index

March 11, 2022

0.1 Logistic Regression Model Comparisons - Lab

0.2 Introduction

In this lab, you'll further investigate how to tune your own logistic regression implementation, as well as that of scikit-learn in order to produce better models.

0.3 Objectives

- Compare the different inputs with logistic regression models and determine the optimal model

In the previous lab, you were able to compare the output of your own implementation of the logistic regression model with that of scikit-learn. However, that model did not include an intercept or any regularization. In this investigative lab, you will analyze the impact of these two tuning parameters.

0.4 Import the data

As with the previous lab, import the dataset stored in `'heart.csv'`:

[1]:
```
# Import the data
import pandas as pd
df = pd.read_csv('heart.csv')

# Print the first five rows of the data
df.head()
```

[1]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	0.708333	1.0	1.000000	0.481132	0.244292	1.0	0.0	0.603053	0.0	0.370968	0.0	0.0	0.333333	1.0
1	0.166667	1.0	0.666667	0.339623	0.283105	0.0	0.5	0.885496	0.0	0.564516	0.0	0.0	0.666667	1.0
2	0.250000	0.0	0.333333	0.339623	0.178082	0.0	0.0	0.770992	0.0	0.225806	1.0	0.0	0.666667	1.0
3	0.562500	1.0	0.333333	0.245283	0.251142	0.0	0.5	0.816794	0.0	0.129032	1.0	0.0	0.666667	1.0
4	0.583333	0.0	0.000000	0.245283	0.520548	0.0	0.5	0.702290	1.0	0.096774	1.0	0.0	0.666667	1.0

0.5 Split the data

Define `X` and `y` as with the previous lab. This time, follow best practices and also implement a standard train-test split. Assign 25% to the test set and set the `random_state` to 17.

[2]:
```
# Define X and y
from sklearn.model_selection import train_test_split

y = df["target"]
X = df.drop("target", axis = 1)

# Split the data into training and test sets


X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = 0.25,
                                                    random_state = 17)
print(y_train.value_counts(),'\n\n', y_test.value_counts())
```

```
1.0    130
0.0     97
Name: target, dtype: int64

 0.0    41
1.0    35
Name: target, dtype: int64
```

0.6 Initial Model - Personal Implementation

Use your code from the previous lab to once again train a logistic regression algorithm on the training set.

[3]:
```
# Your code from previous lab
import numpy as np

def sigmoid(x):
    x = np.array(x)
    return 1/(1 + np.e**(-1*x))

def grad_desc(X, y, max_iterations, alpha, initial_weights=None):
    """Be sure to set default behavior for the initial_weights parameter."""
    if initial_weights is None:
        initial_weights = np.ones((X.shape[1], 1)).flatten()
    weights_col = pd.DataFrame(initial_weights)
    weights = initial_weights
    # Create a for loop of iterations
    for iteration in range(max_iterations):
        # Generate predictions using the current feature weights

        predictions = sigmoid(np.dot(X, weights))
```

```
        # Calculate an error vector based on these initial predictions and
        # the correct labels

        error_vector = y - predictions

        # Calculate the gradient
        # As we saw in the previous lab, calculating the gradient is often
        # the most difficult task.
        # Here, your are provided with the closed form solution for the
        # gradient of the log-loss function derived from MLE
        # For more details on the derivation, see the additional resources
        # section below.

        gradient = np.dot(X.transpose(), error_vector)
        # Update the weight vector take a step of alpha in direction of␣
 ↪gradient
        weights += alpha * gradient
        weights_col = pd.concat([weights_col, pd.DataFrame(weights)], axis=1)
    # Return finalized weights
    return weights, weights_col

weights, weights_col = grad_desc(X_train, y_train, 50000, 0.001)
```

0.7 Make [probability] predictions on the test set

[5]:
```
# Predict on test set

y_hat_test = sigmoid(np.dot(X_test, weights))
np.round(y_hat_test, 2)
```

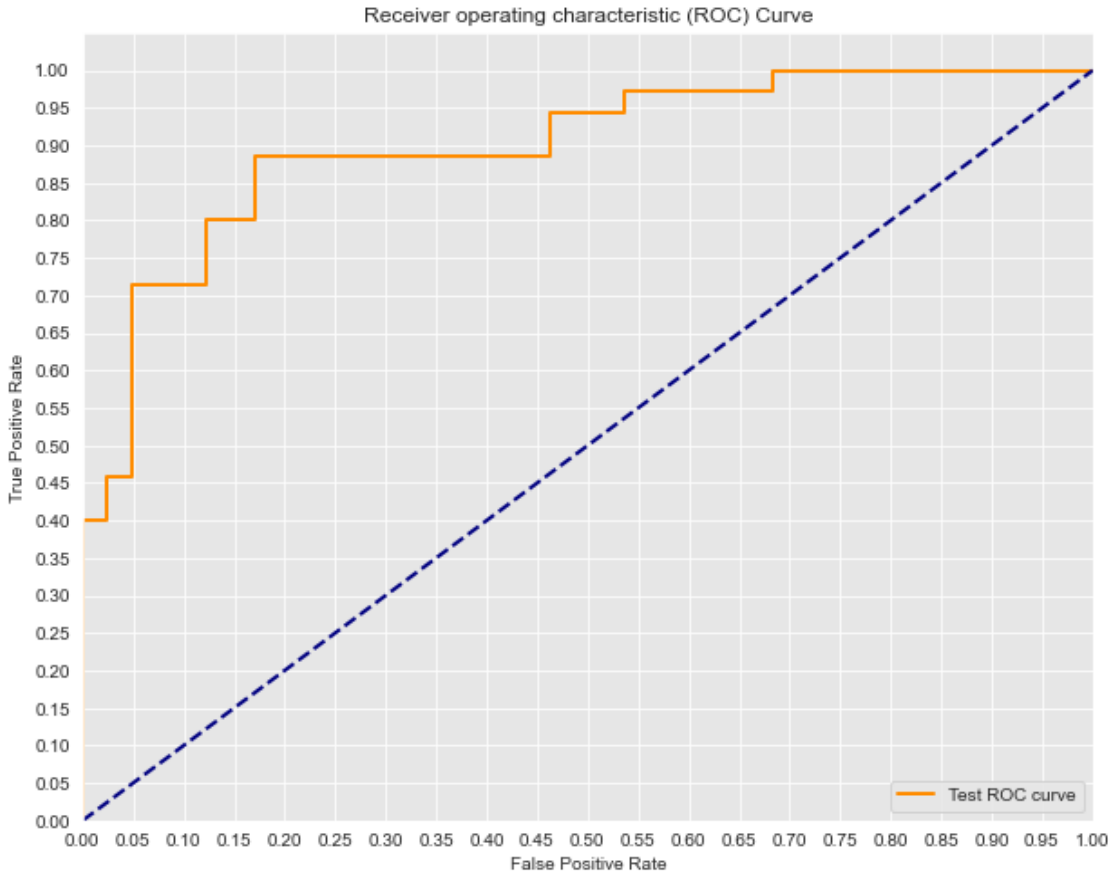
[5]:
```
array([0.96, 0.02, 0.09, 0.12, 0.  , 1.  , 0.25, 0.94, 0.  , 0.8 , 0.04,
       0.69, 0.53, 0.  , 0.99, 0.59, 0.69, 0.01, 0.99, 0.03, 0.98, 0.98,
       0.03, 0.78, 0.76, 0.78, 0.  , 0.08, 0.02, 0.01, 0.74, 0.02, 0.99,
       0.05, 0.35, 0.99, 0.85, 0.31, 0.78, 0.99, 0.97, 0.14, 0.  , 0.01,
       0.96, 0.9 , 0.98, 0.73, 0.02, 0.  , 0.98, 0.  , 0.  , 0.68, 0.85,
       0.  , 0.66, 0.6 , 0.01, 0.97, 0.07, 0.  , 0.98, 0.43, 0.91, 0.08,
       0.81, 0.99, 0.01, 0.26, 0.68, 0.18, 0.98, 0.02, 0.96, 0.94])
```

0.8 Create an ROC curve for your predictions

[6]:
```
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

```
test_fpr, test_tpr, test_thresholds = roc_curve(y_test, y_hat_test)

print('AUC: {}'.format(auc(test_fpr, test_tpr)))

# Seaborn's beautiful styling
sns.set_style('darkgrid', {'axes.facecolor': '0.9'})

plt.figure(figsize=(10, 8))
lw = 2

plt.plot(test_fpr, test_tpr, color='darkorange',
         lw=lw, label='Test ROC curve')

plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.yticks([i/20.0 for i in range(21)])
plt.xticks([i/20.0 for i in range(21)])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.show()
```

```
AUC: 0.8996515679442508
```

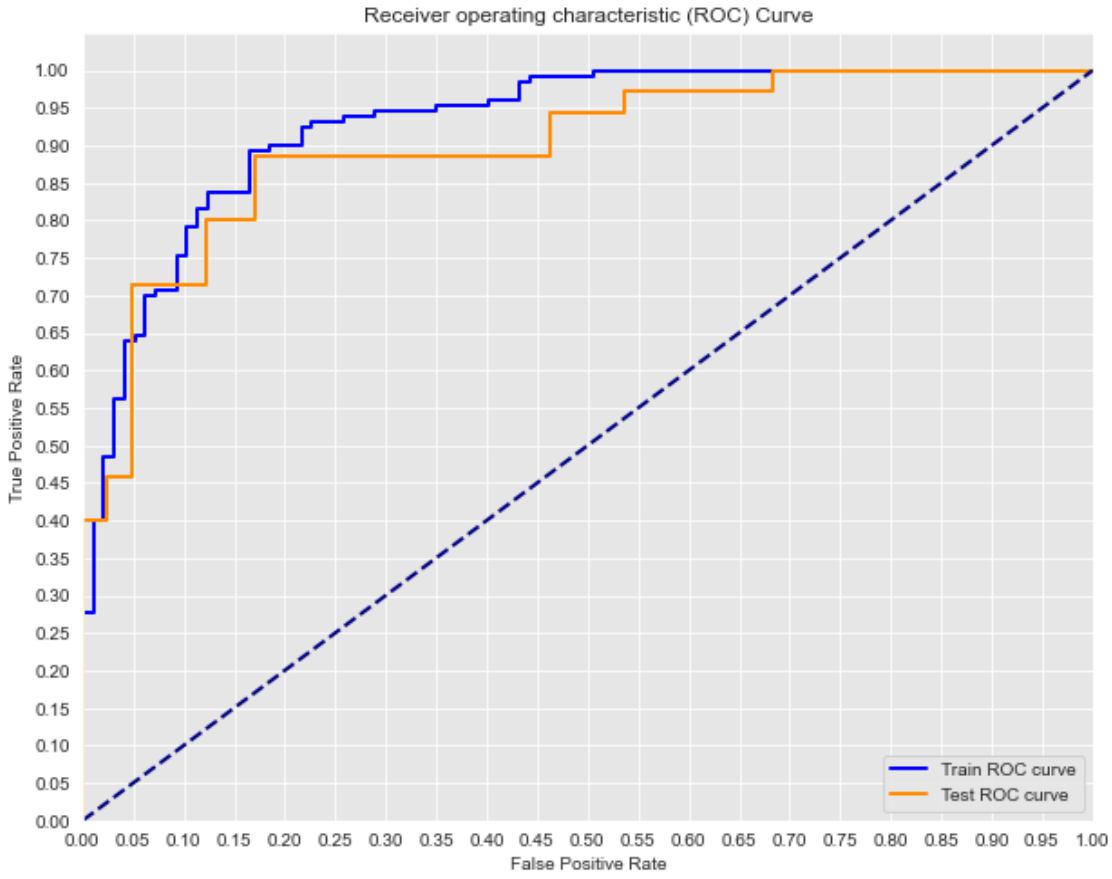


0.9 Update your ROC curve to include the training set

[8]:
```
y_hat_train = sigmoid(np.dot(X_train, weights))

train_fpr, train_tpr, train_thresholds = roc_curve(y_train, y_hat_train)

test_fpr, test_tpr, test_thresholds = roc_curve(y_test, y_hat_test)

# Train AUC
print('Train AUC: {}'.format( auc(train_fpr, train_tpr) ))
print('AUC: {}'.format(auc(test_fpr, test_tpr)))

# Seaborn's beautiful styling
sns.set_style('darkgrid', {'axes.facecolor': '0.9'})

plt.figure(figsize=(10, 8))
lw = 2

plt.plot(train_fpr, train_tpr, color='blue',
```

```
         lw=lw, label='Train ROC curve')
plt.plot(test_fpr, test_tpr, color='darkorange',
         lw=lw, label='Test ROC curve')

plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.yticks([i/20.0 for i in range(21)])
plt.xticks([i/20.0 for i in range(21)])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.show()
```

```
Train AUC: 0.9291038858049168
AUC: 0.8996515679442508
```



0.10 Create a confusion matrix for your predictions

Use a standard decision boundary of 0.5 to convert your probabilities output by logistic regression into binary classifications. (Again this should be for the test set.) Afterward, feel free to use the built-in scikit-learn function to compute the confusion matrix as we discussed in previous sections.

[ ]:
```
# Your code here
```

0.11 Initial Model - scikit-learn

Use scikit-learn to build a similar model. To start, create an identical model as you did in the last section; turn off the intercept and set the regularization parameter, `C`, to a ridiculously large number such as 1e16.

[11]:
```
# Your code here
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression(C = 1e16, fit_intercept=False,
                            solver="liblinear")
logreg.fit(X_train, y_train)
```

[11]:
```
LogisticRegression(C=1e+16, fit_intercept=False, solver='liblinear')
```

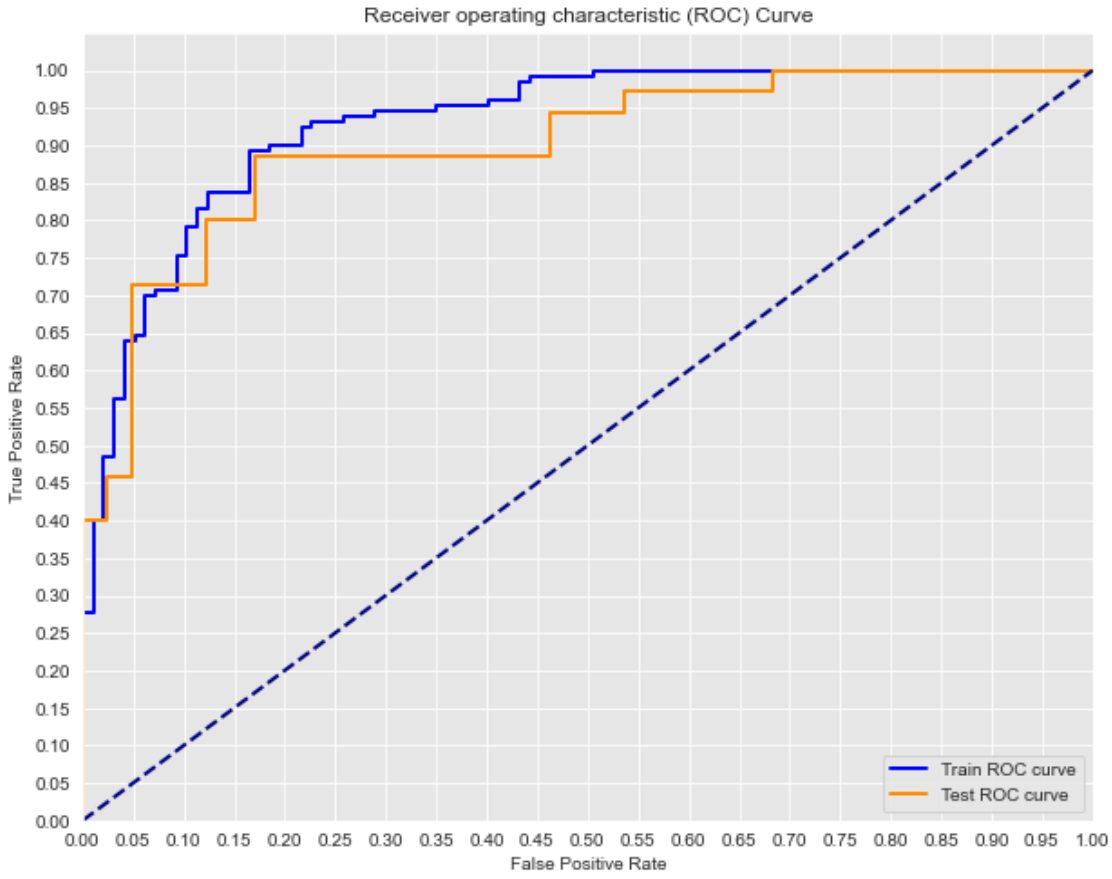
0.12 Create an ROC Curve for the scikit-learn model

Use both the training and test sets

[12]:
```
# Your code here

y_train_score = logreg.fit(X_train, y_train).decision_function(X_train)
y_test_score = logreg.fit(X_train, y_train).decision_function(X_test)

train_fpr, train_tpr, train_thresholds = roc_curve(y_train, y_train_score)
test_fpr, test_tpr, test_thresholds = roc_curve(y_test, y_test_score)


print('Train AUC: {}'.format(auc(train_fpr, train_tpr)))
print('Test AUC: {}'.format(auc(test_fpr, test_tpr)))

plt.figure(figsize=(10, 8))
lw = 2

plt.plot(train_fpr, train_tpr, color='blue',
         lw=lw, label='Train ROC curve')
plt.plot(test_fpr, test_tpr, color='darkorange',
         lw=lw, label='Test ROC curve')

plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
```

```
plt.yticks([i/20.0 for i in range(21)])
plt.xticks([i/20.0 for i in range(21)])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.show()
```

```
Train AUC: 0.9291038858049168
Test AUC: 0.8996515679442508
```



0.13 Add an Intercept

Now add an intercept to the scikit-learn model. Keep the regularization parameter `C` set to a very large number such as 1e16.

[24]:
```
# Create new model
logregi = LogisticRegression(C = 1e16, fit_intercept=True,
                             solver="liblinear")
logregi.fit(X_train, y_train)
```

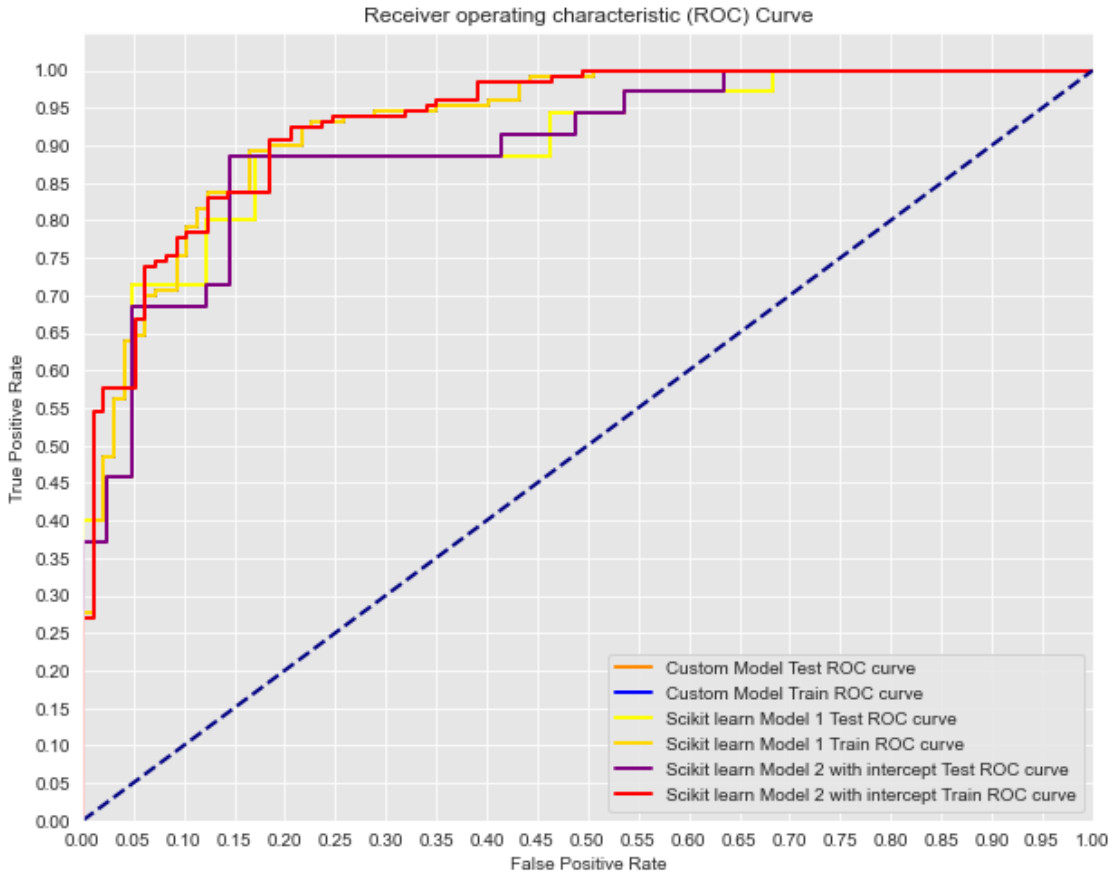
[24]:
```
LogisticRegression(C=1e+16, solver='liblinear')
```

Plot all three models ROC curves on the same graph.

[29]:
```
# Initial model plots
test_fpr, test_tpr, test_thresholds = roc_curve(y_test, y_hat_test)
train_fpr, train_tpr, train_thresholds = roc_curve(y_train, y_hat_train)


print('Custom Model Test AUC: {}'.format(auc(test_fpr, test_tpr)))
print('Custome Model Train AUC: {}'.format(auc(train_fpr, train_tpr)))

plt.figure(figsize=(10,8))
lw = 2

plt.plot(test_fpr, test_tpr, color='darkorange',
         lw=lw, label='Custom Model Test ROC curve')
plt.plot(train_fpr, train_tpr, color='blue',
         lw=lw, label='Custom Model Train ROC curve')


# Second model plots
y_test_score2 = logreg.decision_function(X_test)
y_train_score2 = logreg.decision_function(X_train)

test_fpr2, test_tpr2, test_thresholds2 = roc_curve(y_test, y_test_score2)
train_fpr2, train_tpr2, train_thresholds2 = roc_curve(y_train, y_train_score2)

print('Scikit-learn Model 1 Test AUC: {}'.format(auc(test_fpr2, test_tpr2)))
print('Scikit-learn Model 1 Train AUC: {}'.format(auc(train_fpr2, train_tpr2)))


plt.plot(test_fpr, test_tpr, color='yellow',
         lw=lw, label='Scikit learn Model 1 Test ROC curve')
plt.plot(train_fpr, train_tpr, color='gold',
         lw=lw, label='Scikit learn Model 1 Train ROC curve')


# Third model plots
y_test_score = logregi.decision_function(X_test)
y_train_score = logregi.decision_function(X_train)

test_fpr, test_tpr, test_thresholds = roc_curve(y_test, y_test_score)
train_fpr, train_tpr, train_thresholds = roc_curve(y_train, y_train_score)

print('Scikit-learn Model 2 with intercept Test AUC: {}'.format(auc(test_fpr,␣
 ↪test_tpr)))
```

```
print('Scikit-learn Model 2 with intercept Train AUC: {}'.format(auc(train_fpr,␣
 ↪train_tpr)))


plt.plot(test_fpr, test_tpr, color='purple',
         lw=lw, label='Scikit learn Model 2 with intercept Test ROC curve')
plt.plot(train_fpr, train_tpr, color='red',
         lw=lw, label='Scikit learn Model 2 with intercept Train ROC curve')

# Formatting
plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.yticks([i/20.0 for i in range(21)])
plt.xticks([i/20.0 for i in range(21)])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic (ROC) Curve')
plt.legend(loc="lower right")
plt.show()
```

```
Custom Model Test AUC: 0.8996515679442508
Custome Model Train AUC: 0.9291038858049168
Scikit-learn Model 1 Test AUC: 0.8996515679442508
Scikit-learn Model 1 Train AUC: 0.9291038858049168
Scikit-learn Model 2 with intercept Test AUC: 0.8989547038327527
Scikit-learn Model 2 with intercept Train AUC: 0.9325931800158604
```



0.14 Altering the Regularization Parameter

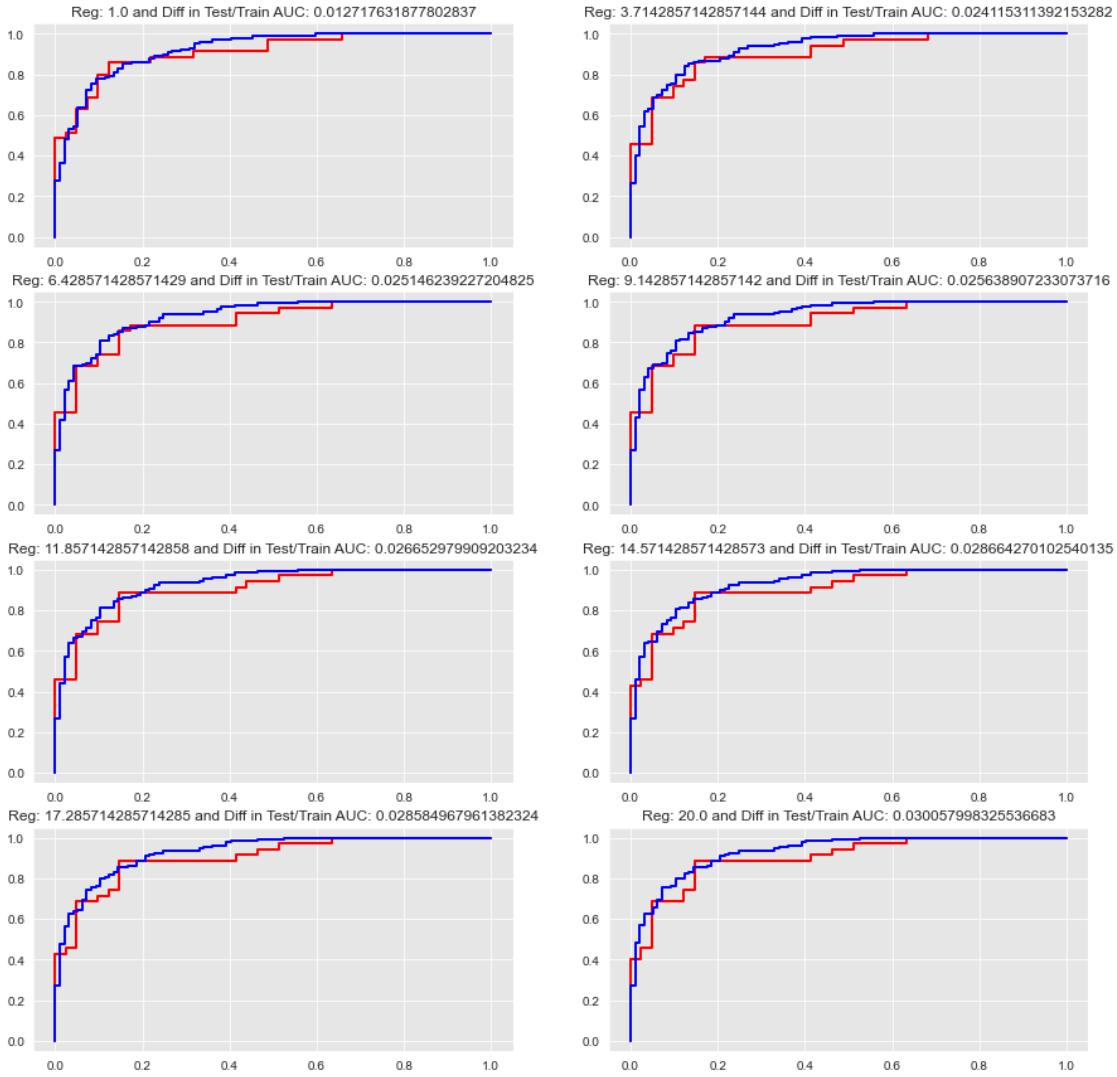
Now, experiment with altering the regularization parameter. At a minimum, create 5 different subplots with varying regularization (`C`) parameters. For each, plot the ROC curve of the training and test set for that specific model.

Regularization parameters between 1 and 20 are recommended. Observe the difference in test and training AUC as you go along.

[37]:
```
# Your code here
def reg(c):
    lw = 2
    logregic = LogisticRegression(C = c, fit_intercept=True,
                           solver="liblinear")
    logregic.fit(X_train, y_train)
    y_test_score_c = logregic.decision_function(X_test)
    y_train_score_c = logregic.decision_function(X_train)

    test_fpr_c , test_tpr_c, test_thresholds_c = roc_curve(y_test,␣
 ↪y_test_score_c)
```

```
    train_fpr_c, train_tpr_c, train_thresholds_c = roc_curve(y_train,␣
 ↪y_train_score_c)

    print('Scikit-learn Model 1 Test AUC: {}'.format(auc(test_fpr_c,␣
 ↪test_tpr_c)))
    print('Scikit-learn Model 1 Train AUC: {}'.format(auc(train_fpr_c,␣
 ↪train_tpr_c)))


    ax.plot(test_fpr_c, test_tpr_c, color='red',
         lw=lw, label='Scikit learn Model 1 Test ROC curve')
    ax.plot(train_fpr_c, train_tpr_c, color='blue',
         lw=lw, label='Scikit learn Model 1 Train ROC curve')
```

[84]:
```
nr = 4
nc = 2
fig, axes = plt.subplots(nrows = nr,ncols = nc, figsize=(15, 15))
n = nr*nc
C = list(np.linspace(1, 20, num = n))
len(C)
for i in range(len(C)):

    ax = axes[i//2][i%2]
    lw = 2
    logregic = LogisticRegression(C = C[i], fit_intercept=True,
                            solver="liblinear")
    logregic.fit(X_train, y_train)
    y_test_score_c = logregic.decision_function(X_test)
    y_train_score_c = logregic.decision_function(X_train)

    test_fpr_c , test_tpr_c, test_thresholds_c = roc_curve(y_test,␣
 ↪y_test_score_c)
    train_fpr_c, train_tpr_c, train_thresholds_c = roc_curve(y_train,␣
 ↪y_train_score_c)
    diff = auc(test_fpr_c, test_tpr_c) - auc(train_fpr_c, train_tpr_c)



    ax.plot(test_fpr_c, test_tpr_c, color='red',
         lw=lw, label='Scikit learn Model 1 Test ROC curve')
    ax.plot(train_fpr_c, train_tpr_c, color='blue',
         lw=lw, label='Scikit learn Model 1 Train ROC curve')

    ax.set_title(f"Reg: {C[i]} and Diff in Test/Train AUC: {np.abs(diff)}")
```



How did the regularization parameter impact the ROC curves plotted above?

0.15 Summary

In this lab, you reviewed many of the accuracy measures for classification algorithms and observed the impact of additional tuning models using intercepts and regularization.