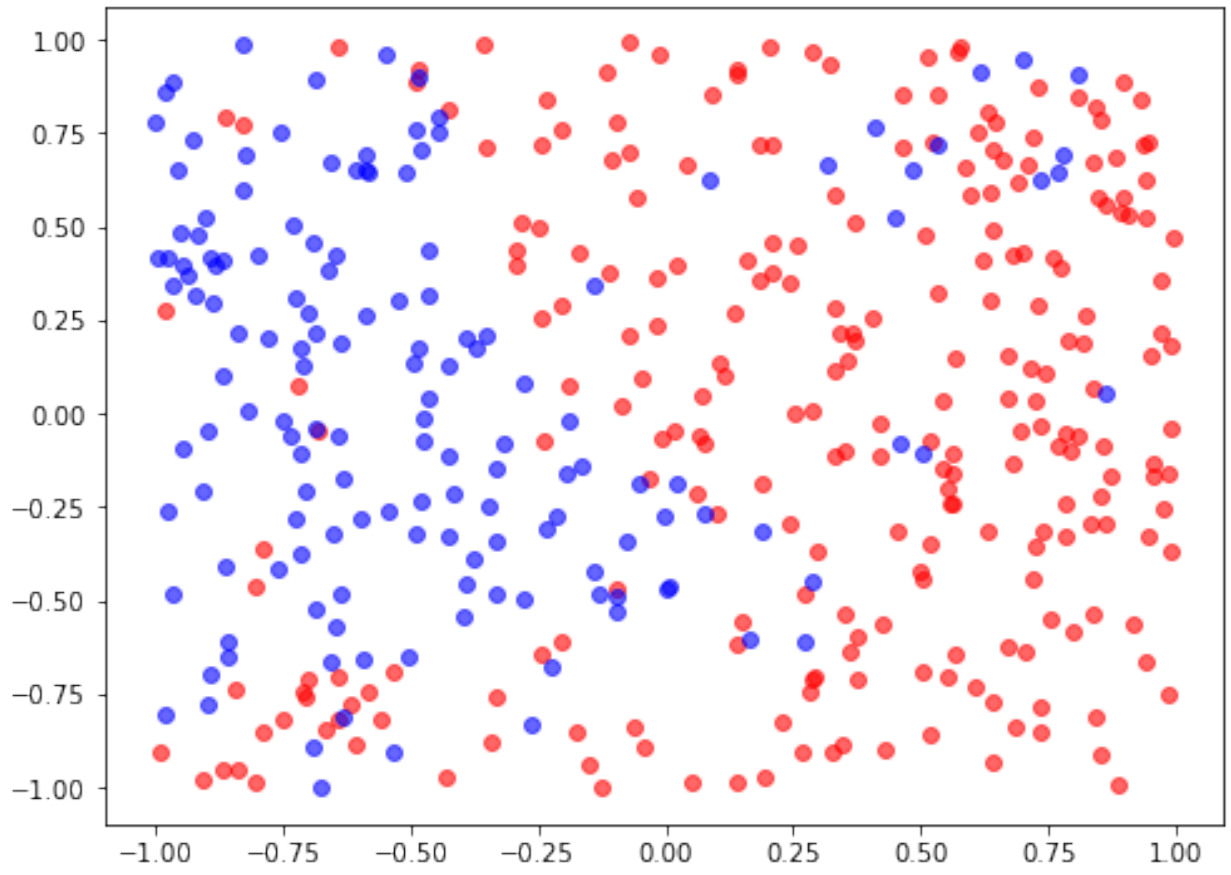
Q4 Kernel Power on SVM and Regularized Logistic Regression

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import plot_roc_curve, roc_auc_score, f1_score, accuracy_score

import warnings
warnings.filterwarnings('ignore')
```

```
data_Q4 = pd.read_csv('nonlineardata.csv', header=None)
data_Q4.columns=['x1','x2', 'label']
data_Q4.head()
```

	x1	x2	label
0	0.107878	0.132034	1.0
1	0.728726	0.287441	1.0
2	-0.130377	-0.483445	-1.0
3	0.009136	-0.465431	-1.0
4	0.790434	0.192522	1.0

```
# plot the data points

plt.figure(figsize=(8,6))
pos_idx = data_Q4['label'] == 1.0
plt.scatter(data_Q4['x1'].loc[pos_idx], data_Q4['x2'].loc[pos_idx],
            color='red', alpha=0.6)
neg_idx = data_Q4['label'] == -1.0
plt.scatter(data_Q4['x1'].loc[neg_idx], data_Q4['x2'].loc[neg_idx],
            color='blue', alpha=0.6);
```




```
# split data

X_q4 = data_Q4[['x1','x2']]
y_q4 = data_Q4['label']
X_train, X_test, y_train, y_test = train_test_split(X_q4, y_q4,
                                                    test_size=0.20, random_state=2022)
```

```
def plotClassifier(model, X, y):
    """plots the decision boundary of the model and the scatterpoints
       of the target values 'y'.

    Assumptions
    -----------
    y : it should contain two classes: '1' and '2'

    Parameters
    ----------
    model : the trained model which has the predict function

    X : the N by D feature array

    y : the N element vector corresponding to the target values

    """
```

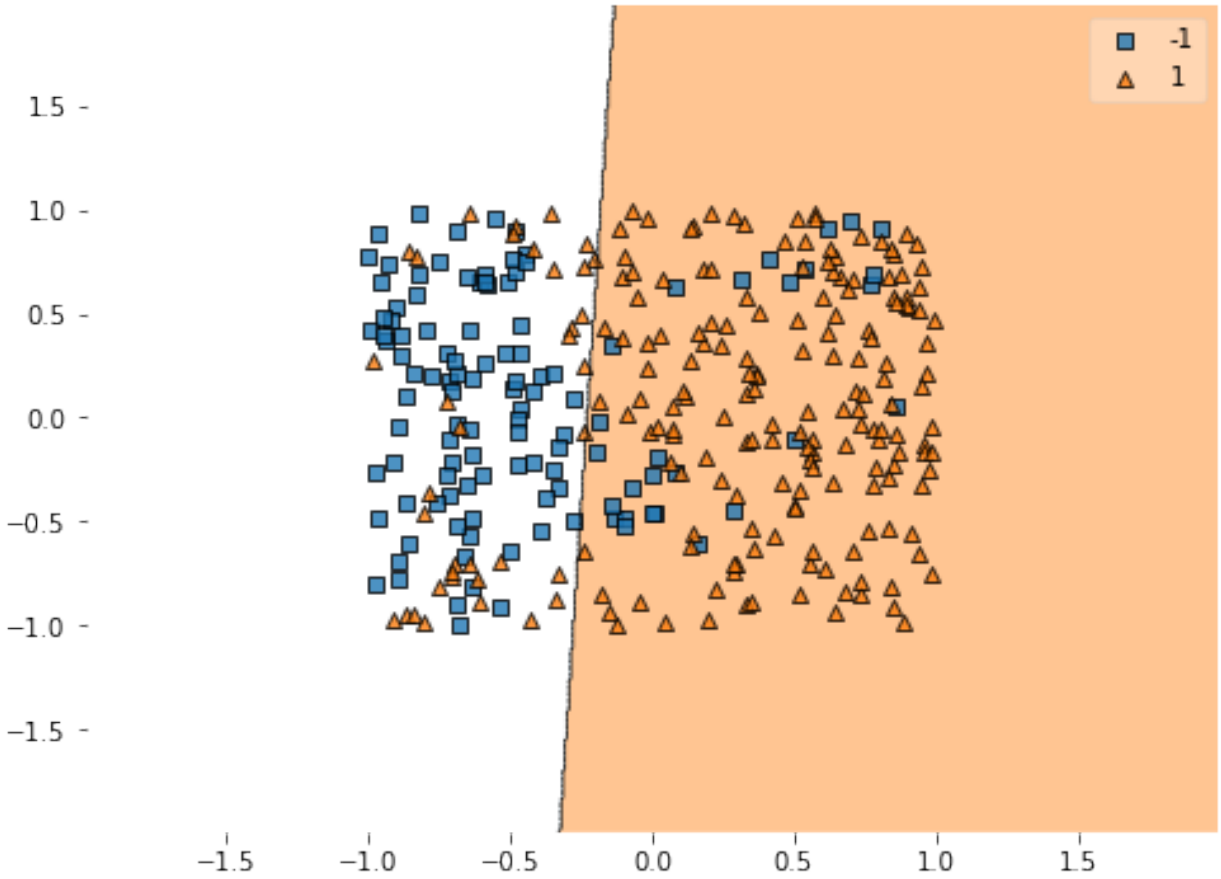
```
    x1 = X[:, 0]
    x2 = X[:, 1]

    x1_min, x1_max = int(x1.min()) - 1, int(x1.max()) + 1
    x2_min, x2_max = int(x2.min()) - 1, int(x2.max()) + 1

    x1_line =  np.linspace(x1_min, x1_max, 200)
    x2_line =  np.linspace(x2_min, x2_max, 200)

    x1_mesh, x2_mesh = np.meshgrid(x1_line, x2_line)

    mesh_data = np.c_[x1_mesh.ravel(), x2_mesh.ravel()]

    y_pred = np.array(model.predict(mesh_data))
    y_pred = np.reshape(y_pred, x1_mesh.shape)

    plt.figure()
    plt.xlim([x1_mesh.min(), x1_mesh.max()])
    plt.ylim([x2_mesh.min(), x2_mesh.max()])

    plt.contourf(x1_mesh, x2_mesh, -y_pred.astype(int), # unsigned int causes problems with negative si
                cmap=plt.cm.RdBu, alpha=0.6)


    y_vals = np.unique(y)
    plt.scatter(x1[y==y_vals[0]], x2[y==y_vals[0]], color="b", label="class %+d" % y_vals[0])
    plt.scatter(x1[y==y_vals[1]], x2[y==y_vals[1]], color="r", label="class %+d" % y_vals[1])
    plt.legend()
```

SVC with different kernels

4.1. Train a soft-margin SVM with linear kernel and C = 100

```
import matplotlib.gridspec as gridspec
from mlxtend.plotting import plot_decision_regions
```

```
C1=100
# fit the svc model
svc_clf_l = SVC(kernel='linear', C=C1)
svc_clf_l.fit(X_q4, y_q4)

fig = plt.subplots(figsize=(8,6))
plot_decision_regions(np.array(X_train), np.array(y_train.astype(int)), svc_clf_l);
plotClassifier(svc_clf_l, np.array(X_train), np.array(y_train));
```


-1
1
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

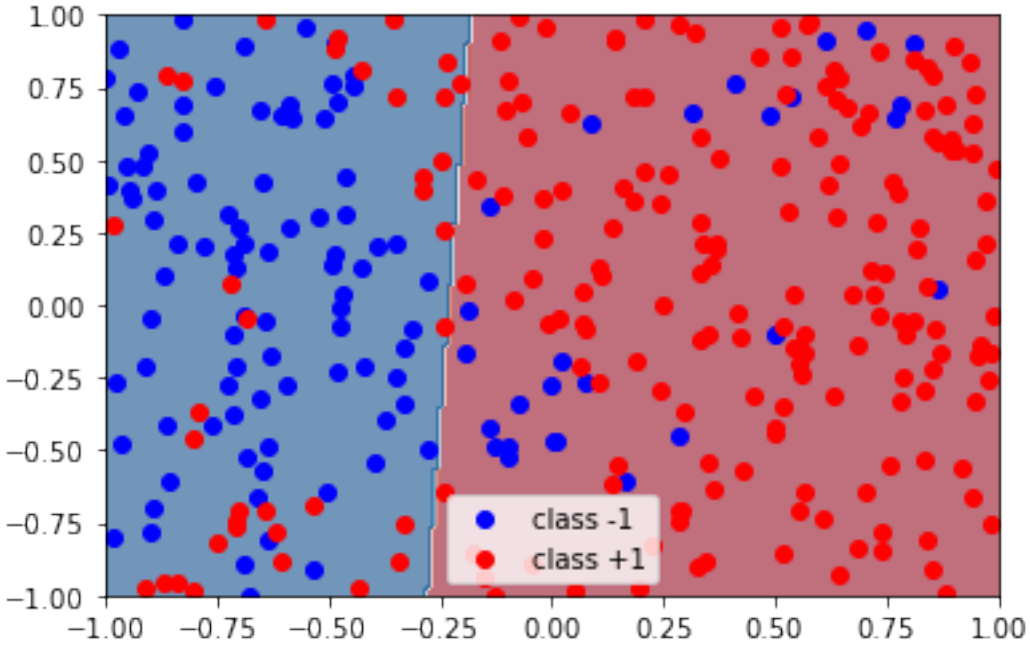

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
class -1
class +1

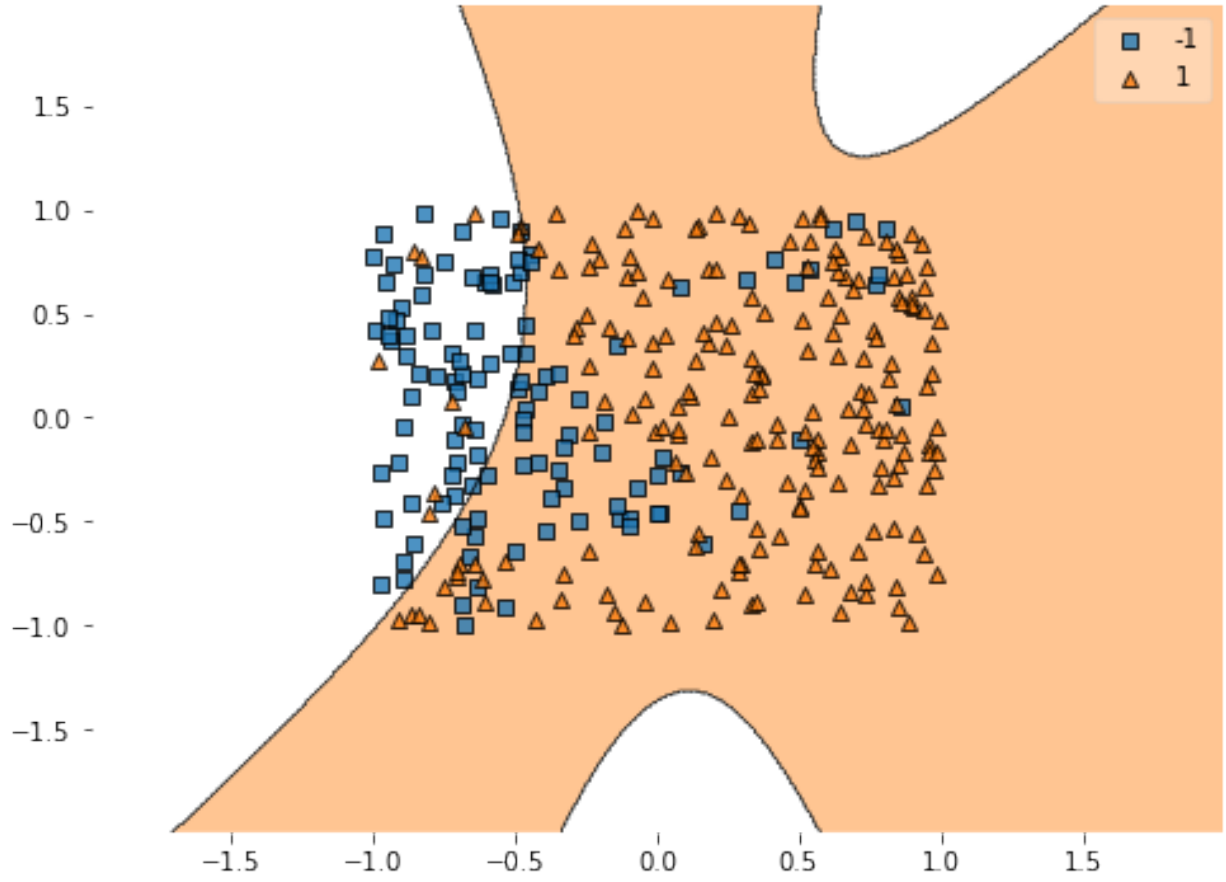
Figure 1: png

```
y_pred = svc_clf_l.predict(X_test)
y_train_pred = svc_clf_l.predict(X_train)
training_error_l = 1 - accuracy_score(y_train, y_train_pred)
test_error_l = 1 - accuracy_score(y_test, y_pred)
print('training error  when using linear kernel is:')
print(training_error_l)

print('testing error  when using linear kernel is:')
print(test_error_l)
```

```
training error  when using linear kernel is:
0.19687500000000002
testing error  when using linear kernel is:
0.23750000000000004
```

4.2. Train a soft-margin SVM with polynomial kernel

```
# fit the svc model
svc_clf_poly = SVC(kernel='poly', C=C1)
svc_clf_poly.fit(X_q4, y_q4)

fig = plt.subplots(figsize=(8,6))
plot_decision_regions(np.array(X_train), np.array(y_train.astype(int)), svc_clf_poly);
```



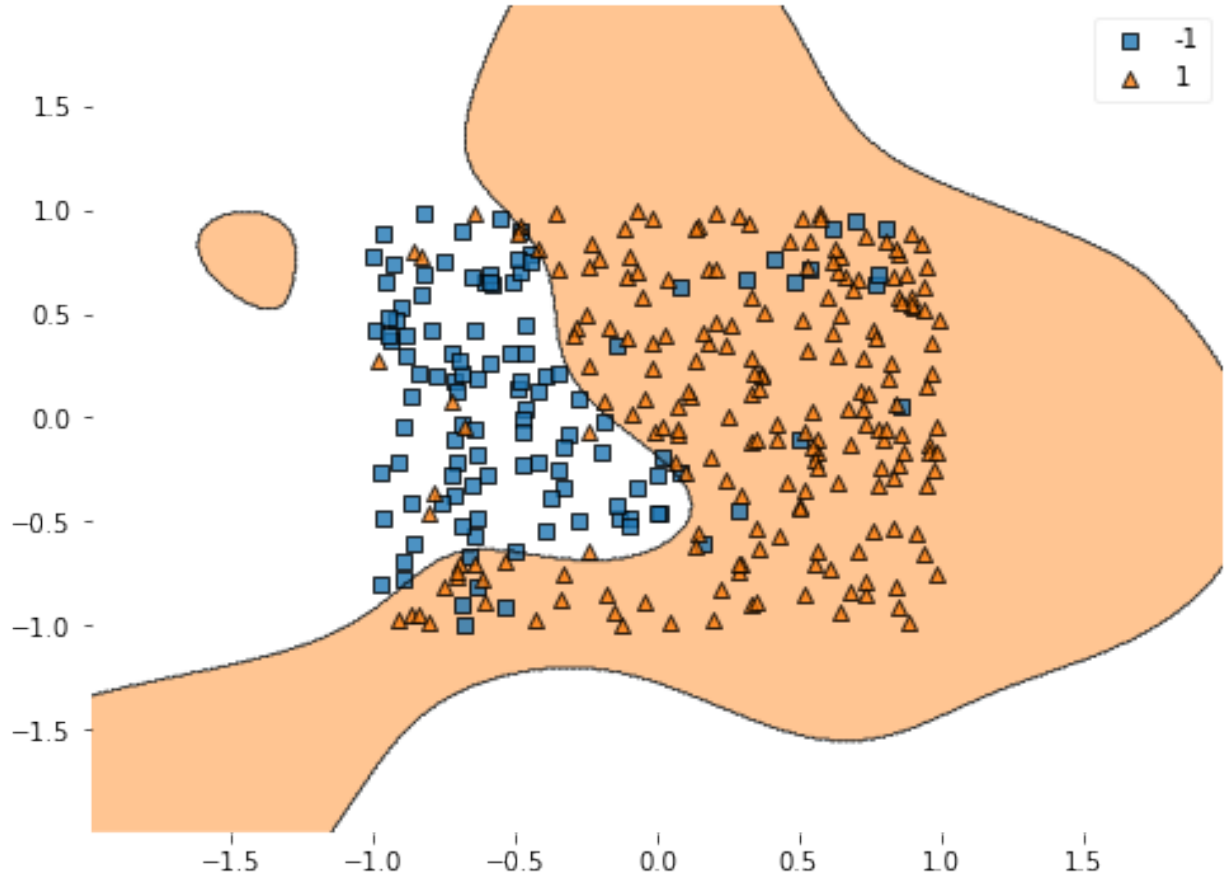

```
y_pred = svc_clf_poly.predict(X_test)
y_train_pred_p = svc_clf_poly.predict(X_train)
training_error_poly = 1 - accuracy_score(y_train, y_train_pred_p)
test_error_poly = 1 - accuracy_score(y_test, y_pred)
print('training error  when using polynomial kernel is:')
print(training_error_poly)

print('testing error  when using polynomial kernel is:')
print(test_error_poly)
```

```
training error  when using linear kernel is:
0.21562499999999996
testing error  when using linear kernel is:
0.26249999999999996
```

4.3. Train a soft-margin SVM with RBF kernel

```
# fit the svc model
svc_clf_rbf = SVC(kernel='rbf', C=C1)
svc_clf_rbf.fit(X_q4, y_q4)
```

```
fig = plt.subplots(figsize=(8,6))
plot_decision_regions(np.array(X_train), np.array(y_train.astype(int)), svc_clf_rbf);
```

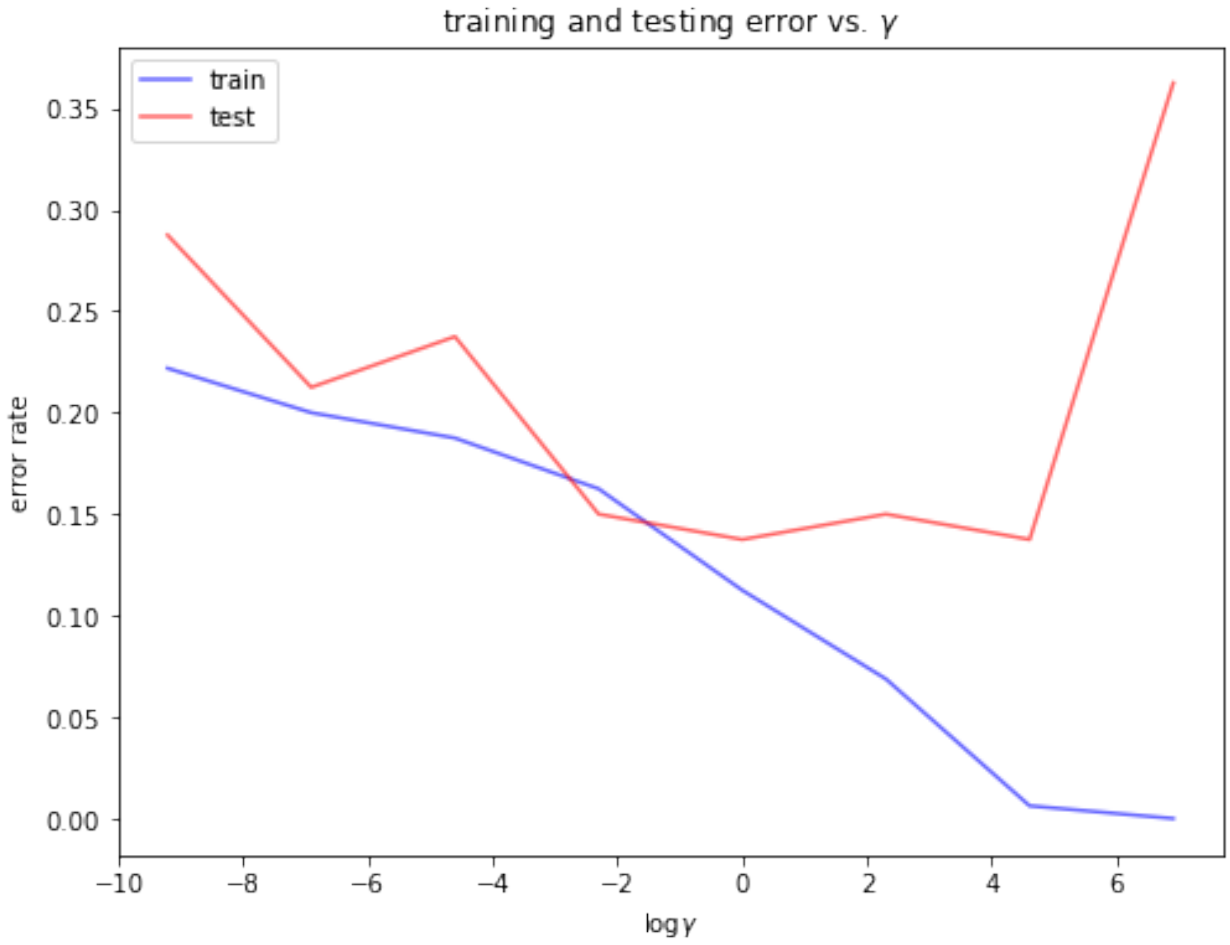



```
y_pred = svc_clf_rbf.predict(X_test)
y_train_pred_rbf = svc_clf_rbf.predict(X_train)
training_error_rbf = 1 - accuracy_score(y_train, y_train_pred_rbf)
test_error_rbf = 1 - accuracy_score(y_test, y_pred)
print('training error  when using rbf kernel is:')
print(training_error_rbf)

print('testing error  when using rbf kernel is:')
print(test_error_rbf)
```

```
training error  when using rbf kernel is:
0.10312500000000002
testing error  when using rbf kernel is:
0.15000000000000002
```

4.4. RBF with different gammas

```
gammas = [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000]

log_gamma = [np.log(g) for g in gammas]

# def PlotRBFError():
train_error_rbf = []
test_error_rbf = []
# train SVC with different gammas
for gamma in gammas:

    svc_rbf = SVC(kernel='rbf', C=C1, gamma = gamma)
    svc_rbf.fit(X_train, y_train)
    # report training error
    y_train_pred = svc_rbf.predict(X_train)
    train_error = 1 - accuracy_score(y_train, y_train_pred)
    train_error_rbf.append(train_error)

    # report testing error
    y_pred = svc_rbf.predict(X_test)
    test_error = 1 - accuracy_score(y_test, y_pred)
    test_error_rbf.append(test_error)



# plot train and test error w.r.t. gamma
fig = plt.subplots(figsize=(8,6))
plt.plot(log_gamma, train_error_rbf, color='blue', alpha=0.6)
plt.plot(log_gamma, test_error_rbf, color='red', alpha=0.6)
plt.title(u'training and testing error vs. $\gamma$');
plt.legend(['train','test'])
plt.xlabel(u'log$\gamma$')
plt.ylabel('error rate');
```



What do you notice about the error rates as γ increases? Why this is the case? Finally, in the bias variance tradeoff, does small γ correspond to high bias or high variance?
Answer:

I used F1 score to show error rate. As γ gets higher, f1 score increases, which means error rate gets lower.

As γ increases, the decision boundary will be closer to data points, and thus could classify more accurately. However, if γ gets too high, it may overfit.

Small γ correspond to high bias. Because it tends to underfit.

Logistic Regression with Different Kernels

```
import pickle
import os
from sklearn.model_selection import train_test_split
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.optimize import approx_fprime
from numpy.linalg import norm
import math
```

```
def check_gradient(model, X, y, dimensionality, verbose=True):
    # This checks that the gradient implementation is correct
    w = np.random.rand(dimensionality)
    f, g = model.funObj(w, X, y)

    # Check the gradient
    estimated_gradient = approx_fprime(w,
                                       lambda w: model.funObj(w,X,y)[0],
                                       epsilon=1e-6)

    implemented_gradient = model.funObj(w, X, y)[1]

    if np.max(np.abs(estimated_gradient - implemented_gradient) > 1e-3):
        raise Exception('User and numerical derivatives differ:\n%s\n%s' %
             (estimated_gradient[:5], implemented_gradient[:5]))
    else:
        if verbose:
            print('User and numerical derivatives agree.')
```

```
def log_1_plus_exp_safe(x):
    out = np.log(1+np.exp(x))
    out[x > 100] = x[x>100]
    out[x < -100] = np.exp(x[x < -100])
    return out
```

```
def findMin(funObj, w, maxEvals, *args, verbose=0):
    """
    Uses gradient descent to optimize the objective function

    This uses quadratic interpolation in its line search to
    determine the step size alpha
    """
    # Parameters of the Optimization
    optTol = 1e-2
    gamma = 1e-4

    # Evaluate the initial function value and gradient
    f, g = funObj(w,*args)
    funEvals = 1

    alpha = 1.
    while True:
        # Line-search using quadratic interpolation to
        # find an acceptable value of alpha
        gg = g.T.dot(g)

        while True:
            w_new = w - alpha * g
            f_new, g_new = funObj(w_new, *args)

            funEvals += 1
            if f_new <= f - gamma * alpha*gg:
                break
```

```
            if verbose > 1:
                print("f_new: %.3f - f: %.3f - Backtracking..." % (f_new, f))

            # Update step size alpha
            alpha = (alpha**2) * gg/(2.*(f_new - f + alpha*gg))

        # Print progress
        if verbose > 0:
            print("%d - loss: %.3f" % (funEvals, f_new))

        # Update step-size for next iteration
        y = g_new - g
        alpha = -alpha*np.dot(y.T,g) / np.dot(y.T,y)

        # Safety guards
        if np.isnan(alpha) or alpha < 1e-10 or alpha > 1e10:
            alpha = 1.

        if verbose > 1:
            print("alpha: %.3f" % (alpha))

        # Update parameters/function/gradient
        w = w_new
        f = f_new
        g = g_new

        # Test termination conditions
        optCond = norm(g, float('inf'))

        if optCond < optTol:
            if verbose:
                print("Problem solved up to optimality tolerance %.3f" % optTol)
            break

        if funEvals >= maxEvals:
            if verbose:
                print("Reached maximum number of function evaluations %d" % maxEvals)
            break

    return w, f
```

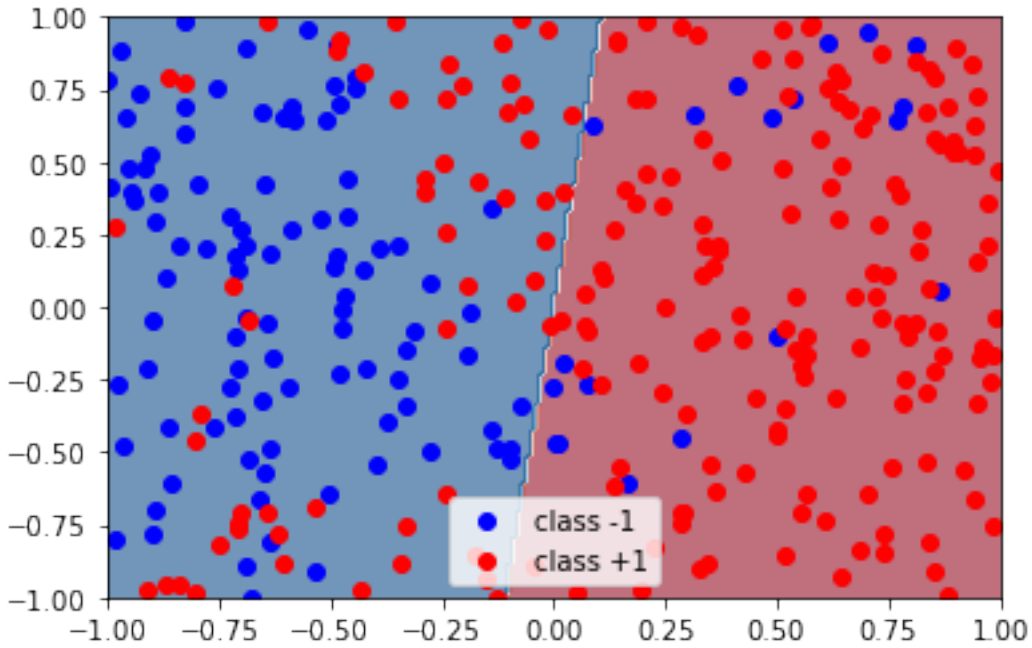
```
class kernelLogRegL2():
    def __init__(self, lammy=1.0, verbose=0, maxEvals=100,
                 kernel_fun=kernel_linear, **kernel_args):
        self.verbose = verbose
        self.lammy = lammy
        self.maxEvals = maxEvals
        self.kernel_fun = kernel_fun
        self.kernel_args = kernel_args

    def funObj(self, u, K, y):
        yKu = y * (K@u)
```

```
        # Calculate the function value
        # f = np.sum(np.log(1. + np.exp(-yKu)))
        f = np.sum(log_1_plus_exp_safe(-yKu))

        # Add L2 regularization
        f += 0.5 * self.lammy * u.T@K@u

        # Calculate the gradient value
        res = - y / (1. + np.exp(yKu))
        g = (K.T@res) + self.lammy * K@u

        return f, g


    def fit(self, X, y):
        n, d = X.shape
        self.X = X

        K = self.kernel_fun(X,X, **self.kernel_args)

        check_gradient(self, K, y, n, verbose=self.verbose)
        self.u, f = findMin(self.funObj, np.zeros(n), self.maxEvals,
                            K, y, verbose=self.verbose)

    def predict(self, Xtest):
        Ktest = self.kernel_fun(Xtest, self.X, **self.kernel_args)
        return np.sign(Ktest@self.u)
```

```
def kernel_linear(X1, X2):
    return X1 @ X2.T

def kernel_poly(X1, X2, p=2):
    #Your code here
    return (1 + X1 @ X2.T)**2


def kernel_RBF(X1, X2, sigma=0.5):

    mat = np.array([[1]* len(X2) for x in range(len(X1))])

    for i in range(len(X1)):
        for j in range(len(X2)):
            mat[i][j] = math.dist(X1[i,:], X2[j,:])

    result = np.exp(-1 * mat / sigma**2)

    return result
```

4.5. Train a L2 regularized Logistic Regression classifier with linear kernel, $\lambda = 0.01$, and report training and testing error.

```
kernelLog_linear = kernelLogRegL2(lammy=0.01)
kernelLog_linear.fit(X_train, y_train)
plotClassifier(kernelLog_linear, np.array(X_train), np.array(y_train))
```

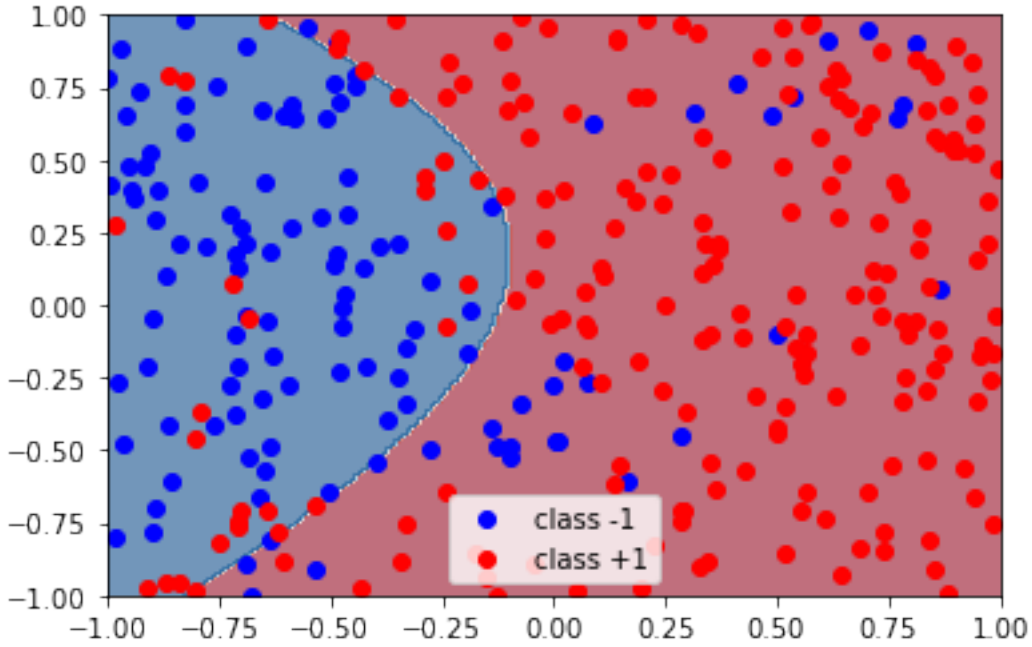



```
y_train_linear = kernelLog_linear.predict(X_train)
y_pred_linear = kernelLog_linear.predict(X_test)
log_l_train_error = 1 - accuracy_score(y_train, y_train_linear)
log_l_test_error = 1 - accuracy_score(y_test, y_pred_linear)
print('logistic regression training error when using linear kernel is:')
print(log_l_train_error)

print('logistic regression testing error when using linear kernel is:')
print(log_l_test_error)
```

```
logistic regression training error when using linear kernel is:
0.23750000000000004
logistic regression testing error when using linear kernel is:
0.23750000000000004
```

4.6. Train a L2 regularized Logistic Regression classifier with polynomial kernel, $\lambda = 0.01$, and report training and testing error.

```
kernelLog_poly = kernelLogRegL2(lammy=0.01, kernel_fun = kernel_poly)
kernelLog_poly.fit(X_train, y_train)
plotClassifier(kernelLog_poly, np.array(X_train), np.array(y_train))
```

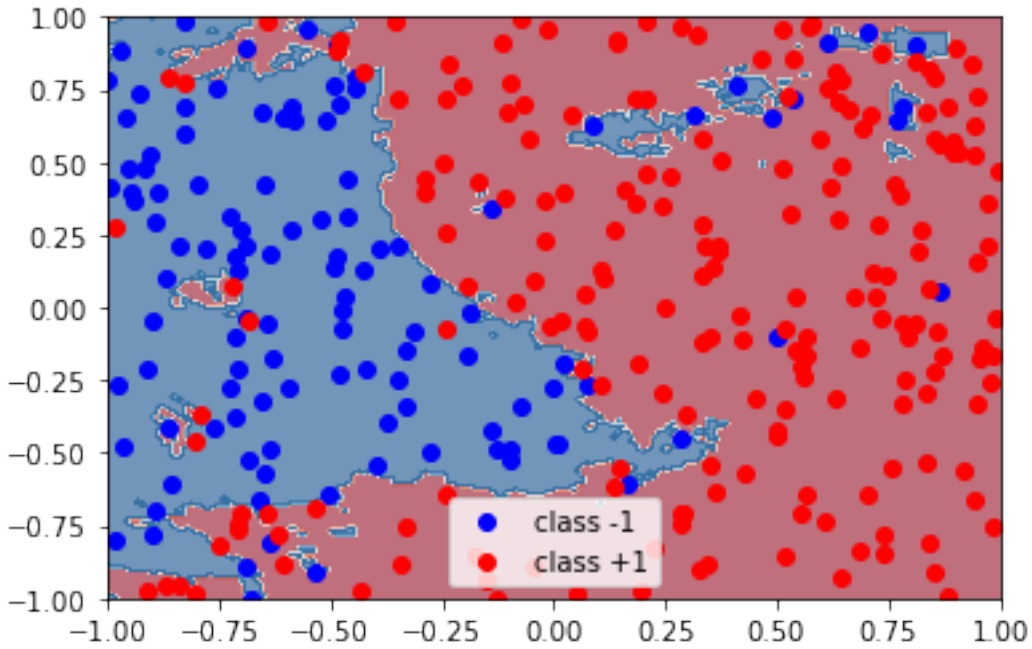



```
y_train_poly = kernelLog_poly.predict(X_train)
y_pred_poly = kernelLog_poly.predict(X_test)
log_p_train_error = 1 - accuracy_score(y_train, y_train_poly)
log_p_test_error = 1 - accuracy_score(y_test, y_pred_poly)
print('logistic regression training error when using polynomial kernel is:')
print(log_p_train_error)

print('logistic regression testing error when using polynomial kernel is:')
print(log_p_test_error)
```

```
logistic regression training error when using polynomial kernel is:
0.18437499999999996
logistic regression testing error when using polynomial kernel is:
0.23750000000000004
```

4.7. Train a L2 regularized Logistic Regression classifier with RBF kernel, $\lambda = 0.01$, $\sigma = 0.5$, and report training and testing error.

```
kernelLog_rbf = kernelLogRegL2(lammy=0.01, kernel_fun = kernel_RBF)
kernelLog_rbf.fit(X_train.values, y_train.values)
plotClassifier(kernelLog_rbf, np.array(X_train), np.array(y_train))
```




```
y_train_rbf = kernelLog_rbf.predict(X_train.values)
y_pred_rbf = kernelLog_rbf.predict(X_test.values)
log_rbf_train_error = 1 - accuracy_score(y_train, y_train_rbf)
log_rbf_test_error = 1 - accuracy_score(y_test, y_pred_rbf)
print('logistic regression training error when using RBF kernel is:')
print(log_rbf_train_error)

print('logistic regression testing error when using RBF kernel is:')
print(log_rbf_test_error)
```

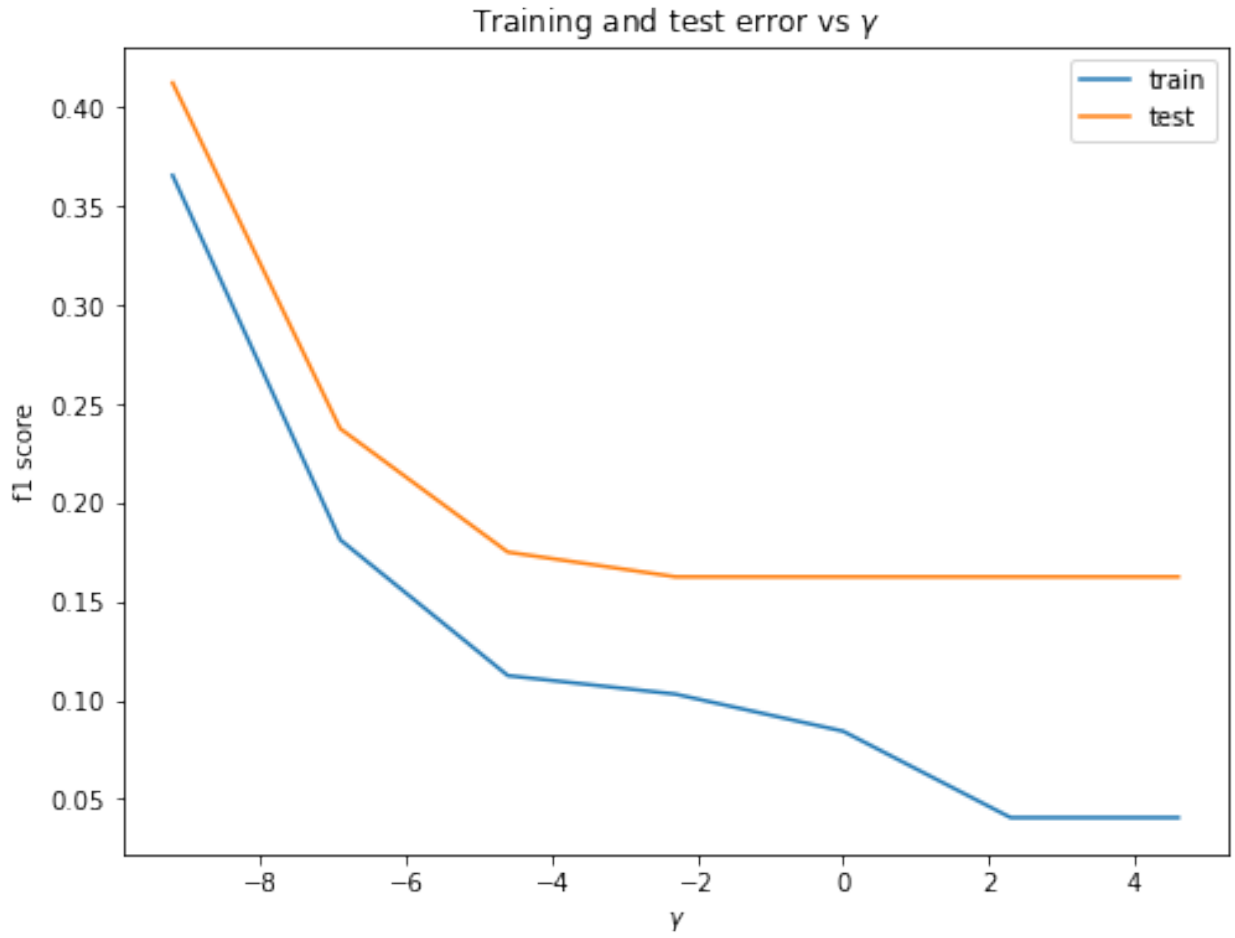
```
logistic regression training error when using RBF kernel is:
0.043749999999999956
logistic regression testing error when using RBF kernel is:
0.16249999999999998
```

4.8. Try different λ and plot the training and test error rates vs γ

```
train_error48 = []
test_error48 = []
gammas = [0.0001, 0.001, 0.01, 0.1, 1, 10, 100]
for gamma in gammas:

    sigma = np.sqrt(1 / gamma)
    kernelLog_rbf48 = kernelLogRegL2(lammy=0.01, kernel_fun = kernel_RBF, sigma=sigma)
    kernelLog_rbf48.fit(X_train.values, y_train.values)
    y_train_rbf48 = kernelLog_rbf48.predict(X_train.values)
    y_pred_rbf48 = kernelLog_rbf48.predict(X_test.values)
    log_rbf_train_error48 = 1 - accuracy_score(y_train, y_train_rbf48)
    log_rbf_test_error48 = 1 - accuracy_score(y_test, y_pred_rbf48)
```

```
    train_error48.append(log_rbf_train_error48)
    test_error48.append(log_rbf_test_error48)
```

```
log_gamma = [np.log(g) for g in gammas]
fig48, ax48 = plt.subplots(figsize=(8,6))
plt.plot(log_gamma, train_error48)
plt.plot(log_gamma, test_error48);
plt.xlabel(u'$\gamma$')
plt.ylabel('f1 score')
plt.legend(['train', 'test'])
plt.title(u'Training and test error vs $\gamma$');
```



Q5 Sparse Logistic Regression

5.1 plot ROC curve

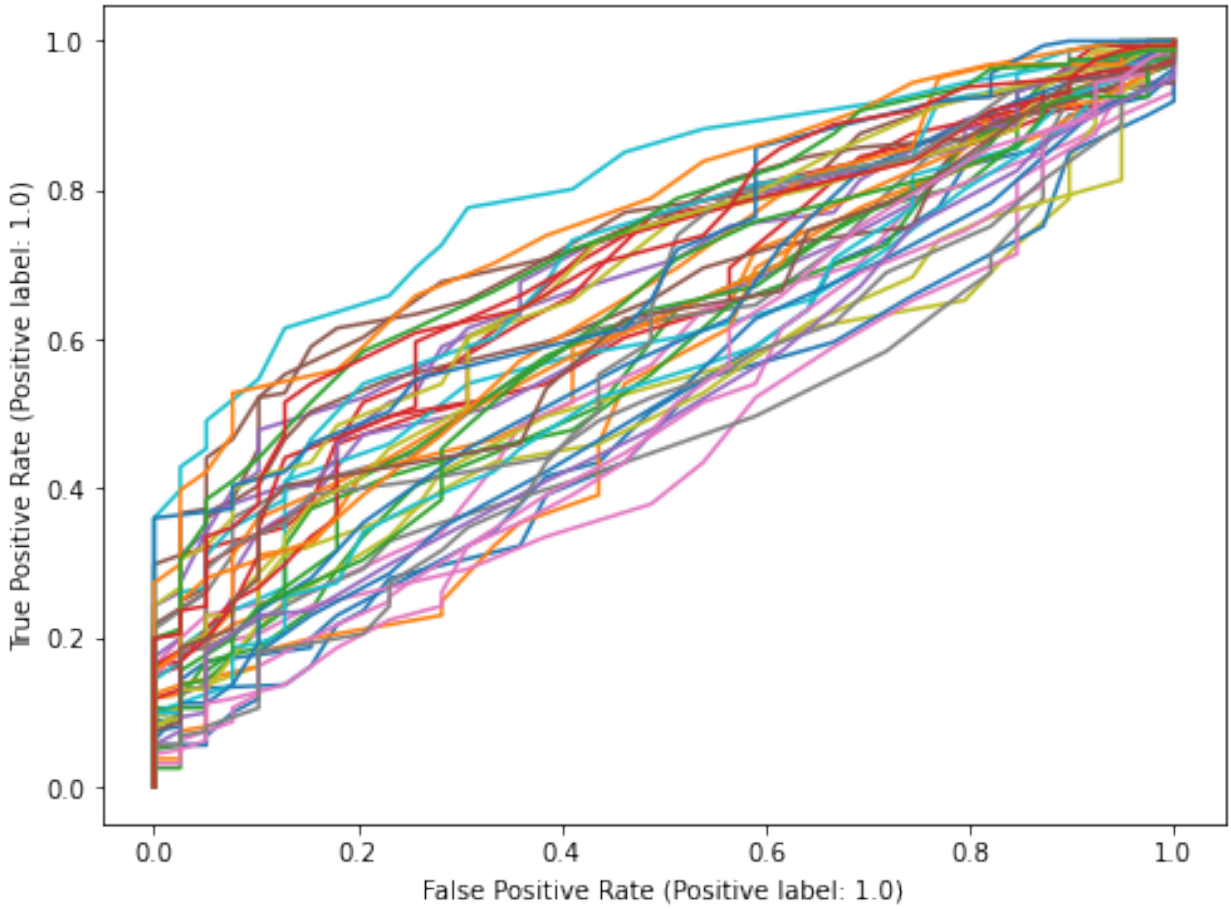
```
from sklearn.linear_model import LogisticRegression
from sklearn import metrics, model_selection
```

```
df_q5 = pd.read_csv('spectf.csv',header=None)
df_q5.shape
# df_q5.head()
```

```
(267, 45)
```

```
X_q5 = df_q5.iloc[:,1:]
y_q5 = df_q5.iloc[:, 0]
X_train_q5, X_test_q5, y_train_q5, y_test_q5 = train_test_split(X_q5, y_q5,
                                                    test_size=0.25, random_state=2022)
```

```
features = X_train_q5.columns
fig5, ax5 = plt.subplots(figsize=(8,6))
# y_train5 = np.array(y_train_q5).reshape(-1,1)
# y_test5 = np.array(y_test_q5).reshape(-1,1)
# fig5, ax5 = plt.subplots(figsize=(8,6))
aucs = {}
for f in features:
    X = np.array(X_train_q5[f]).reshape(-1,1)
    log_clf = LogisticRegression(penalty='none',random_state=0).fit(X=X, y=y_train_q5)
#     X_test5 = np.array(X_test_q5[f]).reshape(-1,1)
    y_pred5 = log_clf.predict(X)

    metrics.plot_roc_curve(log_clf, X, y_train_q5, ax=ax5)
    auc = metrics.roc_auc_score(y_train_q5, y_pred5)
    aucs[f] = auc
    ax5.get_legend().remove()
# plt.show()
```



5.2 Create an algorithm for choosing 300 different subsets of these features

```
# Covariance matrix
X_array = np.array(X_train_q5)
cov = np.cov(X_array.T)
mean_var = np.mean(cov)

# extract features that have AUC score above 0.5
auc_above = {}      # featurs with AUC above 0.5
auc_low = {}       # all other features
for key, value in aucs.items():
    if value > 0.5:
        auc_above[key] = value

    else:
        auc_low[key] = value


# number of features
n_subsets = 0
subsets = []
```

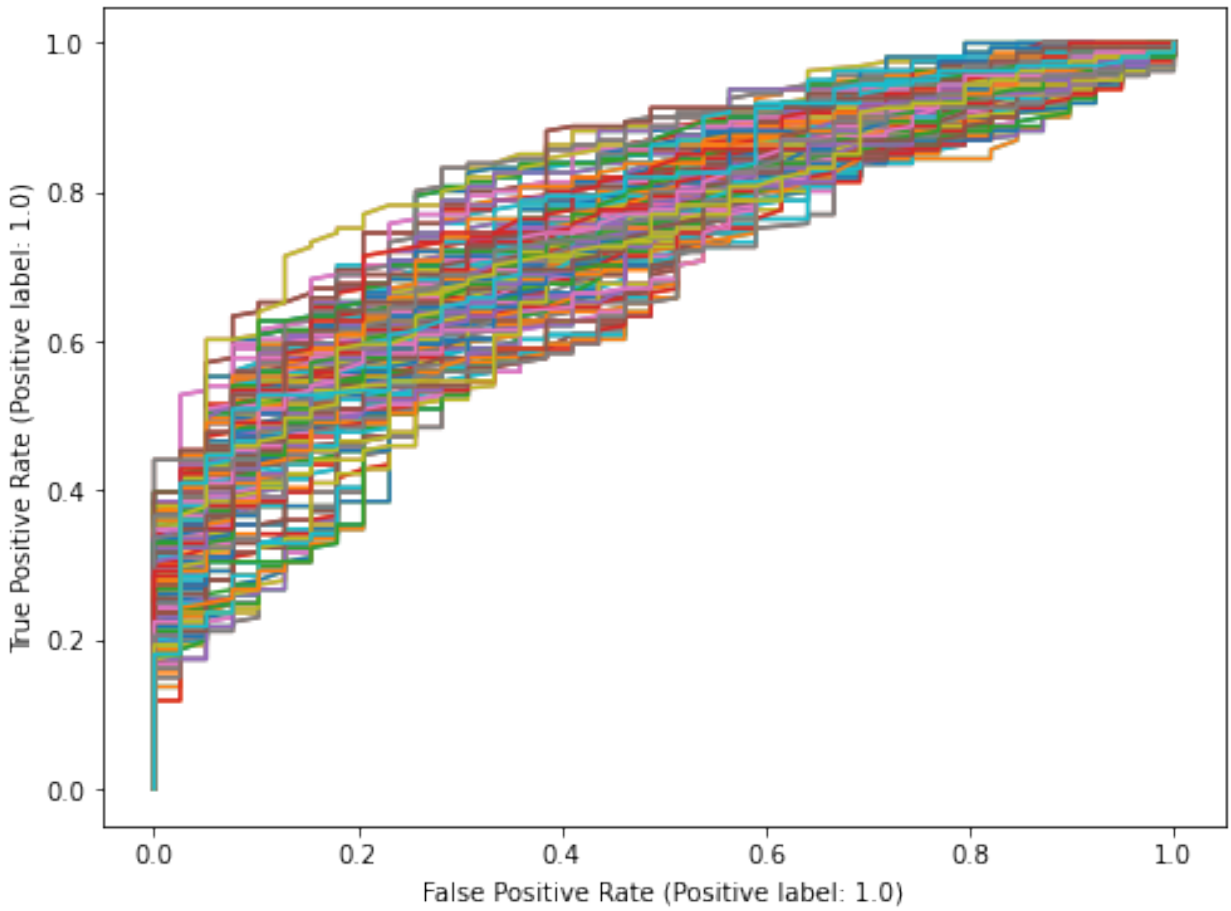
```
# for each such feature, iterate all other features
while n_subsets < 300:
    # for each feature with a higher AUC, keep it
    # and search for other features
    for k, v in auc_above.items():
        for k1, v1 in aucs.items():
            if k != k1:
                cov_k = cov[k-1,k1-1]    # check the covriance
                if cov_k < mean_var:
                    # only take two features that are not highly correlated
                    subsets.append([k,k1])
                    n_subsets += 1
#     print(n_subsets)
# while n_subsets < 300:
    for k2, v2 in auc_low.items():
        for k3, v3 in aucs.items():
            if (k2 != k3) and ([k2,k3] or [k3,k2] not in subsets):
                # exclude identical subsets
                cov_k_ = cov[k2-1,k3-1]
                if cov_k < mean_var:
                    subsets.append([k2,k3])
                    n_subsets += 1


subsets1 = subsets[:300]
```

My algorithm:

I search for subsets with two features. First divide the features into two groups, 'good 'features with higher AUC (above 0.5) and 'bad' ones with AUC of 0.5. Then for each subset, I want to keep one of these 'good' features, and add another feature that is not highly correlated with it by checking the covariance. Repeat the steps until reaching 300 subsets.

5.3 Train 300 logistic regression models by calling a logistic regression solver on each subset of data

```
auc53 = {}
fig53, ax53 = plt.subplots(figsize=(8,6))
for idx, s in enumerate(subsets1):
    X_3 = X_train_q5[[s[0],s[1]]]
    log_clf53 = LogisticRegression(penalty='none',random_state=0).fit(X=X_3, y=y_train_q5)
#     X_test5 = np.array(X_test_q5[f]).reshape(-1,1)
    y_pred53 = log_clf53.predict(X_3)
    # plot ROC
    metrics.plot_roc_curve(log_clf53, X_3, y_train_q5, ax=ax53)
    auc1 = metrics.roc_auc_score(y_train_q5, y_pred53)
    auc53[idx] = auc1
ax53.get_legend().remove()
```



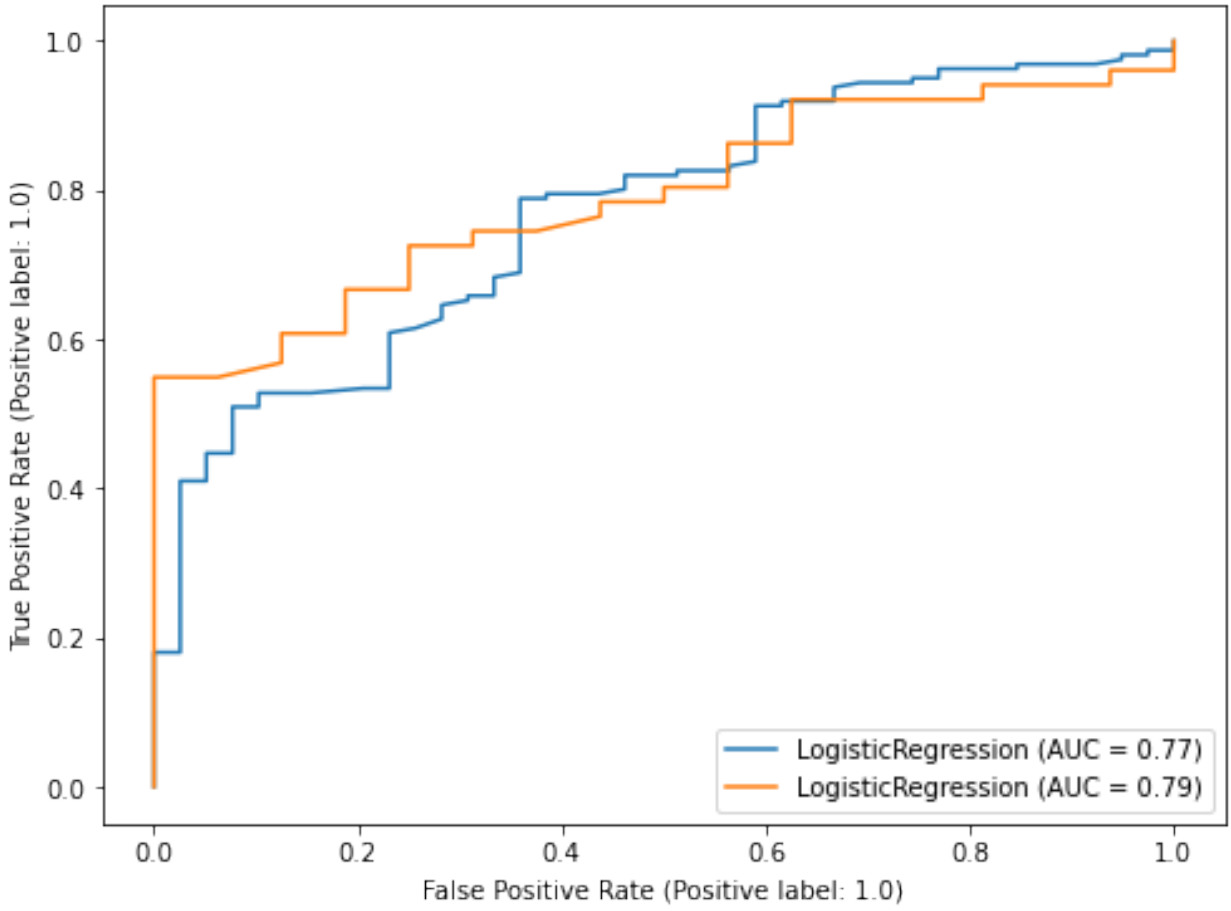

```
max_auc = max(auc53.values())
# min(auc53.values())

def ReturnKey(dictionary, value):

    for k,v in dictionary.items():
        if v == value:

            key = k
    return k

idxmax = ReturnKey(auc53, max_auc)
```

```
# subset with the highest the AUC among all 300 subsets I chose
subset_max = subsets[idxmax]
subset_max
```

```
[43, 36]
```

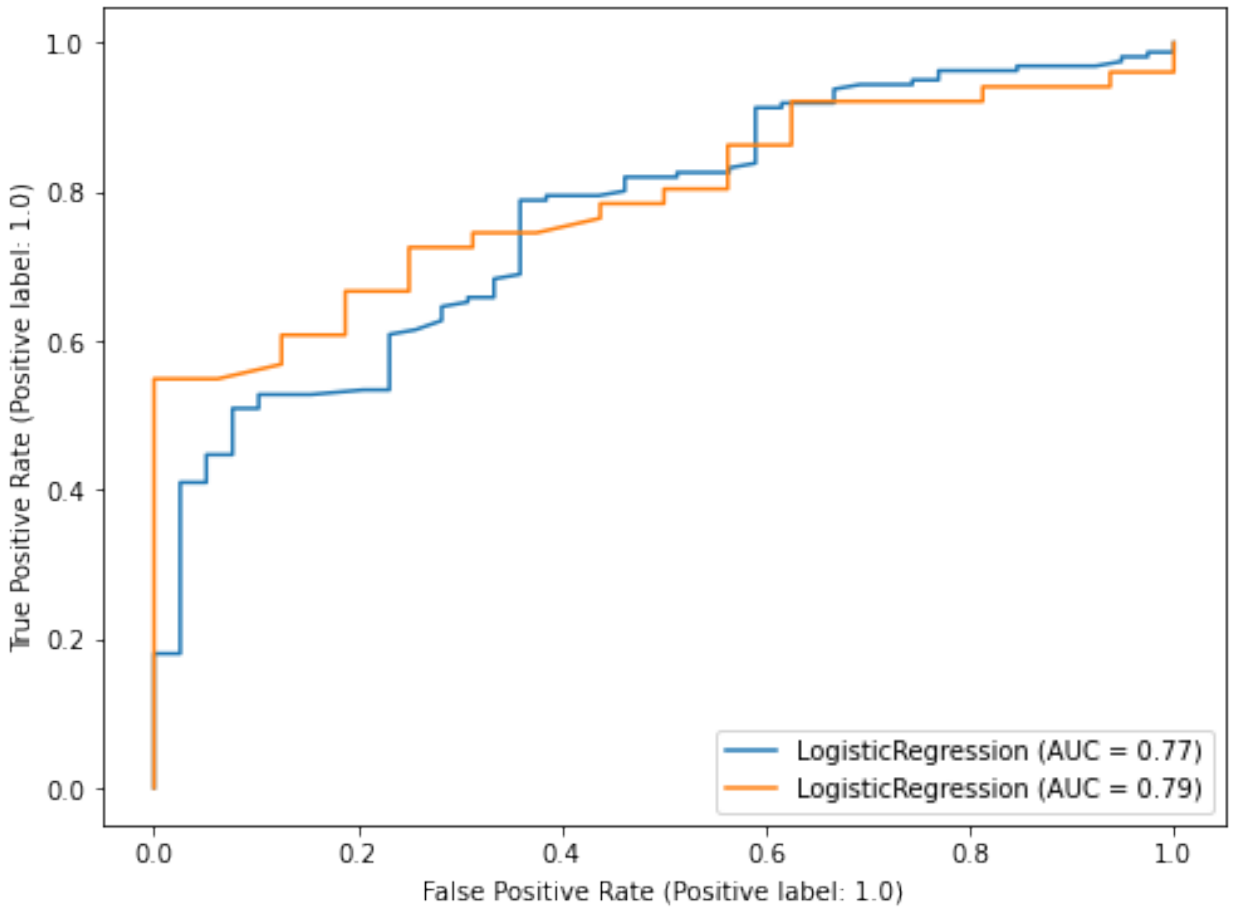
5.4 Put the training ROC for f and the test ROC for f on the same figure and report the test AUC.

```
X_train_best = X_train_q5[[subset_max[0],subset_max[1]]]
X_test_best = X_test_q5[[subset_max[0],subset_max[1]]]
log_clf_best = LogisticRegression(penalty='none',random_state=0).fit(
                                        X=X_train_best, y=y_train_q5)

y_train_pred = log_clf_best.predict(X_train_best)
y_test_pred = log_clf_best.predict(X_test_best)

fig54, ax54 = plt.subplots(figsize=(8,6))
metrics.plot_roc_curve(log_clf_best, X_train_best, y_train_q5, ax=ax54)
metrics.plot_roc_curve(log_clf_best, X_test_best, y_test_q5, ax=ax54);
```




```
# l2 regularized logistic regression

log_clf_l2 = LogisticRegression(penalty='l2',random_state=0).fit(
                                        X=X_train_best, y=y_train_q5)
```

```
y_train_pred = log_clf_l2.predict(X_train_best)
y_test_pred = log_clf_l2.predict(X_test_best)

fig541, ax541 = plt.subplots(figsize=(8,6))
metrics.plot_roc_curve(log_clf_l2, X_train_best, y_train_q5, ax=ax541)
metrics.plot_roc_curve(log_clf_l2, X_test_best, y_test_q5, ax=ax541);
```



It seems this technique works roughly the same as l2-regularized logistic regression.