Hu_Chun_HW6

March 5, 2020

[1]:
```
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, cross_val_score,␣
 ↪GridSearchCV
from sklearn.linear_model import LogisticRegression
import warnings
warnings.filterwarnings('ignore')
```

0.1 Conceptual Exercises

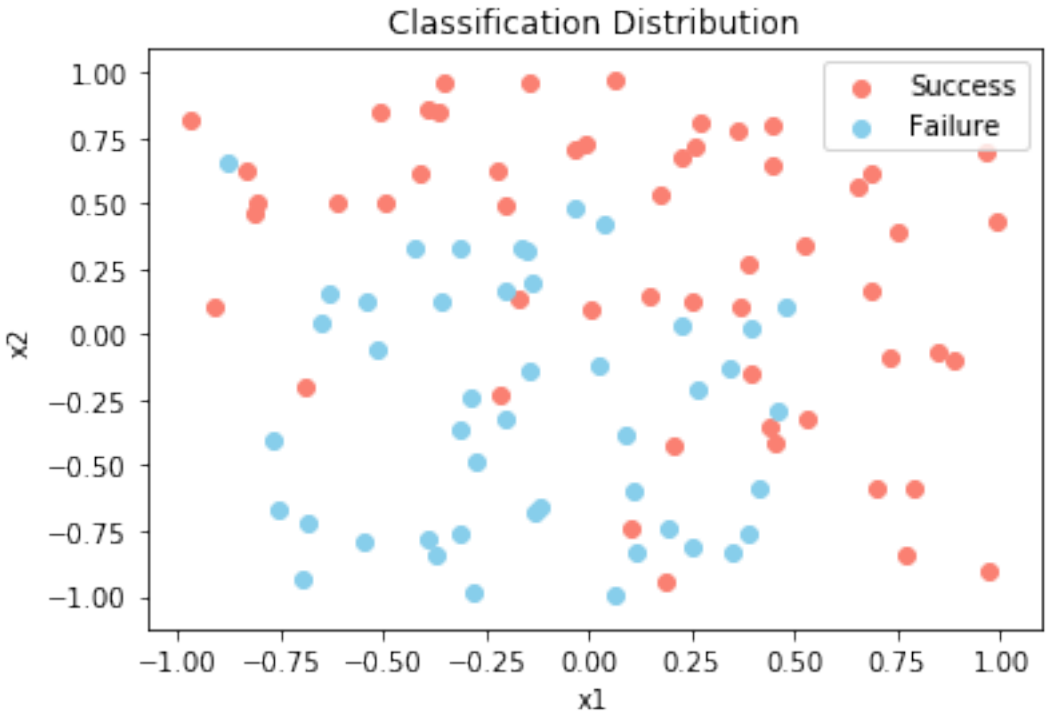
0.1.1 Non-linear separation

1.(15 points) Generate a simulated two-class data set with 100 observations and two features in which there is a visible (clear) but still non-linear separation between the two classes. Show that in this setting, a support vector machine with a radial kernel will outperform a support vector classifier (a linear kernel) on the training data. Which technique performs best on the test data? Make plots and report training and test error rates in order to support your conclusions.

[3]:
```
np.random.seed(123)

# generate two features
x1 = np.random.uniform(-1,1,100)
x2 = np.random.uniform(-1,1,100)

# generate error term
err = np.random.normal(0,0.5,100)

# calculate y
y = x1 + x1**2 + x2 + x2**2 + err

# define success/failure
norm_y = (y-np.min(y))/(np.max(y)-np.min(y))
success = norm_y > 0.4 # create a balanced distribution
failure = norm_y <= 0.4

# plot classification
```

```
plt.scatter(x1[success], x2[success], color='salmon')
plt.scatter(x1[failure], x2[failure], color='skyblue')
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend(['Success', 'Failure'],loc=1)
plt.title('Classification Distribution')
```

[3]:
```
Text(0.5, 1.0, 'Classification Distribution')
```



[4]:
```
X = pd.DataFrame({'x1': x1, 'x2': x2})
```

[5]:
```
X_train, X_test, y_train, y_test = train_test_split(X, success, test_size=0.2,␣
 ↪random_state=42)
```

[6]:
```
svm_radial = SVC(gamma='scale').fit(X_train, y_train)
```

[7]:
```
svm_linear = SVC(kernel='linear').fit(X_train, y_train)
```

[8]:
```
print("Training Error:")
print("SVM with a radial kernel has an error rate of: ", round(1-svm_radial.
 ↪score(X_train, y_train),4))
print("SVM with a linear kernel has an error rate of: ", round(1-svm_linear.
 ↪score(X_train, y_train),4))
```

```
Training Error:
SVM with a radial kernel has an error rate of:  0.175
SVM with a linear kernel has an error rate of:  0.275
```

[9]:
```
print("Test Error:")
print("SVM with a radial kernel has an error rate of: ", round(1-svm_radial.
 ↪score(X_test, y_test),4))
print("SVM with a linear kernel has an error rate of: ", round(1-svm_linear.
 ↪score(X_test, y_test),4))
```

```
Test Error:
SVM with a radial kernel has an error rate of:  0.15
SVM with a linear kernel has an error rate of:  0.2
```

SVM with radial kernel performs better than SVM with linear kernel on both training and test data according to their error rates.

0.1.2 SVM vs. logistic regression

We have seen that we can fit an SVM with a non-linear kernel in order to perform classification using a non-linear decision boundary. We will now see that we can also obtain a non-linear decision boundary by performing logistic regression using non-linear transformations of the features. Your goal here is to compare different approaches to estimating non-linear decision boundaries, and thus assess the benefits and drawbacks of each.

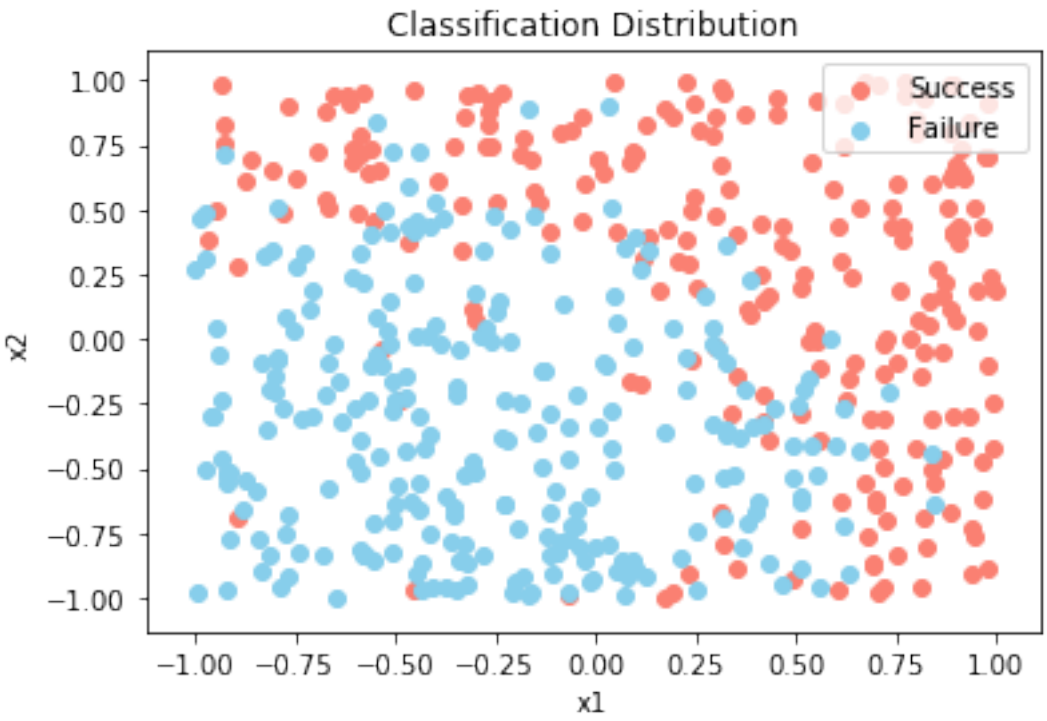
2.(5 points) Generate a data set with $n = 500$ and $p = 2$, such that the observations belong to two classes with some overlapping, non-linear boundary between them.

[10]:
```
# generate two features
x3 = np.random.uniform(-1,1,500)
x4 = np.random.uniform(-1,1,500)

# generate error term
err2 = np.random.normal(0,0.5,500)

# calculate y
y2 = x3 + x3**2 + x4 + x4**2 + err2

# define success/failure
norm_y2 = (y2-np.min(y2))/(np.max(y2)-np.min(y2))
success2 = norm_y2 > 0.4
failure2 = norm_y2 <= 0.4
```

3.(5 points) Plot the observations with colors according to their class labels (y). Your plot should display X_1 on the x-axis and X_2 on the y-axis.

[11]:
```
# plot classification
plt.scatter(x3[success2], x4[success2], color='salmon')
plt.scatter(x3[failure2], x4[failure2], color='skyblue')
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend(['Success', 'Failure'],loc=1)
plt.title('Classification Distribution')
```

[11]:
```
Text(0.5, 1.0, 'Classification Distribution')
```



4.(5 points) Fit a logistic regression model to the data, using X_1 and X_2 as predictors.

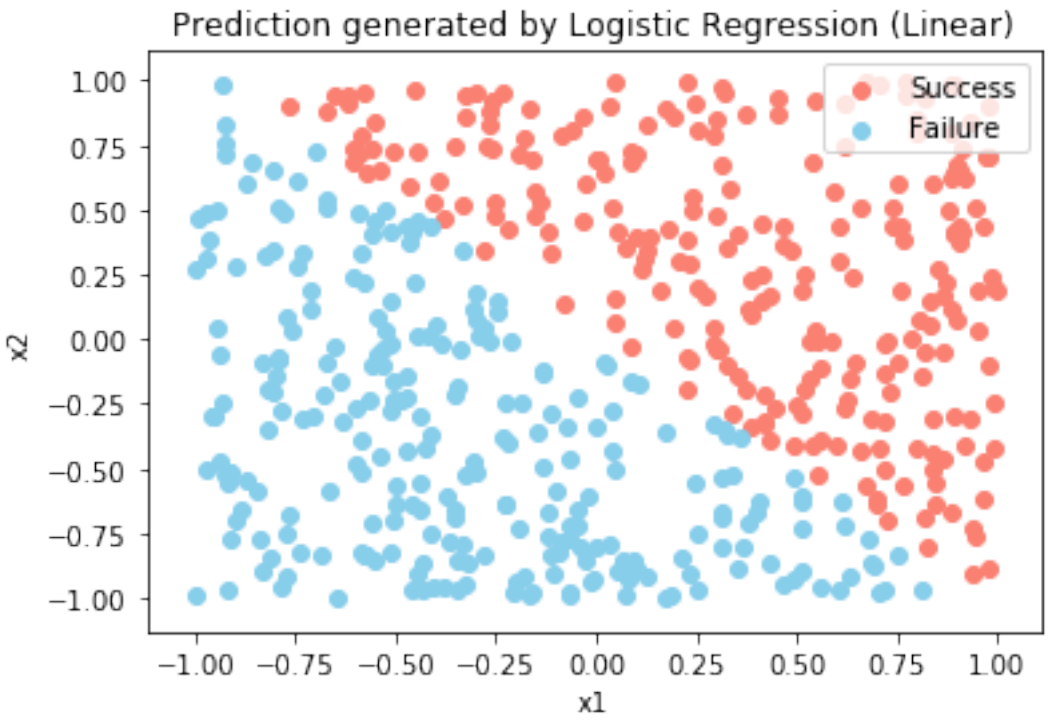
[12]:
```
X = pd.DataFrame({'x1': x3, 'x2': x4})
```

[13]:
```
lm_linear = LogisticRegression().fit(X, success2)
```

5.(5 points) Obtain a predicted class label for each observation based on the logistic model previously fit. Plot the observations, colored according to the predicted class labels (the predicted decision boundary should look linear).

[19]:
```
# obtain class label
lm_linear_pred = lm_linear.predict(X)

# plot classification
plt.scatter(x3[lm_linear_pred], x4[lm_linear_pred], color='salmon')
plt.scatter(x3[~lm_linear_pred], x4[~lm_linear_pred], color='skyblue')
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend(['Success', 'Failure'],loc=1)
plt.title('Prediction generated by Logistic Regression (Linear)')
```

[19]:
```
Text(0.5, 1.0, 'Prediction generated by Logistic Regression (Linear)')
```



(5 points) Now fit a logistic regression model to the data, but this time using some non-linear function of both X_1 and X_2 as predictors (e.g. $X_1^2, X_1 \times X_2, \log(X_2)$, and so on).

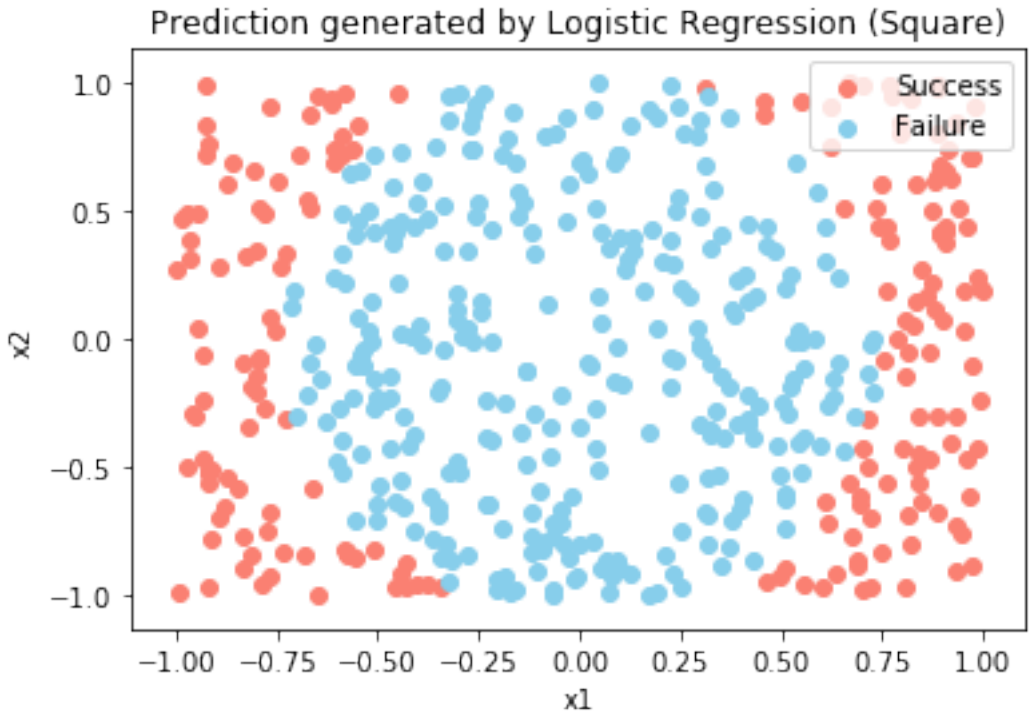
[15]:
```
X_square = pd.DataFrame({'x1': x3**2, 'x2': x4**2})
lm_square = LogisticRegression().fit(X_square, success2)
```

[16]:
```
X_interaction = pd.DataFrame({'x1': x3*x4})
lm_interaction = LogisticRegression().fit(X_interaction, success2)
```

6.(5 points) Now, obtain a predicted class label for each observation based on the fitted model with non-linear transformations of the X features in the previous question. Plot the observations, colored according to the new predicted class labels from the non-linear model (the decision boundary should now be obviously non-linear). If it is not, then repeat earlier steps until you come up with an example in which the predicted class labels and the resultant decision boundary are clearly non-linear.

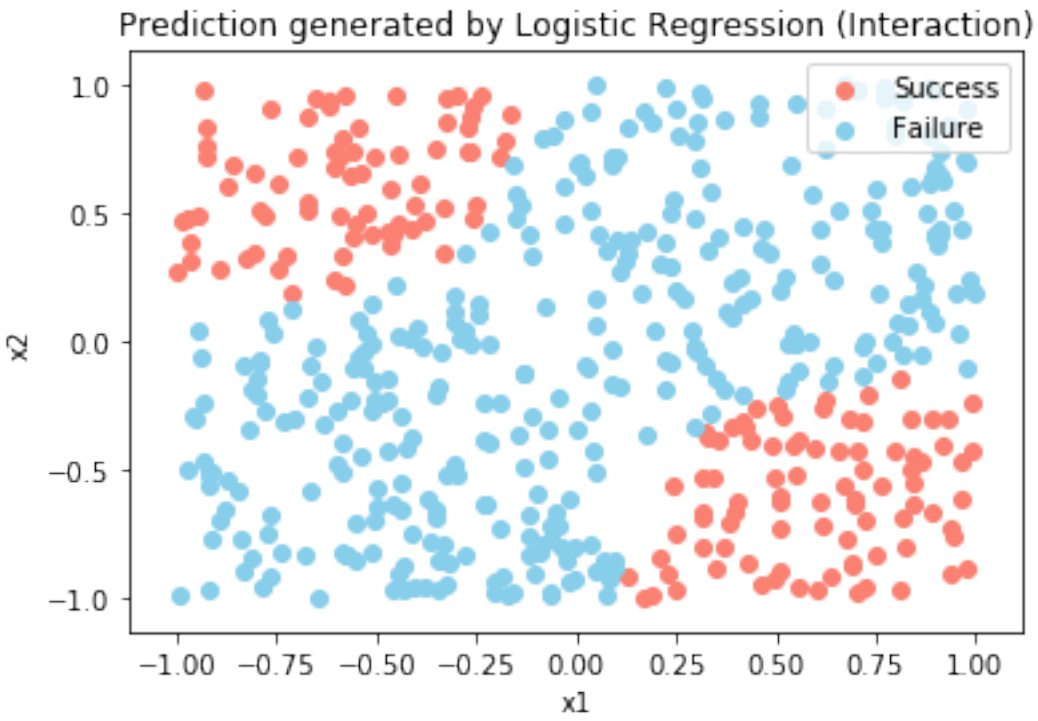
[20]:
```
# obtain class label (square)
lm_square_pred = lm_square.predict(X_square)

# plot classification (square)
plt.scatter(x3[lm_square_pred], x4[lm_square_pred], color='salmon')
plt.scatter(x3[~lm_square_pred], x4[~lm_square_pred], color='skyblue')
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend(['Success', 'Failure'],loc=1)
plt.title('Prediction generated by Logistic Regression (Square)')
```

[20]: `Text(0.5, 1.0, 'Prediction generated by Logistic Regression (Square)')`



[21]:
```
# obtain class label (interaction)
lm_interaction_pred = lm_interaction.predict(X_interaction)

# plot classification (interaction)
plt.scatter(x3[lm_interaction_pred], x4[lm_interaction_pred], color='salmon')
plt.scatter(x3[~lm_interaction_pred], x4[~lm_interaction_pred], color='skyblue')
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend(['Success', 'Failure'],loc=1)
plt.title('Prediction generated by Logistic Regression (Interaction)')
```

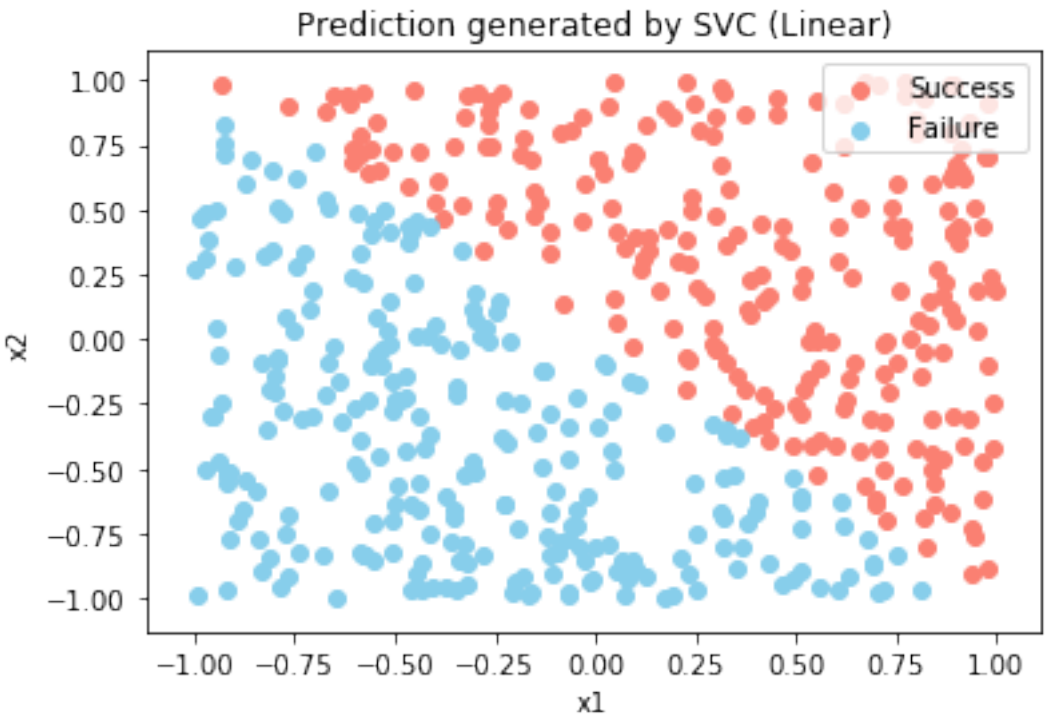
[21]: `Text(0.5, 1.0, 'Prediction generated by Logistic Regression (Interaction)')`



8.(5 points) Now, fit a support vector classifier (linear kernel) to the data with original X_1 and X_2 as predictors. Obtain a class prediction for each observation. Plot the observations, colored according to the predicted class labels.

[22]:
```
svm = SVC(kernel='linear').fit(X, success2)
svm_pred = svm.predict(X)

plt.scatter(x3[svm_pred], x4[svm_pred], color='salmon')
plt.scatter(x3[~svm_pred], x4[~svm_pred], color='skyblue')
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend(['Success', 'Failure'],loc=1)
plt.title('Prediction generated by SVC (Linear)')
```

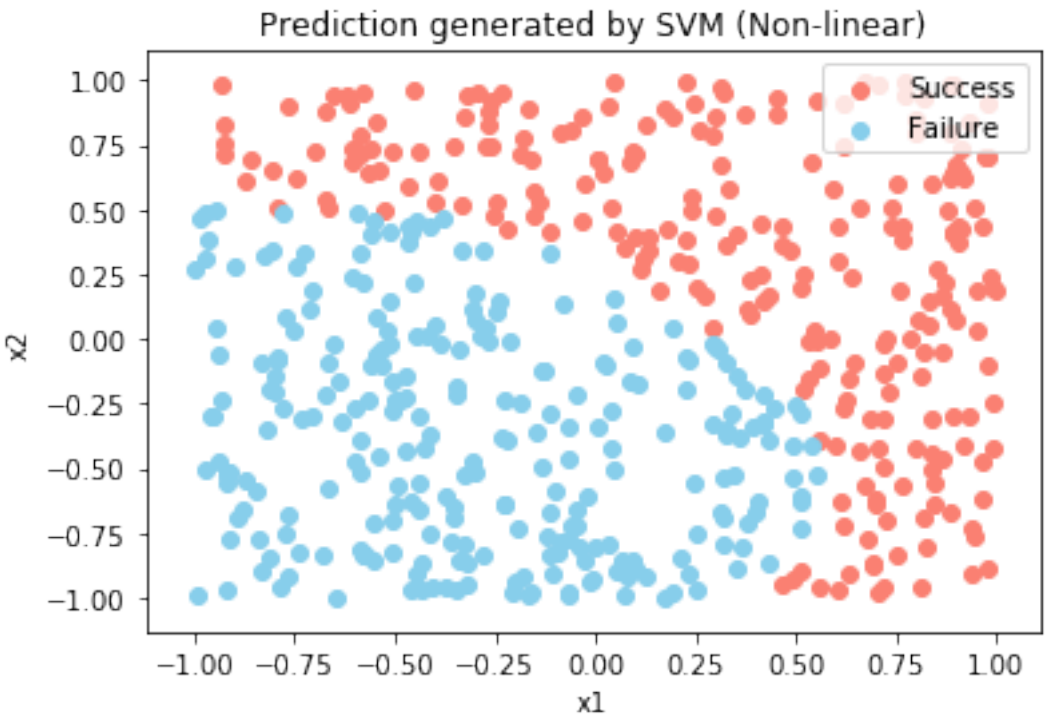
[22]:
```
Text(0.5, 1.0, 'Prediction generated by SVC (Linear)')
```



9.(5 points) Fit a SVM using a non-linear kernel to the data. Obtain a class prediction for each observation. Plot the observations, colored according to the predicted class labels.

[23]:
```
svm2 = SVC(gamma='scale').fit(X, success2)
svm_pred = svm2.predict(X)

plt.scatter(x3[svm_pred], x4[svm_pred], color='salmon')
plt.scatter(x3[~svm_pred], x4[~svm_pred], color='skyblue')
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend(['Success', 'Failure'],loc=1)
plt.title('Prediction generated by SVM (Non-linear)')
```

[23]:
```
Text(0.5, 1.0, 'Prediction generated by SVM (Non-linear)')
```



10.(5 points) Discuss your results and specifically the tradeoffs between estimating non-linear decision boundaries using these two different approaches.

[24]:
```
print("Accuracy Rate:")
print("Logistic Regression (linear): ", lm_linear.score(X, success2))
print("Logistic Regression (squared): ", lm_square.score(X_square, success2))
print("Logistic Regression (interaction): ", lm_interaction.
 ↪score(X_interaction, success2))
print("SVC (linear): ", svm.score(X, success2))
print("SVM (non-linear): ", svm2.score(X, success2))
```

```
Accuracy Rate:
Logistic Regression (linear):  0.814
Logistic Regression (squared):  0.64
Logistic Regression (interaction):  0.586
SVC (linear):  0.816
SVM (non-linear):  0.864
```

From the classification plots above, we can see that the prediction made by logistic regression models, with non-linear function of x1,x2 as predictors (square & interaction) did not accurately capture the shape of our data. This is also reflected in the accuracy score, in which logistic regression with these non-linear predictors have even lower accuracy score than linear models. On the other hand, both linear logistic regression and linear SVC produce similar accuracy scores. SVM with non-linear (radial) kernel performs the best among all the models. It receives the highest accuracy score on prediction our class labels, and its prediction has a similar shape to our original

data. In terms of the tradeoffs, we need to specify non-linear functions to train the non-linear logistic regression model, while SVM has built-in non-linear kernels for us to use directly. Both accuracy and computation wise, SVM is a better choice for estimating non-linear boundaries.

0.2 Tuning cost

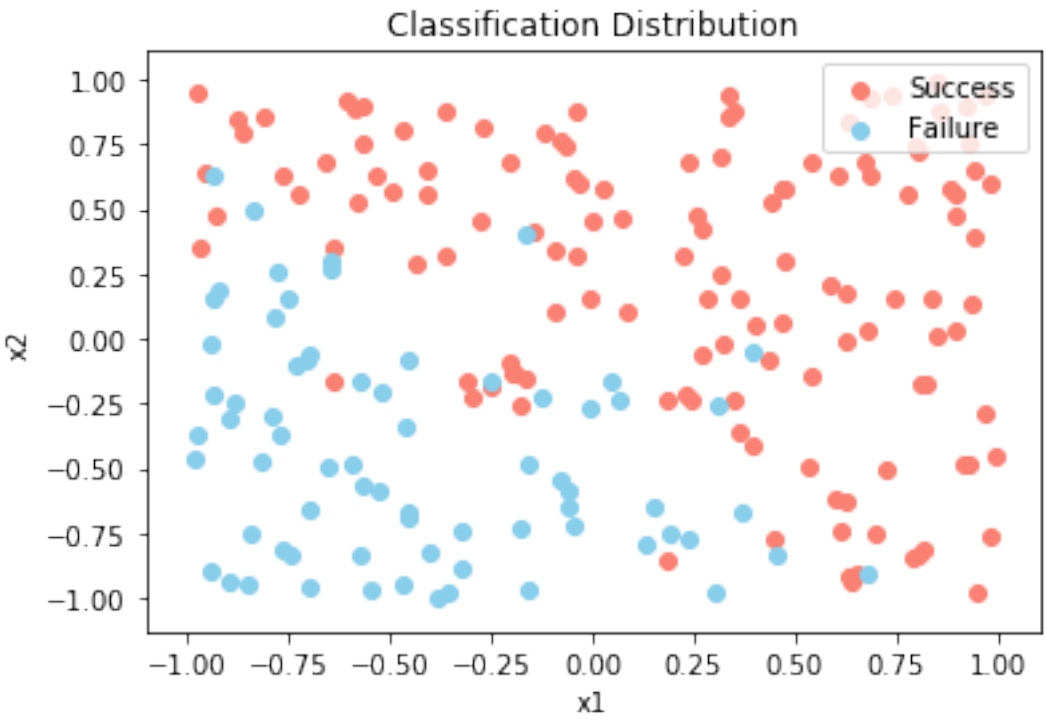
In class we learned that in the case of data that is just barely linearly separable, a support vector classifier with a small value of `cost` that misclassifies a couple of training observations may perform better on test data than one with a huge value of `cost` that does not misclassify any training observations. You will now investigate that claim.

11.(5 points) Generate two-class data with $p = 2$ in such a way that the classes are just barely linearly separable.

[86]:
```
np.random.seed(111)

# generate two features
x1 = np.random.uniform(-1,1,200)
x2 = np.random.uniform(-1,1,200)

# generate error term
err = np.random.normal(0,0.3,200) # smaller SD

# calculate y
y = x1 + x2 + err

# define success/failure
norm_y = (y-np.min(y))/(np.max(y)-np.min(y))
success = norm_y > 0.4
failure = norm_y <= 0.4
```

[87]:
```
# plot classification
plt.scatter(x1[success], x2[success], color='salmon')
plt.scatter(x1[failure], x2[failure], color='skyblue')
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend(['Success', 'Failure'],loc=1)
plt.title('Classification Distribution')
```

[87]:
```
Text(0.5, 1.0, 'Classification Distribution')
```



12.(5 points) Compute the cross-validation error rates for support vector classifiers with a range of `cost` values. How many training errors are made for each value of `cost` considered, and how does this relate to the cross-validation errors obtained?

[99]:
```
X = pd.DataFrame({'x1': x1, 'x2': x2})
X_train, X_test, y_train, y_test = train_test_split(X, success, test_size=0.2,␣
↪random_state=42)
```

[109]:
```
costs = [0.1, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4]
```

[110]:
```
cv_error = []
training_error = []
for c in costs:
    model = SVC(C=c, kernel='linear', random_state=42)
    cv = 1 - np.mean(cross_val_score(model, X_train, y_train, cv=10,␣
↪scoring='accuracy'))
    cv_error.append(cv)
    train = 1 - model.fit(X_train, y_train).score(X_train, y_train)
    training_error.append(train)

error_rates = pd.DataFrame({'Cost': costs, 'CV Error': cv_error, 'Training␣
↪Error': training_error})
```

[111]:
```
error_rates
```

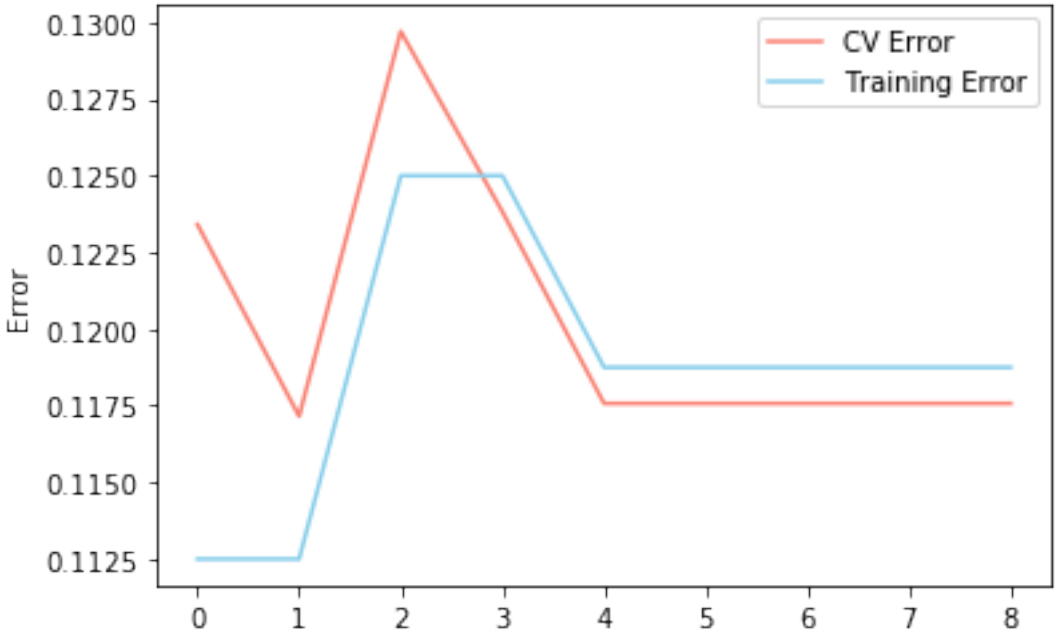
[111]:

	Cost	CV Error	Training Error
0	0.1	0.123407	0.11250

```
1   0.5  0.117157         0.11250
2   1.0  0.129706         0.12500
3   1.5  0.123824         0.12500
4   2.0  0.117574         0.11875
5   2.5  0.117574         0.11875
6   3.0  0.117574         0.11875
7   3.5  0.117574         0.11875
8   4.0  0.117574         0.11875
```

[112]:
```
plt.plot(cv_error, color='salmon')
plt.plot(training_error, color='skyblue')
plt.ylabel('Error')
plt.legend(['CV Error', 'Training Error'],loc=1)
```

[112]:
```
<matplotlib.legend.Legend at 0x12912a3c8>
```



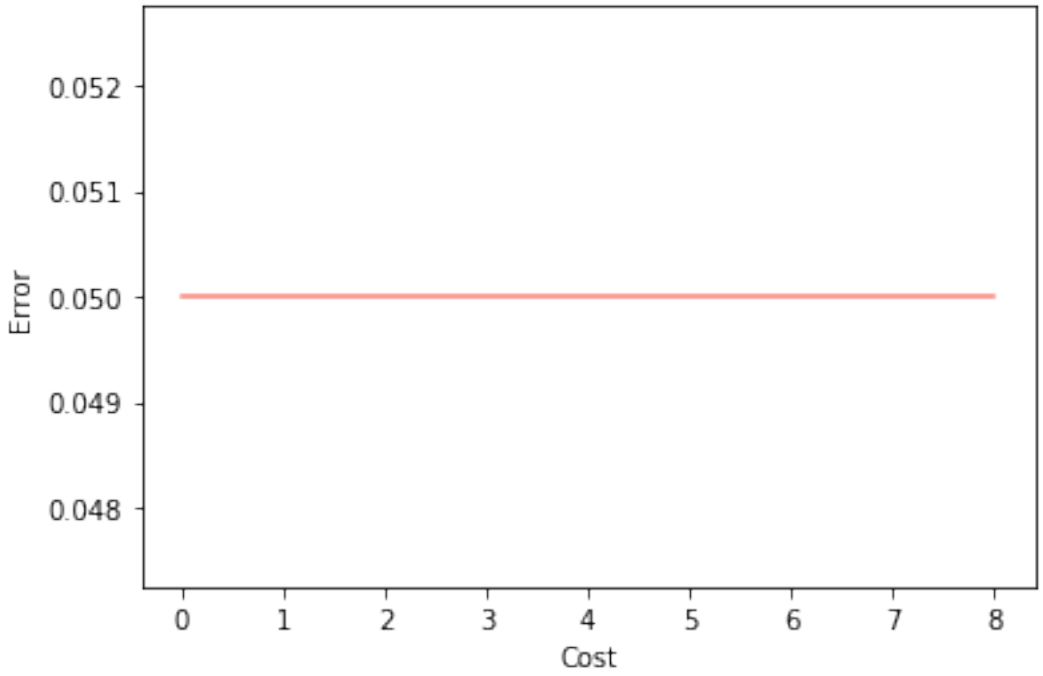
As cost value increases, the training error first reaches a maximum at cost value=1, and then starts to decrease. It soon reaches a plateau when cost value=2. Meanwhile, the CV error is highly correlated to the training error and follows a similar pattern. It first reaches the maximum point at cost value=1, and then decreases to a plateau. Both errors reach the maximum at cost value=1. The training error reaches the minimum rate at cost value=0.01, and the CV error reaches the minimum rate at cost value=0.5.

13.(5 points) Generate an appropriate test data set, and compute the test errors corresponding to each of the values of cost considered. Which value of cost leads to the fewest test errors, and how does this compare to the values of cost that yield the fewest training errors and the fewest cross-validation errors?

[113]:
```
test_error = []
for c in costs:
    model = SVC(C=c, kernel='linear', random_state=42)
    test = 1 - model.fit(X_train, y_train).score(X_test, y_test)
    test_error.append(test)
```

[116]:
```
plt.plot(test_error, color='salmon')
plt.xlabel('Cost')
plt.ylabel('Error')
```

[116]:
```
Text(0, 0.5, 'Error')
```



The test error is consistent across all cost values, suggesting that it reaches the minimum point at cost value=0.01. This is consistent with the training error, which also reaches the minimum value at cost value=0.01.

14.(5 points) Discuss your results.

Our results suggest that in the case of data that is barely linearly separable, small cost value at 0.01 is sufficient to bring the lowest training and test errors. Since the strength of the regularization is inversely proportional to cost value, a small cost value is able to regularize this type of data pretty well. Therefore, in this case, we agree to the argument that a support vector classifier with a small value of cost that misclassifies a couple of training observations performs better than one with a huge value of cost that does not misclassify any training observations.

0.3 Application: Predicting attitudes towards racist college professors

This week, building on last week's problem set, you will approach this classification problem from an SVM-based framework.

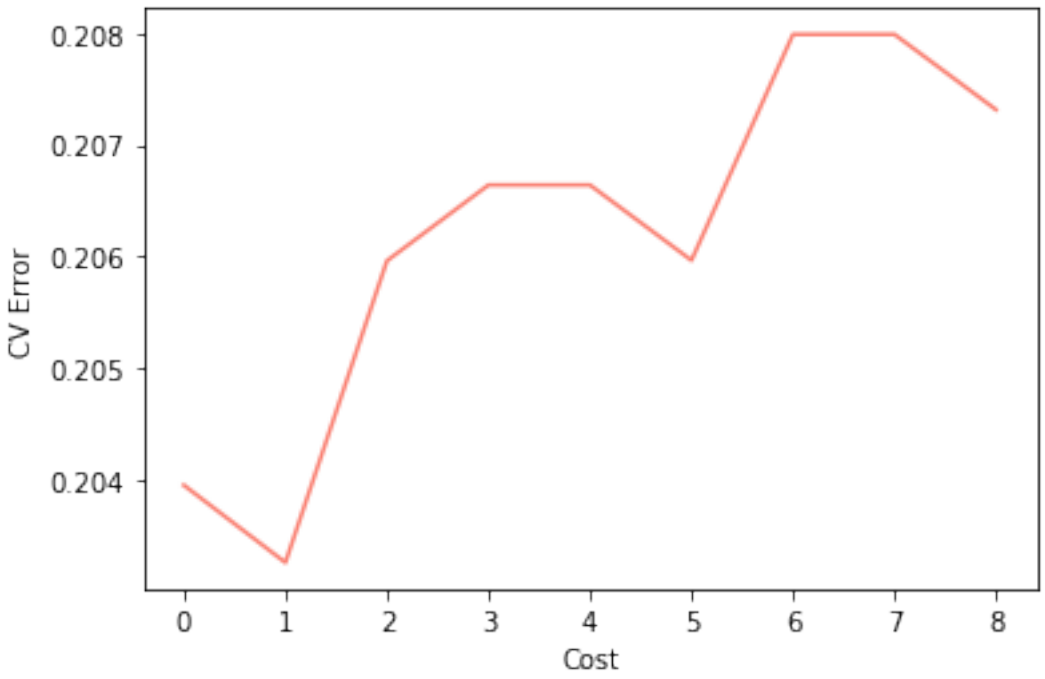
15.(5 points) Fit a support vector classifier to predict `colrac` as a function of all available predictors, using 10-fold cross-validation to find an optimal value for cost. Report the CV errors associated with different values of cost, and discuss your results.

[117]:
```
gss_train = pd.read_csv("gss_train.csv")
gss_test = pd.read_csv("gss_test.csv")
X_train = gss_train.drop('colrac',axis=1)
y_train = gss_train['colrac']
```

[118]:
```
cv_error = []
for c in costs:
    model = SVC(C=c, kernel='linear', random_state=42)
    cv = 1 - np.mean(cross_val_score(model, X_train, y_train, cv=10,␣
 ↪scoring='accuracy'))
    cv_error.append(cv)
```

[119]:
```
plt.plot(cv_error, color='salmon')
plt.xlabel('Cost')
plt.ylabel('CV Error')
```

[119]:
```
Text(0, 0.5, 'CV Error')
```



In general, the CV errors after 10-fold cross validation are pretty consistent across all cost values. The lowest error rate is reached at cost value=0.5. Although the graph suggests a large flunctuation of the values, the range of y axis is only from 0.203 to 0.208. This is consistent with what we observed in the case of data that is just barely linearly separable.

16.(15 points) Repeat the previous question, but this time using SVMs with radial and polynomial basis kernels, with different values for `gamma` and `degree` and `cost`. Present and discuss your

results (e.g., fit, compare kernels, cost, substantive conclusions across fits, etc.).

[120]:
```
models = {
    SVC(kernel='rbf'): {'C': [0.1, 0.5, 1, 2], 'degree': [2, 3, 4, 5],'gamma':␣
 ↪['scale', 'auto']},
    SVC(kernel='poly'): {'C': [0.1, 0.5, 1, 2], 'degree': [2, 3, 4, 5],'gamma':␣
 ↪['scale', 'auto']}
}
```

[121]:
```
for model, param in models.items():
    gscv = GridSearchCV(model, param, cv=10, refit=True)
    gscv.fit(X_train, y_train)
    print(gscv.best_estimator_)
    print(gscv.best_score_)
    print('-----------------')
```

```
SVC(C=2, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=2, gamma='scale', kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
0.761647535449021
-----------------
SVC(C=0.1, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=2, gamma='auto', kernel='poly',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
0.7913571910871033
-----------------
```

The GridSearchCV helps us find the best parameters for the model. For model using SVM with radial basis kernel, the best parameters turn out to be C=2, degree=2, and gamma='scale'. The mean accuracy score is 0.7616. For model using SVM with polynomial basis kernel, the best parameters are C=0.1, degree=2, and gamma='auto'. The mean accuracy score is 0.7914. The polynimoial kernel performs better than the radial kernel, but all three models (including linear SVC) perform pretty well on our dataset. Computation wise, linear SVC is the best model because it is the easiet to tune and the fastest to train.