

Problem 4

Starter Code from Kernel_starter.ipynb

```
In [ ]: 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
```

```
In [ ]: 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.')
```

```
In [ ]: 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 = 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
             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 %+"
plt.scatter(x1[y==y_vals[1]], x2[y==y_vals[1]], color="r", label="class %+"
plt.legend()

```

```

In [ ]: 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" % optTo
    break

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

return w, f

```

```

In [ ]: 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

```

```

In [ ]: def kernel_linear(X1, X2):
        return X1@X2.T

```

```

In [ ]: class kernelLogRegL2():
        def __init__(self, lammy=1.0, verbose=0, maxEvals=100, kernel_fun=kernel_l
            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, ver

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

```

Start of my own code

```

In [ ]: import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
import matplotlib.pyplot as plt

#import the nonlineardata.csv file
df = pd.read_csv("nonlineardata.csv",header=None)

#access the x and y data sets
data = df.to_numpy()
X = data[:,0:2]
Y = data[:,2]

#split the data into 80% training and 20% testing with random state to 2022
X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size=0.2,random_

```

4.1 SVM with linear Kernel, C = 100

```

In [ ]: clf = SVC(C=100,kernel="linear")
clf.fit(X_train,Y_train)

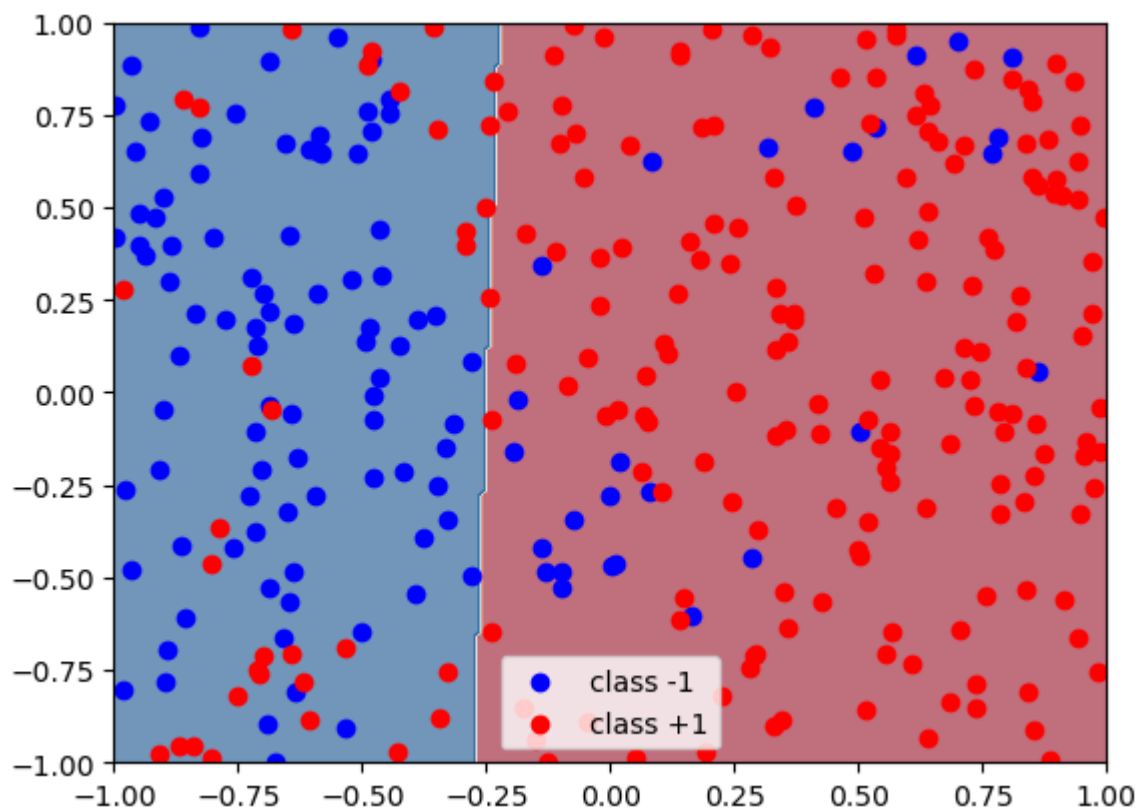
train_accuracy = clf.score(X_train,Y_train)
test_accuracy = clf.score(X_test,Y_test)

out_str = "Test Accuracy: {}\nTrain Accuracy: {}".format(test_accuracy,train_a
print(out_str)

plotClassifier(clf,X_train,Y_train)

```

Test Accuracy: 0.7625
Train Accuracy: 0.8125



4.2 Polynomial Kernel, $r = 1$, degree = 2, $C = 100$

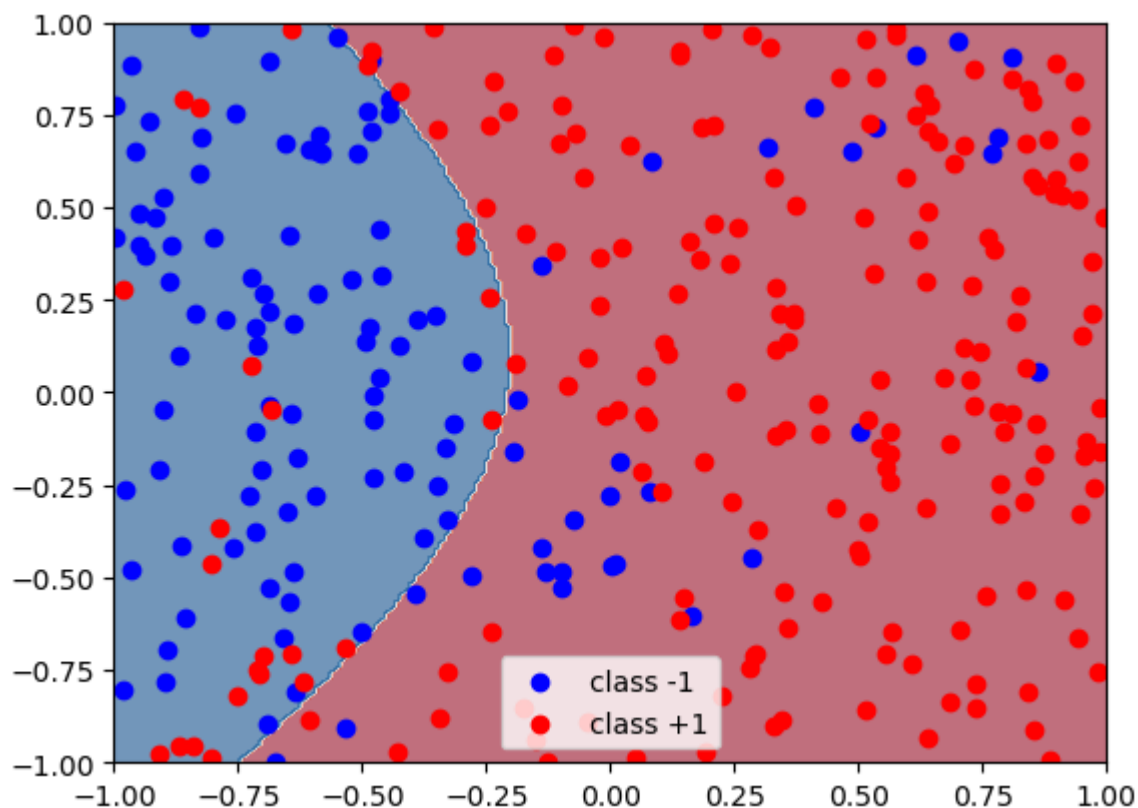
```
In [ ]: clf = SVC(C=100,kernel="poly",degree = 2, coef0 = 1)
clf.fit(X_train,Y_train)

train_accuracy = clf.score(X_train,Y_train)
test_accuracy = clf.score(X_test,Y_test)

out_str = "Test Accuracy: {}\nTrain Accuracy: {}".format(test_accuracy,train_a
print(out_str)

plotClassifier(clf,X_train,Y_train)
```

Test Accuracy: 0.775
Train Accuracy: 0.828125



4.3 RBF Kernel with $\sigma = 0.5$, $C = 100$

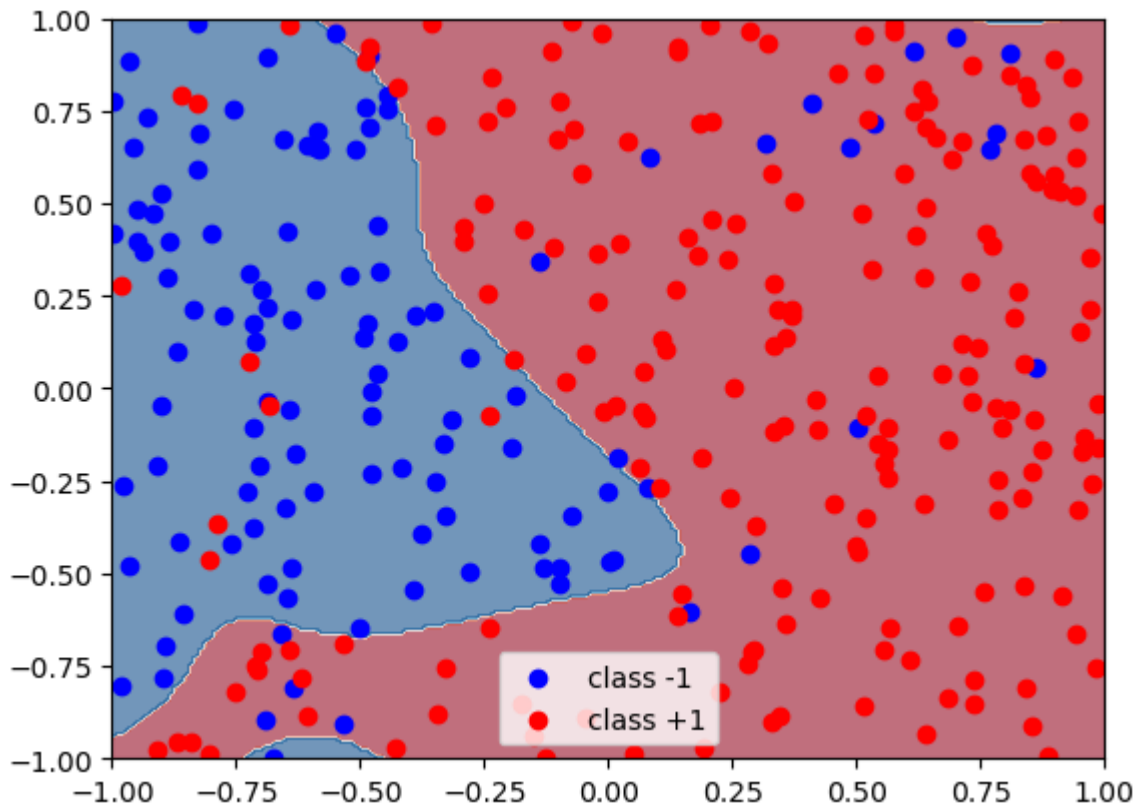
```
In [ ]: clf = SVC(C=100, kernel="rbf", gamma=1/pow(0.5,2))
clf.fit(X_train, Y_train)

train_accuracy = clf.score(X_train, Y_train)
test_accuracy = clf.score(X_test, Y_test)

out_str = "Test Accuracy: {}\nTrain Accuracy: {}".format(test_accuracy, train_a
print(out_str)

plotClassifier(clf, X_train, Y_train)
```

Test Accuracy: 0.85
Train Accuracy: 0.9



4.4 RBF Kernel Performance over several values of gamma

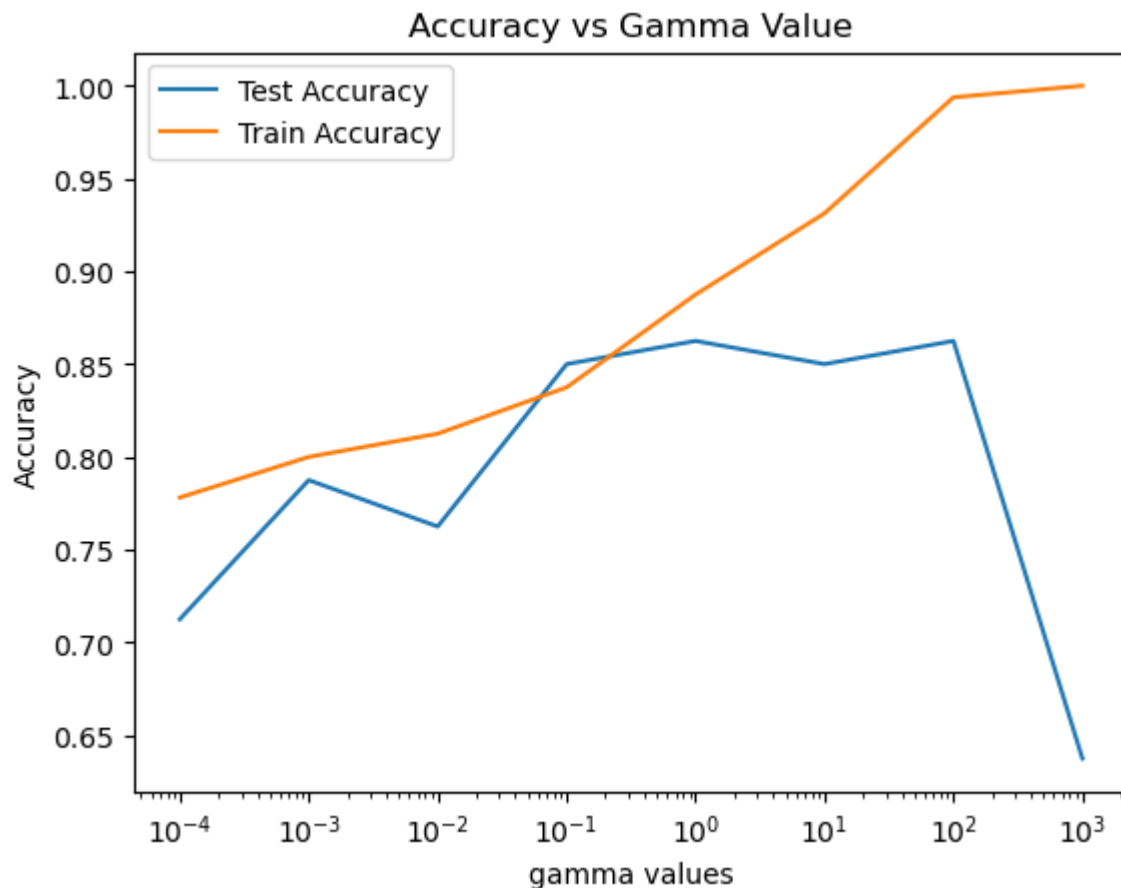
```
In [ ]: gamma_vals = np.array([1e-4, 1e-3, 1e-2, 1e-1, 1, 1e1, 1e2, 1e3])
test_accuracies = np.zeros(np.shape(gamma_vals))
train_accuracies = np.zeros(np.shape(gamma_vals))

for i in range(np.size(gamma_vals)):
    clf = SVC(C=100, kernel="rbf", gamma=gamma_vals[i])
    clf.fit(X_train, Y_train)

    train_accuracies[i] = clf.score(X_train, Y_train)
    test_accuracies[i] = clf.score(X_test, Y_test)

fig, ax = plt.subplots()
ax.plot(gamma_vals, test_accuracies, label = "Test Accuracy")
ax.plot(gamma_vals, train_accuracies, label = "Train Accuracy")
ax.set_xlabel("gamma values")
ax.set_xscale("log")
ax.set_ylabel("Accuracy")
ax.set_title("Accuracy vs Gamma Value")
ax.legend()
```

```
Out[ ]: <matplotlib.legend.Legend at 0x7f21aefdbac0>
```



As gamma increases, the training error goes to 100 while the testing error decreases. This indicates that the model is overfitting at high values of gamma.

The reason that the model overfits for high values of gamma is due to the fact that the kernel is defined with $\exp(-\gamma \|x_i - x_j\|^2)$. Given this, for large values of gamma, the kernel term basically goes to zero leaving only the regularization term. Given this, the model then tries to exactly fit the training data which results in the model overfitting.

For small values of gamma, there is a high variance. This means that the model is more generalized and not overfit at the expense of not classifying all of the points in the training set perfectly.

4.5 L2 regularized Logistic Regression, Linear Kernel, lambda = 0.01

```
In [ ]: clf = kernelLogRegL2(lammy=0.01)
         clf.fit(X_train,Y_train)

def get_accuracy(model,X,Y):
    f_x = model.predict(X)
    correct = f_x[f_x == Y]
    return float(np.size(correct))/float(np.size(Y))

train_accuracy = get_accuracy(clf,X_train,Y_train)
test_accuracy = get_accuracy(clf,X_test,Y_test)
```



```
out_str = "Test Accuracy: {}\nTrain Accuracy: {}".format(test_accuracy, train_a
print(out_str)
```

```
plotClassifier(clf,X_train,Y_train)
```

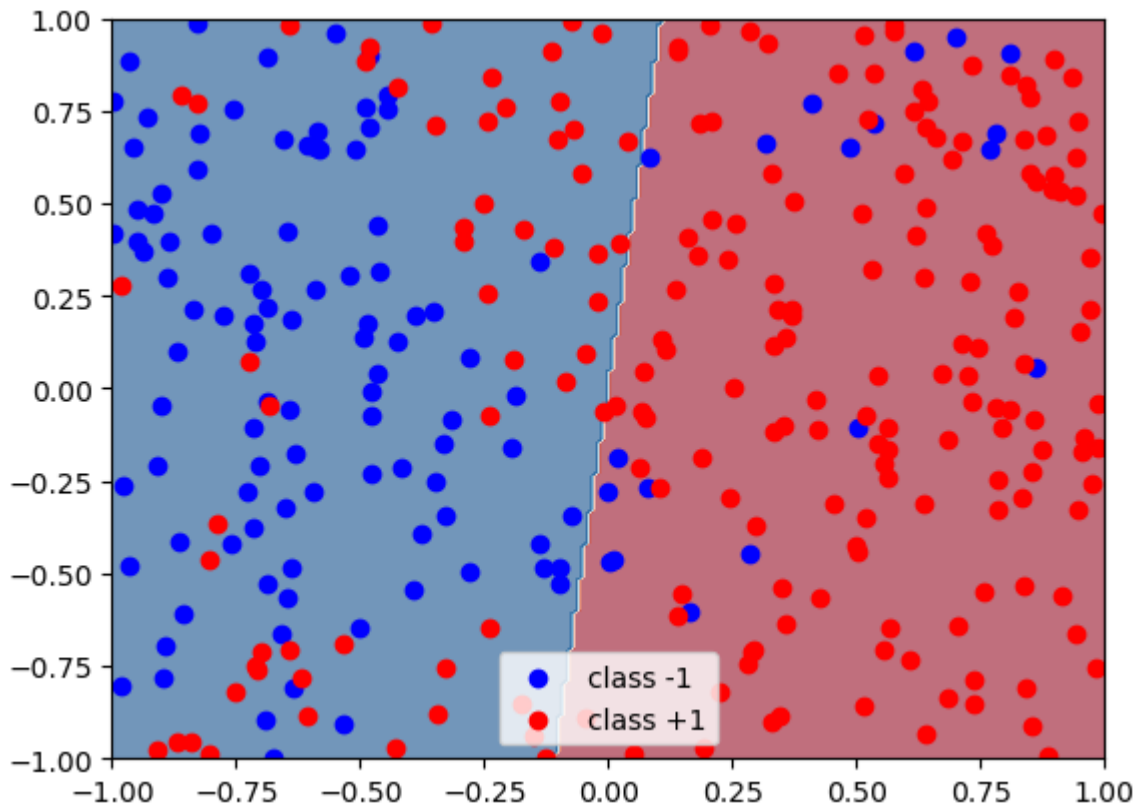
```
Test Accuracy: 0.7625
Train Accuracy: 0.7625
```

```
/tmp/ipykernel_11428/470417373.py:2: RuntimeWarning: overflow encountered in e
xp
```

```
out = np.log(1+np.exp(x))
```

```
/tmp/ipykernel_11428/2422820963.py:20: RuntimeWarning: overflow encountered in
exp
```

```
res = - y / (1. + np.exp(yKu))
```



4.6: L2 Regularized Logistic Regression with polynomial kernel, $\lambda = 0.01$

```
In [ ]: def kernel_poly(X1, X2, p=2):
        return np.power(np.add(1, np.matmul(X1,X2.transpose())) ,p)
```

```
In [ ]: clf = kernelLogRegL2(lammy=0.01, kernel_fun=kernel_poly, p=3)
        clf.fit(X_train, Y_train)
```

```
def get_accuracy(model, X, Y):
    f_x = model.predict(X)
    correct = f_x[f_x == Y]
    return float(np.size(correct))/float(np.size(Y))
```

```
train_accuracy = get_accuracy(clf, X_train, Y_train)
test_accuracy = get_accuracy(clf, X_test, Y_test)
```

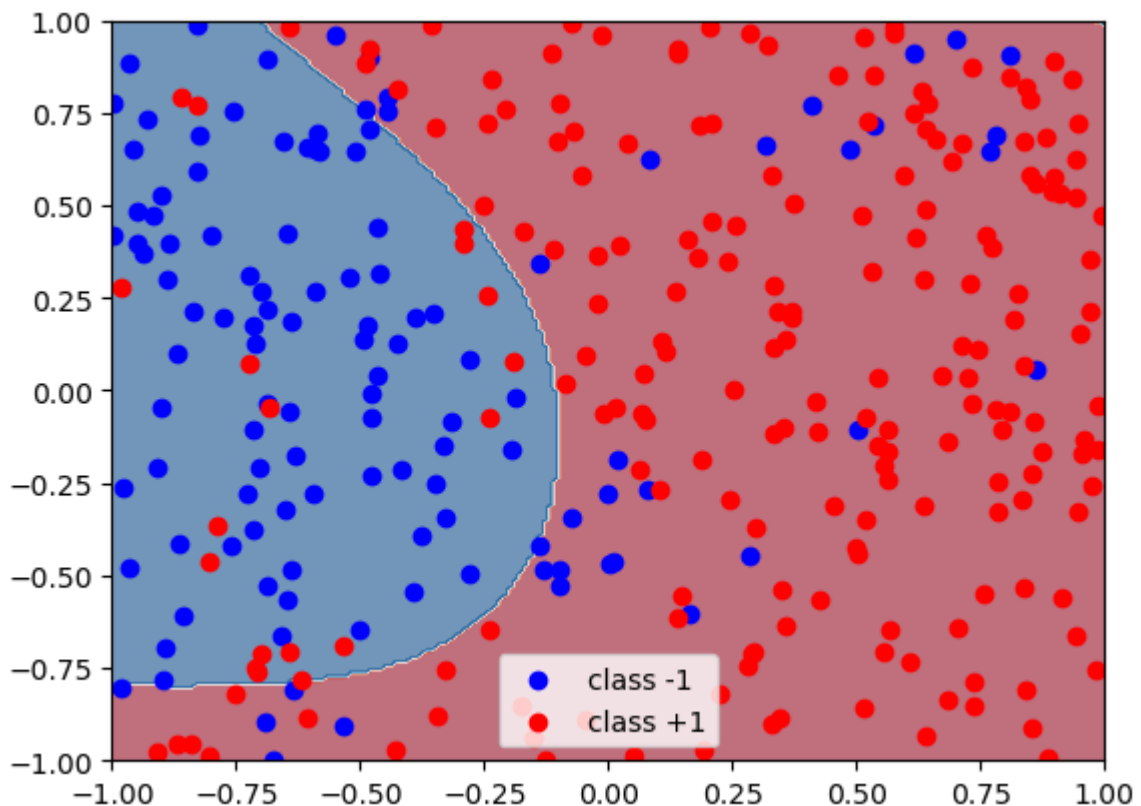
```
out_str = "Test Accuracy: {}\nTrain Accuracy: {}".format(test_accuracy, train_a
```

```
print(out_str)

plotClassifier(clf,X_train,Y_train)
```

Test Accuracy: 0.8625
Train Accuracy: 0.834375

```
/tmp/ipykernel_11428/470417373.py:2: RuntimeWarning: overflow encountered in e
xp
  out = np.log(1+np.exp(x))
/tmp/ipykernel_11428/2422820963.py:20: RuntimeWarning: overflow encountered in
exp
  res = - y / (1. + np.exp(yKu))
```



4.7 L2 Regularized Logistic Regression with RBF Kernel, sigma = 0.5, Lambda = 0.01

```
In [ ]: def kernel_RBF(X1, X2, gamma = 1):
    #Your code here
    n = np.shape(X1)[0]
    p = np.shape(X2)[0]
    K = np.zeros((n,p))

    #while inefficient, use a for loop to compute k
    for r in range(n):
        for c in range(p):
            K[r,c] = np.exp(-1 * gamma * pow(np.linalg.norm(X1[r] - X2[c]),2))

    return K
```

```
In [ ]: clf = kernelLogRegL2(lammy=0.01,kernel_fun=kernel_RBF,gamma = 1/pow(0.5,2))
        clf.fit(X_train,Y_train)
```

```
def get_accuracy(model,X,Y):
    f_x = model.predict(X)
    correct = f_x[f_x == Y]
    return float(np.size(correct))/float(np.size(Y))

train_accuracy = get_accuracy(clf,X_train,Y_train)
test_accuracy = get_accuracy(clf,X_test,Y_test)

out_str = "Test Accuracy: {}\nTrain Accuracy: {}".format(test_accuracy,train_a
print(out_str)

plotClassifier(clf,X_train,Y_train)
```

/tmp/ipykernel_11428/470417373.py:2: RuntimeWarning: overflow encountered in exp

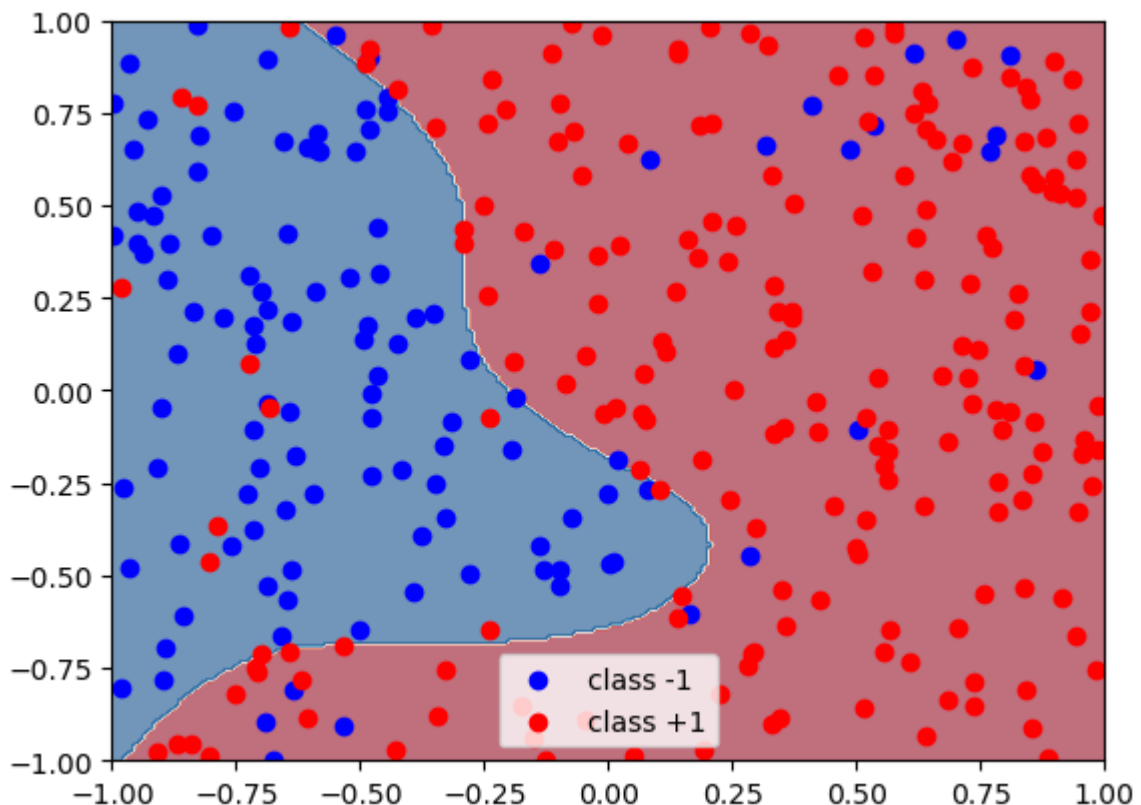
```
    out = np.log(1+np.exp(x))
```

/tmp/ipykernel_11428/2422820963.py:20: RuntimeWarning: overflow encountered in exp

```
    res = - y / (1. + np.exp(yKu))
```

Test Accuracy: 0.85

Train Accuracy: 0.8875



4.8 RBF Kernel with various values for gamma

```
In [ ]: gamma_vals = np.array([1e-4,1e-3,1e-2,1e-1,1,1e1,1e2,1e3])
test_accuracies = np.zeros(np.shape(gamma_vals))
train_accuracies = np.zeros(np.shape(gamma_vals))

for i in range(np.size(gamma_vals)):
    clf = kernelLogRegL2(lammy=0.01,kernel_fun=kernel_RBF,gamma = gamma_vals[i]
    clf.fit(X_train,Y_train)

    train_accuracies[i] = get_accuracy(clf,X_train,Y_train)
```

```

test_accuracies[i] = get_accuracy(clf,X_test,Y_test)

fig,ax = plt.subplots()
ax.plot(gamma_vals,test_accuracies,label = "Test Accuracy")
ax.plot(gamma_vals,train_accuracies,label = "Train Accuracy")
ax.set_xlabel("gamma values")
ax.set_xscale("log")
ax.set_ylabel("Accuracy")
ax.set_title("Accuracy vs Gamma Value")
ax.legend()

```

```

/tmp/ipykernel_11428/470417373.py:2: RuntimeWarning: overflow encountered in exp

```

```

    out = np.log(1+np.exp(x))

```

```

/tmp/ipykernel_11428/2422820963.py:20: RuntimeWarning: overflow encountered in exp

```

```

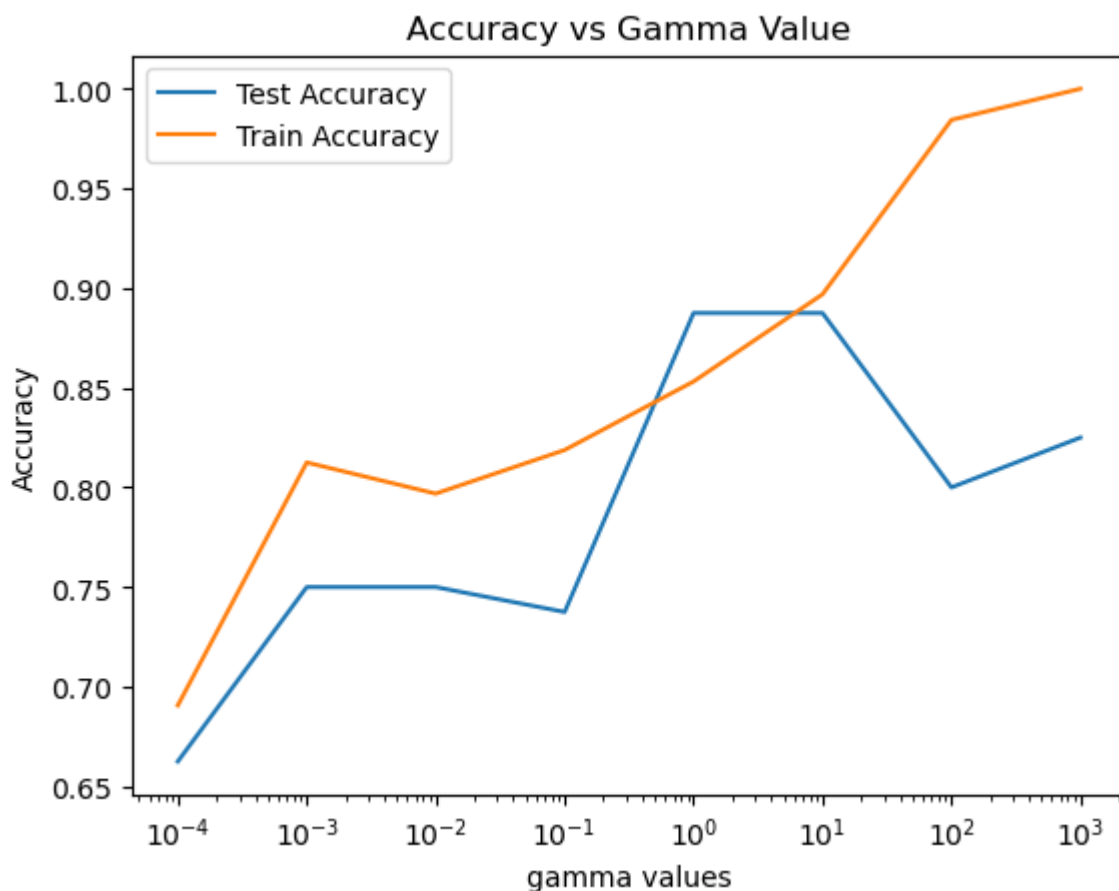
    res = - y / (1. + np.exp(yKu))

```

```

Out[ ]: <matplotlib.legend.Legend at 0x7f21ac242e50>

```



Problem 5 Sparse Logistic Regression

5.1 ROC curves for PECTF dataset

```

In [ ]: import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve
from sklearn.metrics import auc

```

```
import matplotlib.pyplot as plt

#import the nonlineardata.csv file
df = pd.read_csv("spectf.csv",header=None)

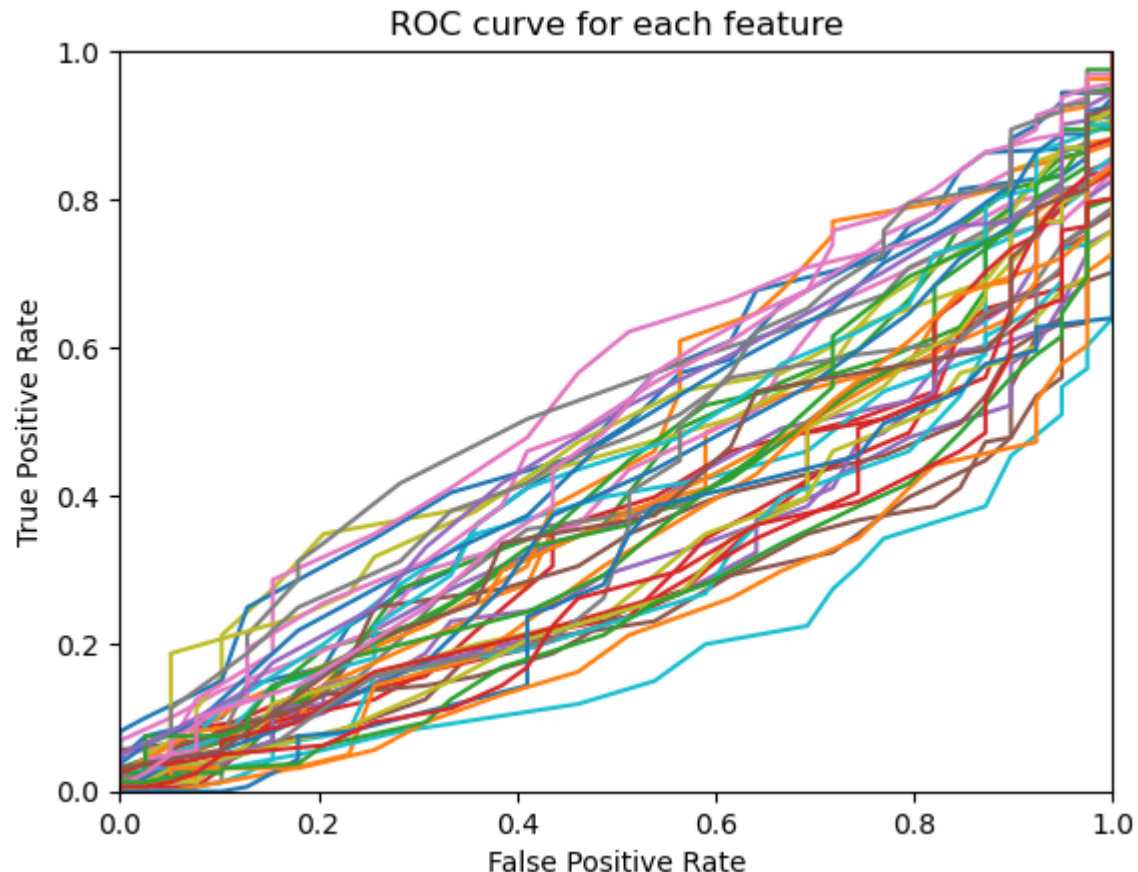
#access the x and y data sets
data = df.to_numpy()
X = data[:,1:]
Y = data[:,0]

#split the data into 80% training and 20% testing with random state to 2022
X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size=0.25,random
```

```
In [ ]: num_features = np.shape(X)[1]
feature_AUCs = np.zeros(num_features)

#initialize the plot
fig,ax = plt.subplots()
for i in range(num_features):
    fpr,tpr,thresholds = roc_curve(Y_train,X_train[:,i])
    feature_AUCs[i] = auc(fpr,tpr)
    ax.plot(fpr,tpr)
ax.set_xlabel("False Positive Rate")
ax.set_ylabel("True Positive Rate")
ax.set_xlim(0,1)
ax.set_ylim(0,1)
ax.set_title("ROC curve for each feature")
```

```
Out[ ]: Text(0.5, 1.0, 'ROC curve for each feature')
```



5.2 Creating 300 different subsets of features

To start, I recognize that the best features will have an AUC furthest from 0.5. For example, on the above plot, there are some features with an AUC well below 0.5, but this just means that the negation of the prediction was right more often than not (which is still useful). Given this, I prioritize the features with AUC furthest from 0.5. To do this, I sorted the list of features with AUC's furthest from 0.5

Next, I decided to create 150 subsets with 2 features and 150 subsets with 3 features.

For the subsets with 2 features, the first feature was randomly selected from the best 25% of features while the second feature was randomly selected from all of the features.

For the subsets with 3 features, the first feature was randomly selected from the best 15%, the second was randomly selected from the best 30%, and the third was randomly selected from all of the features

```
In [ ]: def get_subset_feature_indicies(num_features,best_AUC_indicies,max_percentile_
        #array to track the feature indicies
        feature_indicies = []

        #variable for the selected index
        selected_index = 0
        for percentile in max_percentile_for_features:
            #for the current percentile, randomly select an index that hasn't yet
            max_idx = np.floor(percentile * num_features)
            idx = np.random.randint(0,max_idx)
            while best_AUC_indicies[idx] in feature_indicies:
                idx = np.random.randint(0,max_idx)

            #once a feature has been found that hasn't been previously selected, a
            feature_indicies.append(best_AUC_indicies[idx])

        return feature_indicies
```

```
In [ ]: #sort for the features with the AUC furthest away from 0.5
best_AUC_indicies = np.flip(np.argsort(np.absolute(np.subtract(feature_AUCs,0.
print(feature_AUCs[best_AUC_indicies]))

feature_indicies = []

#randomly select feature indicies with 2 features
max_percentile_for_features = [0.10,0.25]
for i in range(0,150):
    feature_indicies.append(get_subset_feature_indicies(num_features,best_AUC_

#randomly select feature indicies with 3 features
max_percentile_for_features = [0.05,0.15,0.30]
for i in range(0,150):
    feature_indicies.append(get_subset_feature_indicies(num_features,best_AUC_
```

```
[0.19533365 0.22893773 0.25863991 0.25911769 0.26039178 0.27695493
0.29009396 0.29081064 0.2938366 0.29964963 0.30737379 0.31278866
0.31692945 0.3212295 0.33548336 0.33580188 0.33882784 0.35570951
0.36391145 0.37641344 0.37991718 0.38469502 0.38573021 0.39010989
0.39241918 0.39433031 0.40149705 0.41567129 0.41774168 0.41837872
0.43629559 0.44051601 0.44792164 0.45859213 0.47396082 0.52500398
0.47642937 0.5226947 0.47802198 0.48041089 0.48096831 0.48550725
0.50708712 0.50294633]
```

5.3 Train Logistic regression on all 3 features

```
In [ ]: from sklearn.linear_model import LogisticRegression

log_regression_models = []
log_regression_AUCs = np.zeros((300,1))

#initialize the plot
fig,ax = plt.subplots()

for i in range(len(feature_indicies)):

    features = feature_indicies[i]
    X_vals = X_train[:,features]
    log_regression_models.append(LogisticRegression(penalty="none"))
    log_regression_models[i].fit(X_vals,Y_train)
    f_x = log_regression_models[i].predict_proba(X_vals)

    #plot the ROC curve and save the AUC value

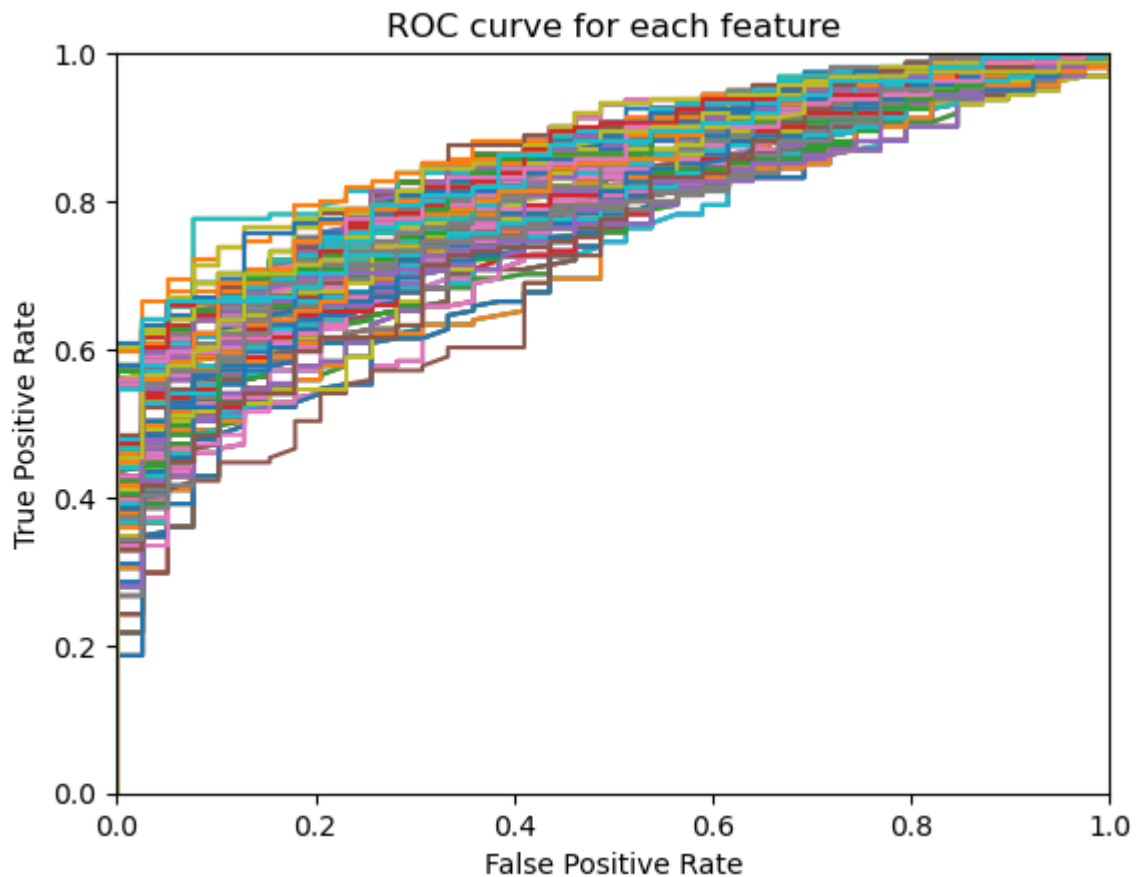
    fpr,tpr,thresholds = roc_curve(Y_train,f_x[:,1])
    log_regression_AUCs[i] = (auc(fpr,tpr))
    ax.plot(fpr,tpr)

    ax.set_xlabel("False Positive Rate")
    ax.set_ylabel("True Positive Rate")
    ax.set_xlim(0,1)
    ax.set_ylim(0,1)
    ax.set_title("ROC curve for each feature")

    max_auc_idx = np.argmax(log_regression_AUCs)
    f = log_regression_models[max_auc_idx]
    f_feature_indicies = feature_indicies[max_auc_idx]

    out_str = "Best Model: AUC = {}, using feature indicies: {}".format(log_regres
print(out_str)
```

```
Best Model: AUC = [0.87609492], using feature indicies: [39, 25, 33]
```

5.4 Analyzing Best Model

```
In [ ]: # plot the test and train ROC for f on the same figure
X_vals_train = X_train[:,f_feature_indicies]
X_vals_test = X_test[:,f_feature_indicies]

f_x_train = f.predict_proba(X_vals_train)[:,1]
f_x_test = f.predict_proba(X_vals_test)[:,1]

#plot the ROC curve and save the AUC value
fig,ax = plt.subplots()

#plot training ROC
fpr,tpr,thresholds = roc_curve(Y_train,f_x_train)
train_auc = auc(fpr,tpr)
ax.plot(fpr,tpr,label = "Training Set")

fpr,tpr,thresholds = roc_curve(Y_test,f_x_test)
test_auc = auc(fpr,tpr)
ax.plot(fpr,tpr,label = "Test Set")

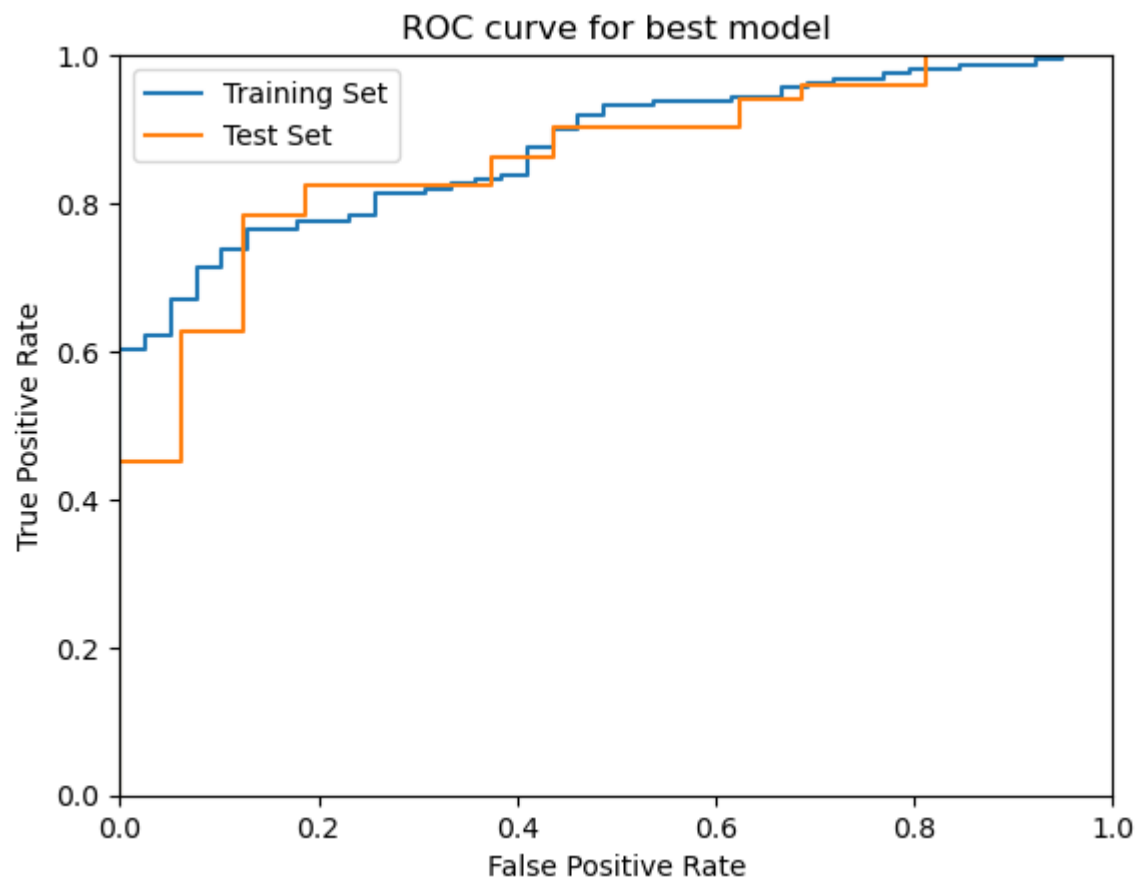
ax.set_xlabel("False Positive Rate")
ax.set_ylabel("True Positive Rate")
ax.set_xlim(0,1)
ax.set_ylim(0,1)
ax.set_title("ROC curve for best model")
ax.legend()

#report test and train AUC
```



```
out_str = "Test AUC: {} Train AUC: {}".format(test_auc,train_auc)
print(out_str)
```

Test AUC: 0.8602941176470589 Train AUC: 0.8760949195731804



```
In [ ]: # compare to regularized logistic regression
clf = LogisticRegression(penalty= "l2",C=0.01)
clf.fit(X_train,Y_train)
f_x_train = clf.predict_proba(X_train)[: ,1]
f_x_test = clf.predict_proba(X_test)[: ,1]

#plot the ROC curve and save the AUC value
fig,ax = plt.subplots()

#plot training ROC
fpr,tpr,thresholds = roc_curve(Y_train,f_x_train)
train_auc = auc(fpr,tpr)
ax.plot(fpr,tpr,label = "Training Set")

fpr,tpr,thresholds = roc_curve(Y_test,f_x_test)
test_auc = auc(fpr,tpr)
ax.plot(fpr,tpr,label = "Test Set")

ax.set_xlabel("False Positive Rate")
ax.set_ylabel("True Positive Rate")
ax.set_xlim(0,1)
ax.set_ylim(0,1)
ax.set_title("ROC curve for best model")
ax.legend()

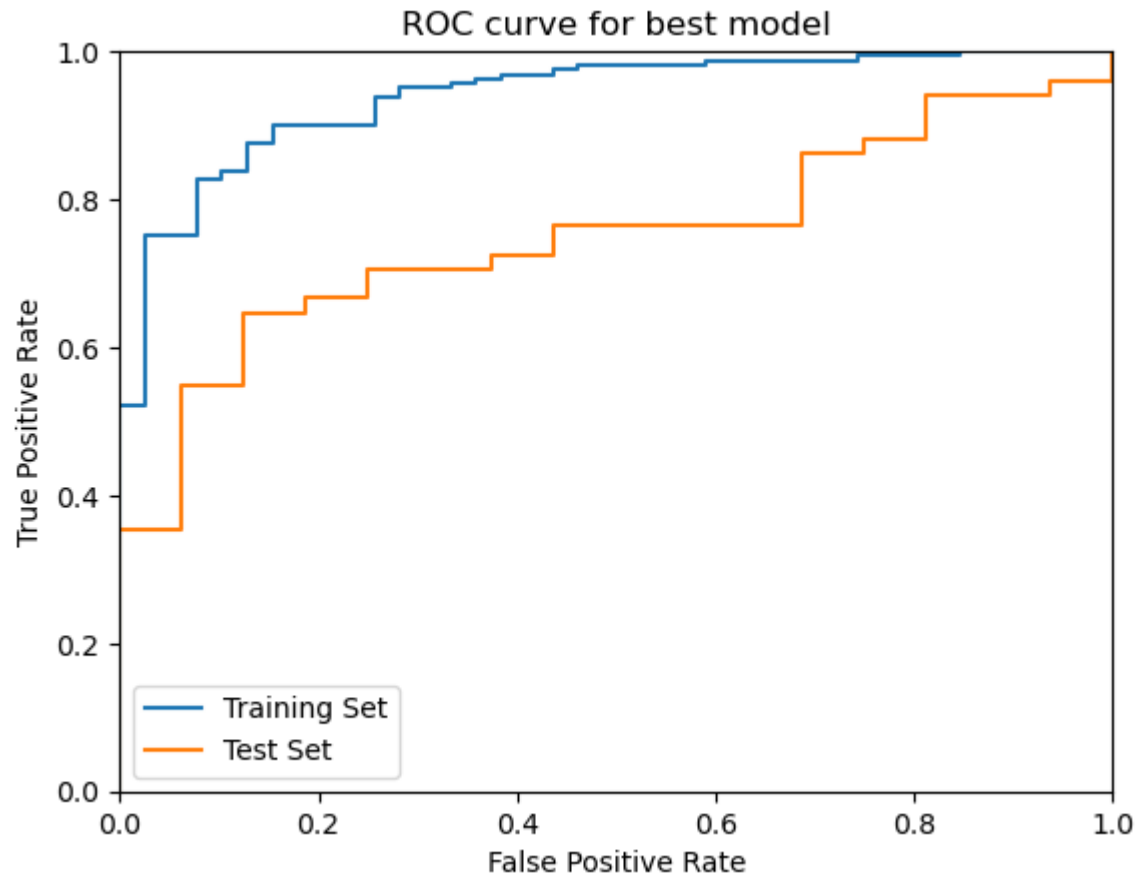
#report test and train AUC
```

```
out_str = "Test AUC: {} Train AUC: {}".format(test_auc, train_auc)
print(out_str)
```

Test AUC: 0.75 Train AUC: 0.9396400700748527

/home/david/anaconda3/lib/python3.9/site-packages/sklearn/linear_model/_logistic.py:814: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
<https://scikit-learn.org/stable/modules/preprocessing.html>
Please also refer to the documentation for alternative solver options:
https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
n_iter_i = _check_optimize_result(



My model generalized far better than l2-regularized logistic regression did. Given this, the results show that even with 3 features, sparse-logistic regression will perform just as good (on the test set), if not better, than regularized l2 logistic regression.

6 Ridge Regression and Friends

6.1 Comparing Ridge Regression to Kernel Ridge Regression

```
In [ ]: #generate a data set with 100 samples with 5e6 features in the distribution
from sklearn.datasets import make_regression
```

```
X,Y = make_regression(
    n_samples = 100,
    n_features= 5000000,
    n_informative= 100,
    noise= 50
)
```

```
In [ ]: # training with Ridge Regression
from sklearn.linear_model import Ridge

clf = Ridge(alpha = 0.01)
clf.fit(X,Y)
```

```
Out[ ]: Ridge(alpha=0.01)
```

```
In [ ]: from sklearn.kernel_ridge import KernelRidge

clf = KernelRidge(kernel="linear",alpha=0.01)
clf.fit(X,Y)
```

```
Out[ ]: KernelRidge(alpha=0.01)
```

See overleaf part of answer for explanation on times

6.2 Ridge Regression and Lasso Regression

```
In [ ]: # create a data set with 1000 samples and 500 features
from sklearn.datasets import make_regression

X,Y = make_regression(
    n_samples = 1000,
    n_features= 500,
    n_informative= 10,
    noise= 30
)

X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size=0.2,random_
```

```
In [ ]: from sklearn.linear_model import Ridge

alpha_vals = np.array([1e-4,1e-3,1e-2,1e-1,1,1e1,1e2,1e3])

# fit ridge regression

#arrays to track accuracies and coefficient vectors
test_accuracies = np.zeros(np.shape(alpha_vals))
train_accuracies = np.zeros(np.shape(alpha_vals))
ridge_coef_vals = np.zeros((np.shape(alpha_vals)[0],np.shape(X_train)[1]))

for i in range(np.size(alpha_vals)):
    clf = Ridge(alpha=alpha_vals[i])
    clf.fit(X_train,Y_train)

    train_accuracies[i] = clf.score(X_train,Y_train)
    test_accuracies[i] = clf.score(X_test,Y_test)
```

```

ridge_coef_vals[i,:] = clf.coef_

fig,ax = plt.subplots(2)
fig.tight_layout(pad=3.0)

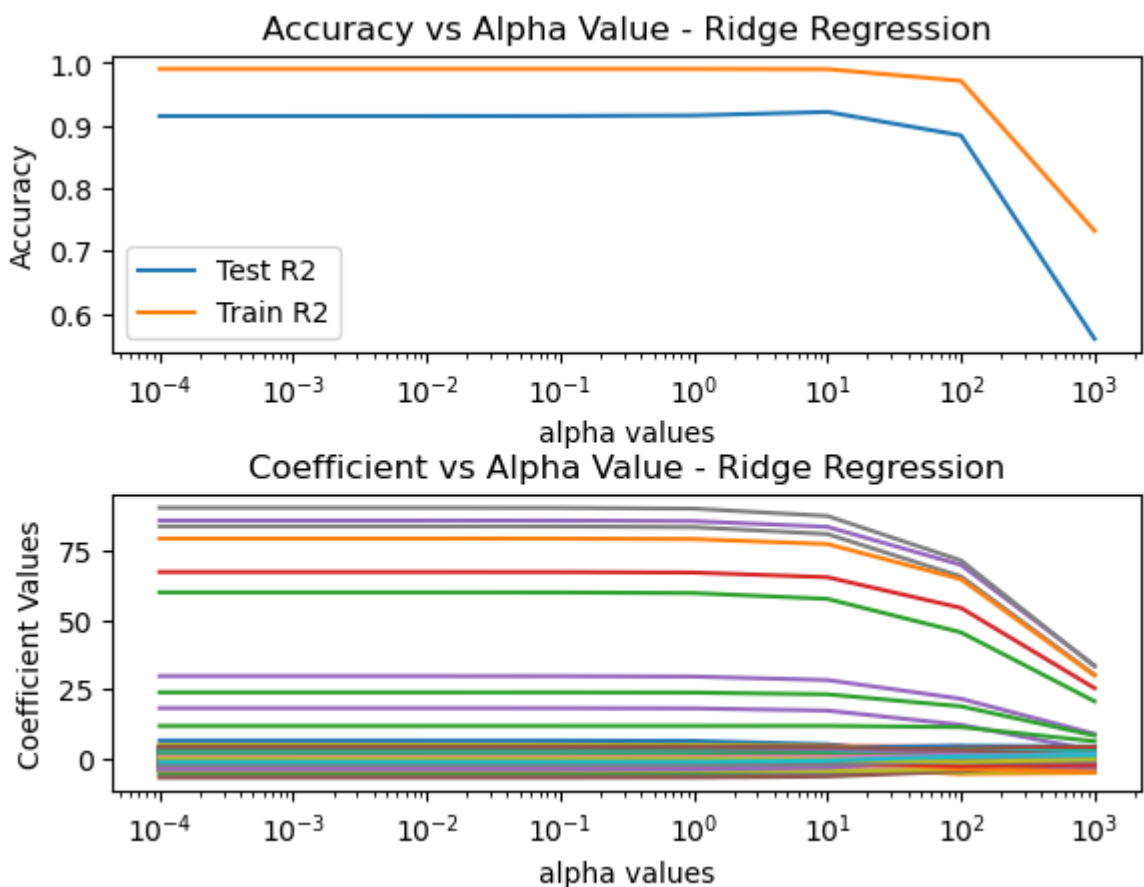
#plot the accuracies
ax[0].plot(alpha_vals,test_accuracies,label = "Test R2")
ax[0].plot(alpha_vals,train_accuracies,label = "Train R2")
ax[0].set_xlabel("alpha values")
ax[0].set_ylabel("Accuracy")
ax[0].set_xscale("log")
ax[0].set_title("Accuracy vs Alpha Value - Ridge Regression")
ax[0].legend()

#plot the coefficients
for i in range(np.shape(X_train)[1]):
    ax[1].plot(alpha_vals,ridge_coef_vals[:,i])

ax[1].set_title("Coefficient vs Alpha Value - Ridge Regression")
ax[1].set_xlabel("alpha values")
ax[1].set_xscale("log")
ax[1].set_ylabel("Coefficient Values")

```

Out[]: Text(42.59722222222214, 0.5, 'Coefficient Values')



```

In [ ]: from sklearn.linear_model import Lasso

# fit ridge regression

#arrays to track accuracies and coefficient vectors
test_accuracies = np.zeros(np.shape(alpha_vals))

```

```

train_accuracies = np.zeros(np.shape(alpha_vals))
lasso_coef_vals = np.zeros((np.shape(alpha_vals)[0], np.shape(X_train)[1]))

for i in range(np.size(alpha_vals)):
    clf = Lasso(alpha=alpha_vals[i])
    clf.fit(X_train, Y_train)

    train_accuracies[i] = clf.score(X_train, Y_train)
    test_accuracies[i] = clf.score(X_test, Y_test)
    lasso_coef_vals[i, :] = clf.coef_

fig, ax = plt.subplots(2)
fig.tight_layout(pad=3.0)

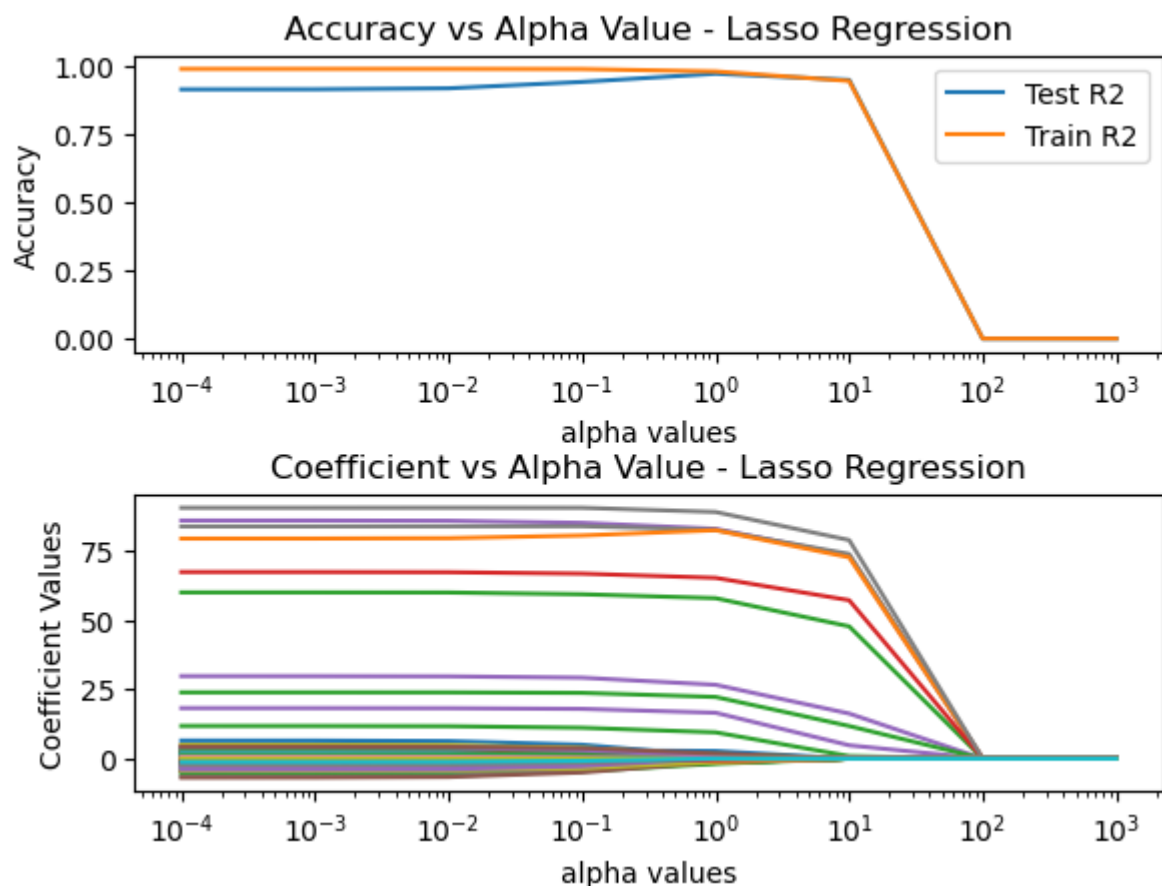
#plot the accuracies
ax[0].plot(alpha_vals, test_accuracies, label = "Test R2")
ax[0].plot(alpha_vals, train_accuracies, label = "Train R2")
ax[0].set_xlabel("alpha values")
ax[0].set_ylabel("Accuracy")
ax[0].set_xscale("log")
ax[0].set_title("Accuracy vs Alpha Value - Lasso Regression")
ax[0].legend()

#plot the coefficients
for i in range(np.shape(X_train)[1]):
    ax[1].plot(alpha_vals, lasso_coef_vals[:, i])

ax[1].set_title("Coefficient vs Alpha Value - Lasso Regression")
ax[1].set_xlabel("alpha values")
ax[1].set_xscale("log")
ax[1].set_ylabel("Coefficient Values")

```

Out[]: Text(42.59722222222214, 0.5, 'Coefficient Values')



I simulated a data set with quite a bit of added noise and only 10 out of the 500 features being informative (done so that I could better see the effect of l_1 vs l_2 regularization).

As the strength of regularization increased, the coefficients in the lambda decreased.

Based on my observations, one possible advantage of Lasso over Ridge Regression is that it can automatically perform feature selection by nature of using the l_1 norm instead of the l_2 norm. Additionally, even though Lasso uses less features, it still seemed to fit the simulated model quite well.

6.3: Regularization Paths for Lasso and Ridge Regression

```
In [ ]: fig, ax = plt.subplots(2)
fig.tight_layout(pad=3.0)

#plot the ridge coefficients
for i in range(np.shape(X_train)[1]):
    ax[0].plot(alpha_vals, ridge_coef_vals[:,i])
    ax[1].plot(alpha_vals, lasso_coef_vals[:,i])

ax[0].set_title("Coefficient vs Alpha Value - Ridge Regression")
ax[0].set_xlabel("alpha values")
ax[0].set_xscale("log")
ax[0].set_ylabel("Coefficient Values")
```

```
#plot the Lasso
ax[1].set_title("Coefficient vs Alpha Value - Lasso Regression")
ax[1].set_xlabel("alpha values")
ax[1].set_xscale("log")
ax[1].set_ylabel("Coefficient Values")
```

Out[]: Text(42.59722222222214, 0.5, 'Coefficient Values')

