Rim_Nak Won_HW06

March 8, 2020

Problem Set 6

MACS 30100 Winter 2020

Nak Won Rim

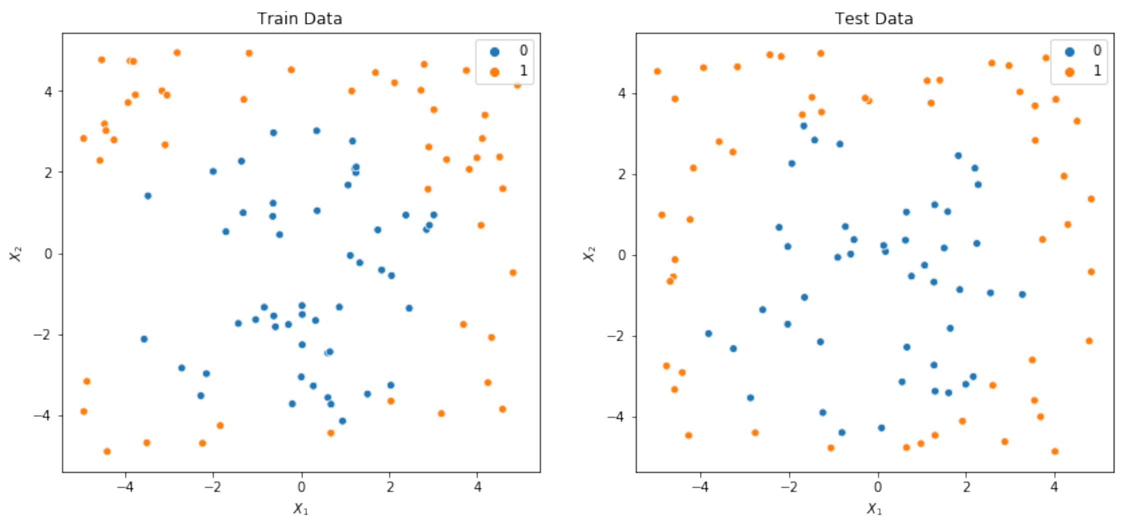
[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score, GridSearchCV
```

1. *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.

[2]:
```
# Part 1: Generate two-class data set
np.random.seed(1234)
x_1 = np.random.uniform(-5, 5, 100)
x_2 = np.random.uniform(-5, 5, 100)
X_train = np.column_stack([x_1, x_2])
y_train = np.int64((x_1 + x_1 **2 + x_2 + x_2 **2) > 15)

# generating the visualization to show that it has non-linear separation
fig, ax = plt.subplots(ncols=2, figsize = (14,6))
sns.scatterplot(x=X_train[:, 0], y=X_train[:, 1], hue=y_train,ax=ax[0])
ax[0].legend(loc=1)
ax[0].set_xlabel('$X_1$')
ax[0].set_ylabel('$X_2$')
ax[0].set_title('Train Data')
x_1_ = np.random.uniform(-5, 5, 100)
x_2_ = np.random.uniform(-5, 5, 100)
X_test = np.column_stack([x_1_, x_2_])
y_test = np.int64((x_1_ + x_1_ **2 + x_2_ + x_2_ **2) > 15)
sns.scatterplot(x=X_test[:, 0], y=X_test[:, 1], hue=y_test, ax=ax[1])
```

```
ax[1].legend(loc=1)
ax[1].set_xlabel('$X_1$')
ax[1].set_ylabel('$X_2$')
ax[1].set_title('Test Data');
```



As we can see clearly from the above plots, the separation between the two classes is non-linear.

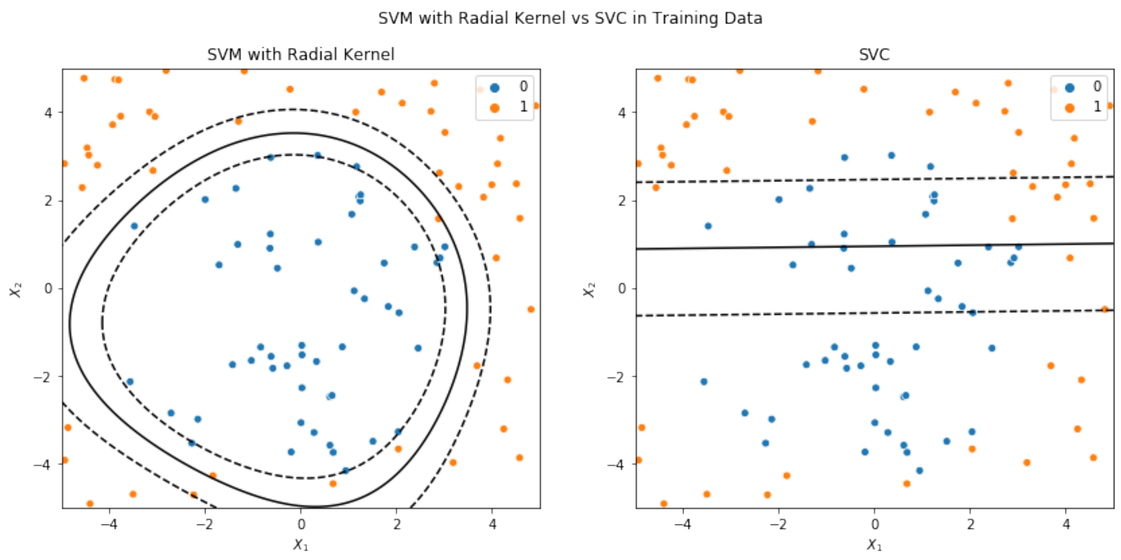
[3]:
```
# support vector machine with a radial kernel training
svm_rk = SVC(kernel='rbf').fit(X_train,y_train)

# plotting svm with train data
fig, ax = plt.subplots(ncols=2, figsize = (14,6))
sns.scatterplot(x=X_train[:, 0], y=X_train[:, 1], hue=y_train, ax=ax[0])
XX, YY = np.mgrid[-5:5:200j, -5:5:200j]
Z = svm_rk.decision_function(np.c_[XX.ravel(), YY.ravel()])
Z = Z.reshape(XX.shape)
ax[0].contour(XX, YY, Z, colors=['k', 'k', 'k'], linestyles=['--', '-', '--'],
             levels=[-.5, 0, .5])
ax[0].legend(loc=1)
ax[0].set_xlabel('$X_1$')
ax[0].set_ylabel('$X_2$')
ax[0].set_title('SVM with Radial Kernel')

# support vector classifier (a linear kernel) training
svm_lr = SVC(kernel='linear').fit(X_train,y_train)

# plotting svc with train data
sns.scatterplot(x=X_train[:, 0], y=X_train[:, 1], hue=y_train, ax=ax[1])
XX, YY = np.mgrid[-5:5:200j, -5:5:200j]
Z = svm_lr.decision_function(np.c_[XX.ravel(), YY.ravel()])
```

```
Z = Z.reshape(XX.shape)
ax[1].contour(XX, YY, Z, colors=['k', 'k', 'k'], linestyles=['--', '-', '--'],
              levels=[-.5, 0, .5])
ax[1].legend(loc=1)
ax[1].set_xlabel('$X_1$')
ax[1].set_ylabel('$X_2$')
ax[1].set_title('SVC')
fig.suptitle('SVM with Radial Kernel vs SVC in Training Data');
```



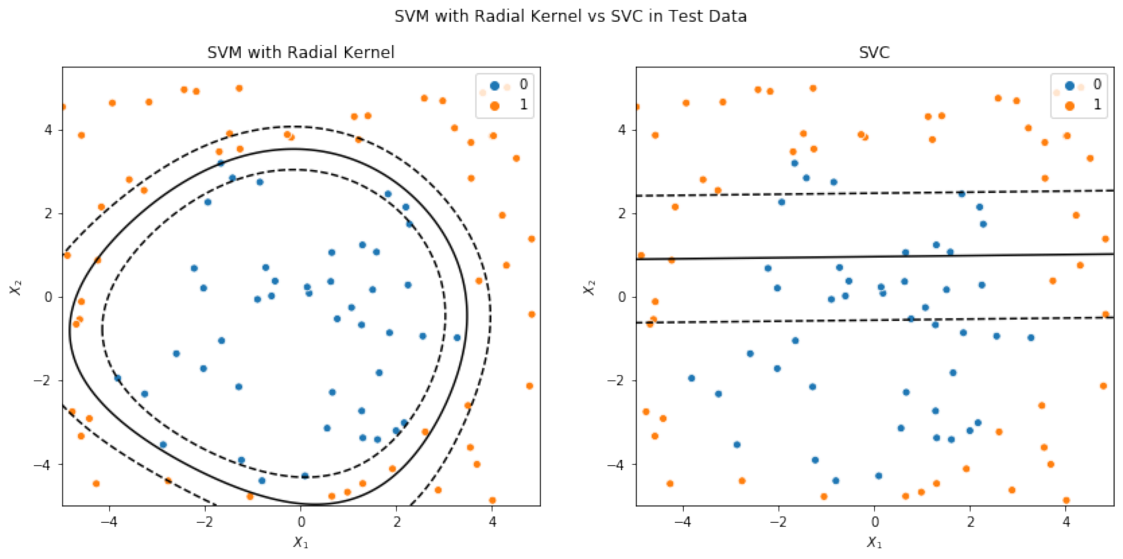
[4]:
```
print("Training error rate for SVM with Radial Kernel:", 1 - svm_rk.
 ↪score(X_train, y_train))
print("Training error rate for SVC:", 1 - svm_lr.score(X_train, y_train))
```

```
Training error rate for SVM with Radial Kernel: 0.040000000000000036
Training error rate for SVC: 0.28
```

The plot above clearly shows that a support vector machine with a radial kernel outperforms the support vector classifier in the training set. The training error rate further corroborates this finding since the training error rate for SVM with a radial kernel is much smaller than the training error rate of SVC. This is not surprising because a linear kernel will be unable to fit the non-linear separation properly.

[5]:
```
# plotting svm with test data
fig, ax = plt.subplots(ncols=2, figsize = (14,6))
sns.scatterplot(x=X_test[:, 0], y=X_test[:, 1], hue=y_test, ax=ax[0])
XX, YY = np.mgrid[-5:5:200j, -5:5:200j]
Z = svm_rk.decision_function(np.c_[XX.ravel(), YY.ravel()])
Z = Z.reshape(XX.shape)
ax[0].contour(XX, YY, Z, colors=['k', 'k', 'k'], linestyles=['--', '-', '--'],
                levels=[-.5, 0, .5])
ax[0].legend(loc=1)
ax[0].set_xlabel('$X_1$')
ax[0].set_ylabel('$X_2$')
ax[0].set_title('SVM with Radial Kernel')

# plotting svm hyperplane with test data
sns.scatterplot(x=X_test[:, 0], y=X_test[:, 1], hue=y_test, ax=ax[1])
XX, YY = np.mgrid[-5:5:200j, -5:5:200j]
Z = svm_lr.decision_function(np.c_[XX.ravel(), YY.ravel()])
Z = Z.reshape(XX.shape)
ax[1].contour(XX, YY, Z, colors=['k', 'k', 'k'], linestyles=['--', '-', '--'],
                levels=[-.5, 0, .5])
ax[1].legend(loc=1)
ax[1].set_xlabel('$X_1$')
ax[1].set_ylabel('$X_2$')
ax[1].set_title('SVC')
fig.suptitle('SVM with Radial Kernel vs SVC in Test Data');
```



[6]:
```
print("Test error rate for SVM with Radial Kernel:", 1 - svm_rk.score(X_test,␣
↪y_test))
print("Test error rate for SVC:", 1 - svm_lr.score(X_test, y_test))
```

```
Test error rate for SVM with Radial Kernel: 0.09999999999999998
Test error rate for SVC: 0.33999999999999997
```

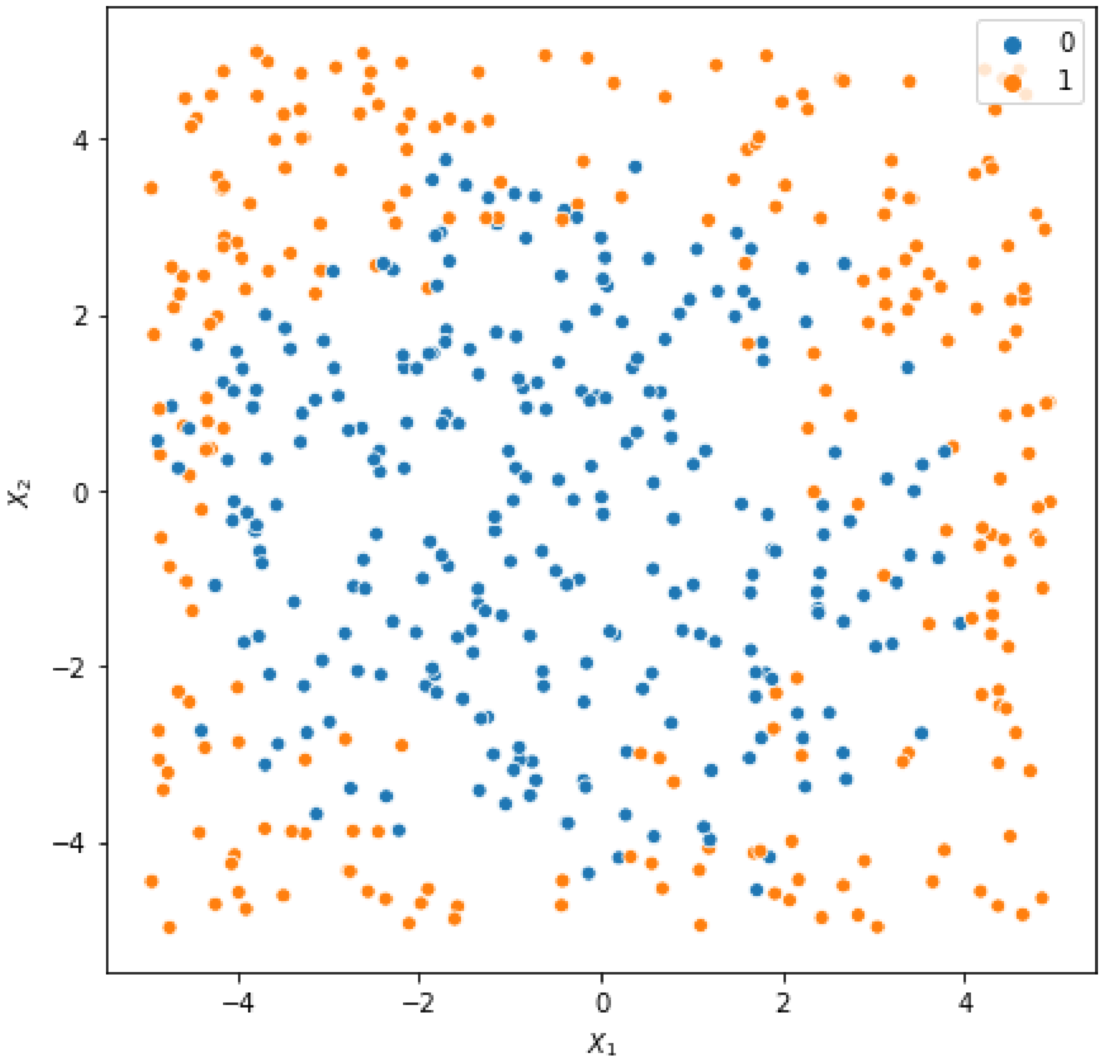
Analogous to the results from the training data set, the plot above clearly shows that a support vector machine with radial kernel outperforms the support vector classifier in the test set. Furthermore, we can see that the test error rate for SVM with a radial kernel is much smaller than the test error rate of SVC. The error rate slightly increases for the test set, which is natural, but the pattern persists. Again, this is an expected result because a linear kernel will be unable to fit the non-linear separation properly.

2. 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.

[7]:
```
np.random.seed(4321)
x2_1 = np.random.uniform(-5, 5, 500)
x2_2 = np.random.uniform(-5, 5, 500)
epsilon = np.random.uniform(-10, 10, 500)
X = np.column_stack([x2_1, x2_2])
y = np.int64((x2_1 + x2_1 **2 + x2_2 + x2_2 **2 + epsilon) > 15)
```

3. 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.

[8]:
```
plt.figure(figsize=(7,7))
sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=y)
plt.legend(loc=1)
plt.xlabel('$X_1$')
plt.ylabel('$X_2$');
```



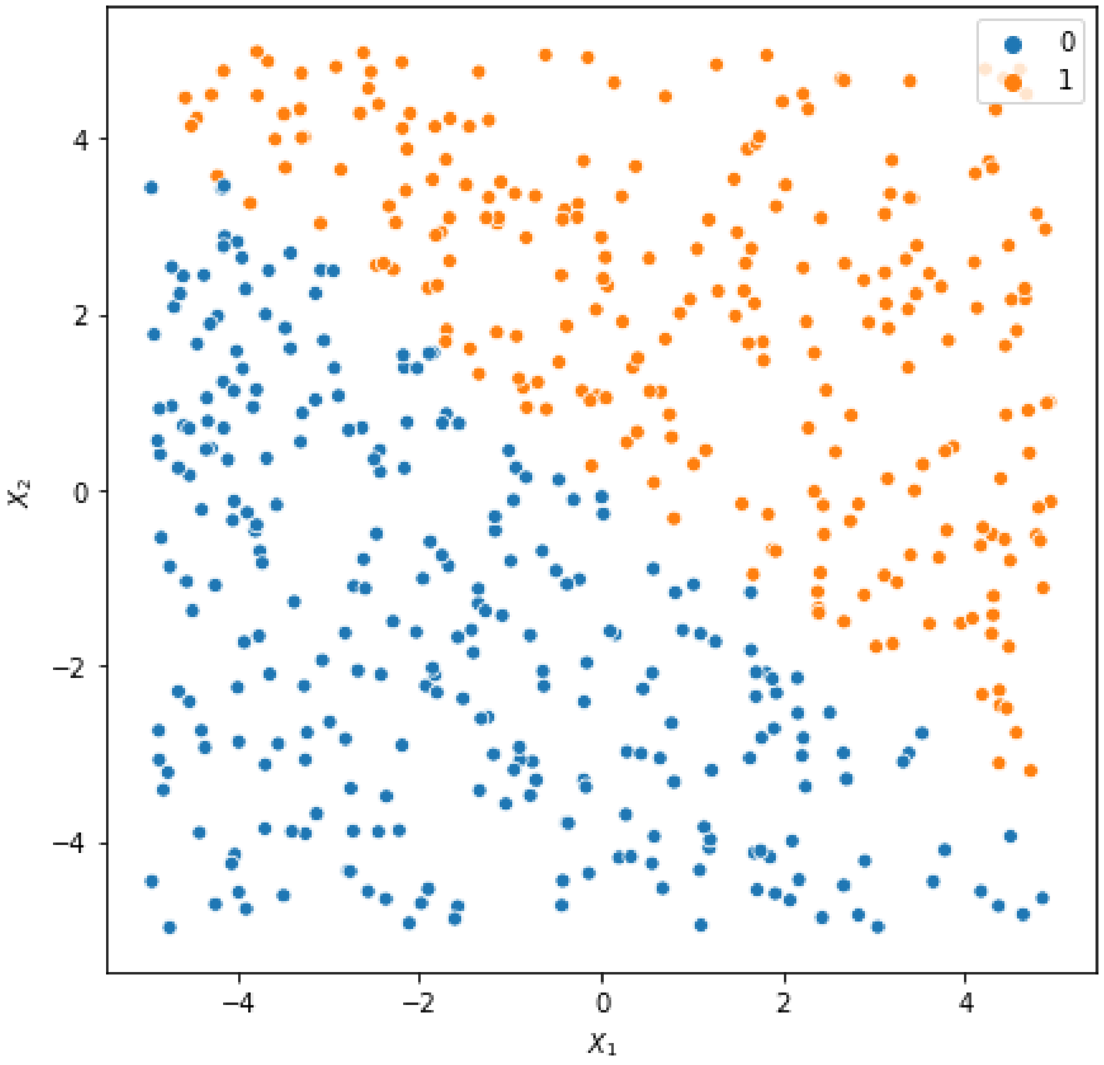
As we can see clearly from the above plots, the separation between the two classes is non-linear and there is some overlap in classes in the boundary.

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

[9]:
```
lr = LogisticRegression().fit(X,y)
```

5. 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).

[10]:
```
lr_pred = lr.predict(X)
plt.figure(figsize=(7,7))
sns.scatterplot(x=X[:,0], y=X[:,1], hue=lr_pred)
plt.legend(loc=1)
plt.xlabel('$X_1$')
plt.ylabel('$X_2$');
```



As we can see clearly from the above plots, the predicted decision boundary is linear.

6. 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).

I will fit two logistic regression model using two non-linear function (X_1^2, X_2^2 and $X_1 \times X_2$). Note that I did not fit $\log(X_1), \log(X_2)$ since some data in my simulated result were very close to zero and `numpy.log` threw errors because of it.

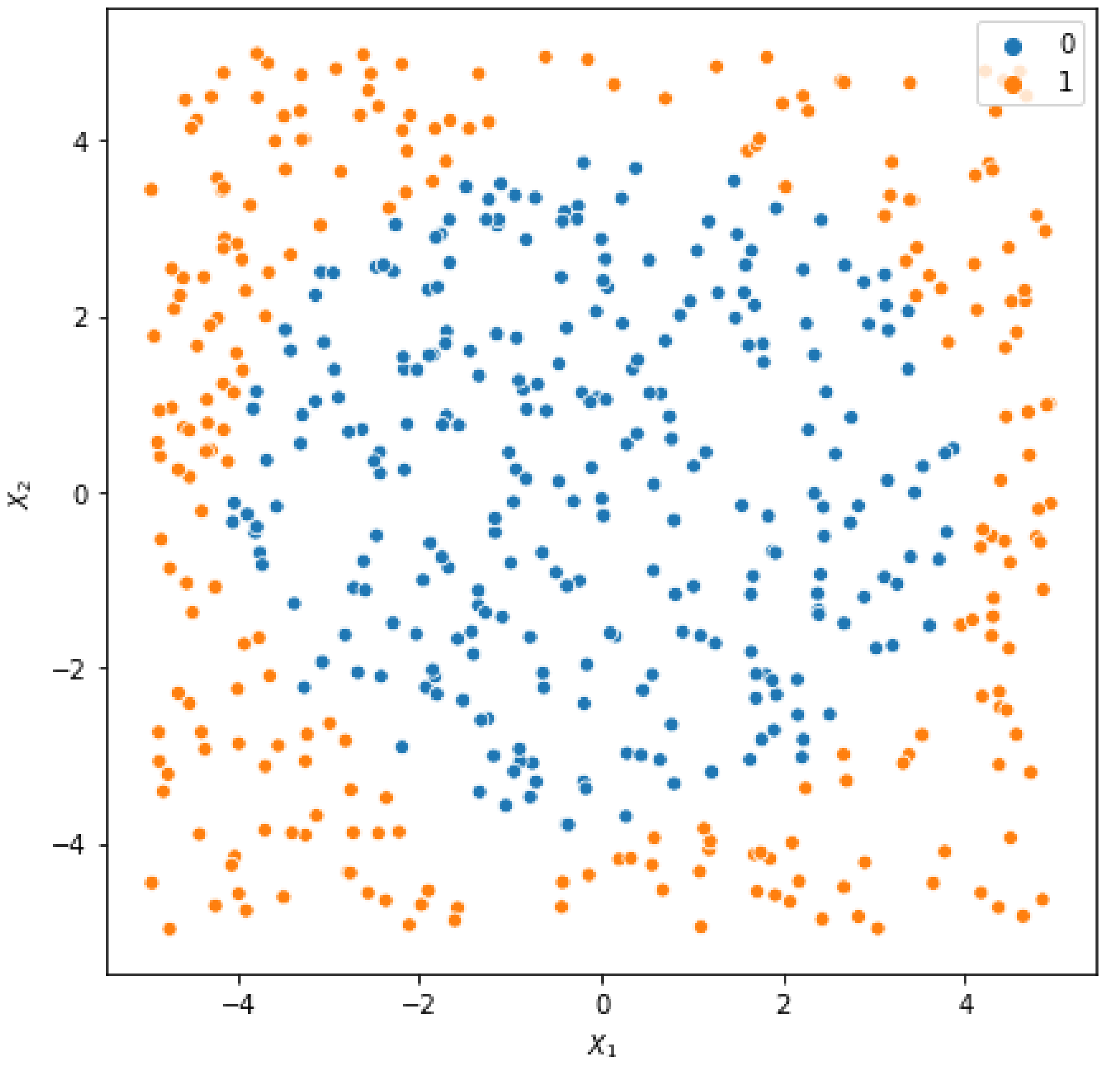
[11]:
```
lr_sq = LogisticRegression().fit(X ** 2, y)
lr_intact = LogisticRegression().fit(np.reshape(X[:,0] * X[:,1], (500,1)), y)
```

7. 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.

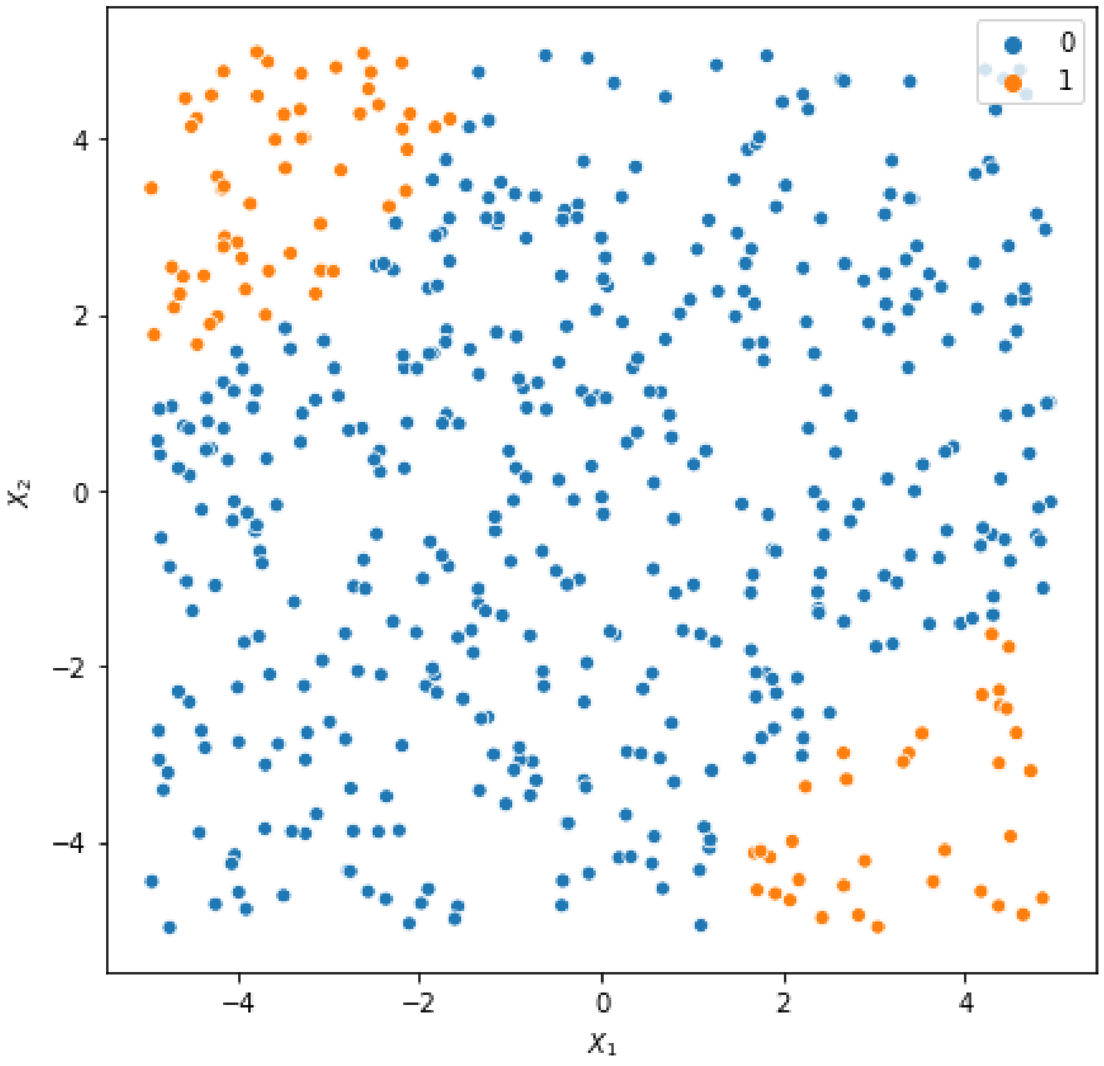
[12]:
```
# obtaining predictions
lr_sq_pred = lr_sq.predict(X ** 2)

# plotting
plt.figure(figsize=(7,7))
sns.scatterplot(x=X[:,0], y=X[:,1], hue=lr_sq_pred)
plt.legend(loc=1)
plt.xlabel('$X_1$')
plt.ylabel('$X_2$');
```



[13]:
```
# obtaining predictions
lr_intact_pred = lr_intact.predict(np.reshape(X[:,0] * X[:,1], (500,1)))

# plotting
plt.figure(figsize=(7,7))
sns.scatterplot(x=X[:,0], y=X[:,1], hue=lr_intact_pred)
plt.legend(loc=1)
plt.xlabel('$X_1$')
plt.ylabel('$X_2$');
```

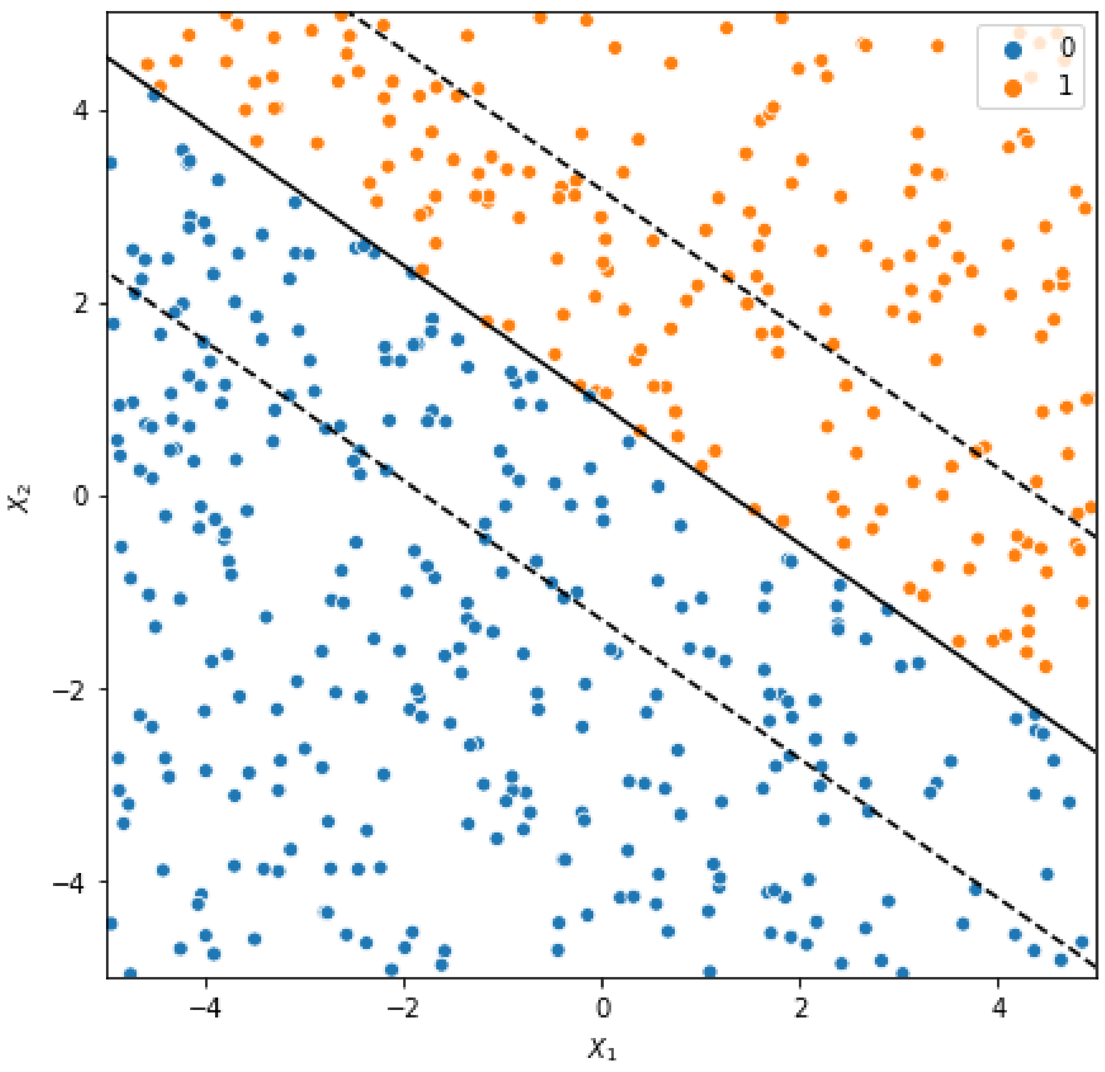


8. Now, fit a support vector classifier (linear kernel) to the data with original $X_1$ and $X_2$ as predictors. Obtain a class prediction for each observation. Plot the observations, colored according to the predicted class labels.

[14]:
```
# svc fitting
svm_lr2 = SVC(kernel='linear').fit(X,y)

# obtaining predictions
svm_lr2_pred = svm_lr2.predict(X)

# plotting
plt.figure(figsize=(7,7))
sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=svm_lr2_pred)
XX, YY = np.mgrid[-5:5:200j, -5:5:200j]
Z = svm_lr2.decision_function(np.c_[XX.ravel(), YY.ravel()])
Z = Z.reshape(XX.shape)
plt.contour(XX, YY, Z, colors=['k', 'k', 'k'], linestyles=['--', '-', '--'],
            levels=[-.5, 0, .5])
plt.legend(loc=1)
plt.xlabel('$X_1$')
plt.ylabel('$X_2$');
```

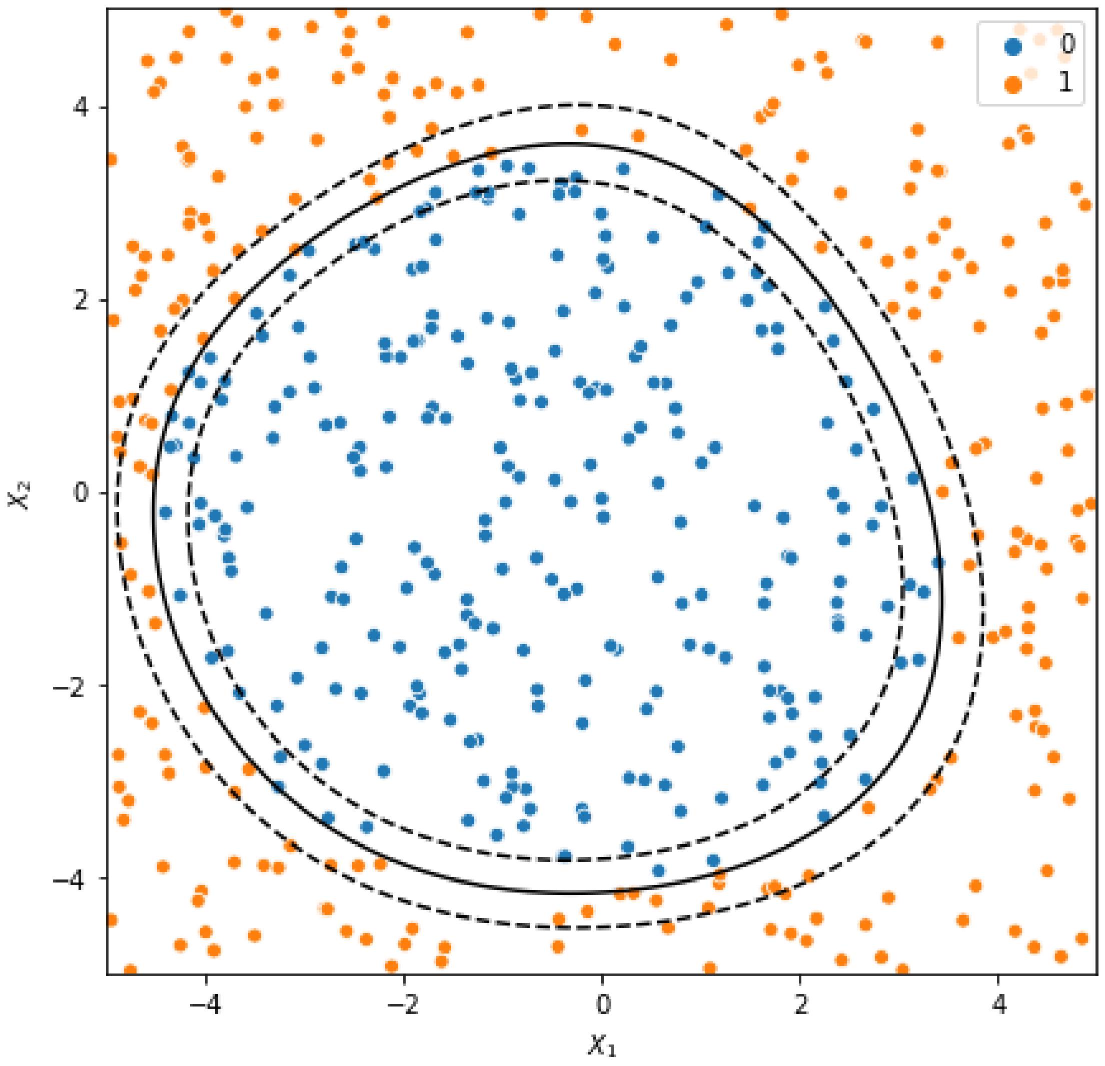


9. Fit a SVM using a non-linear kernel to the data. Obtain a class prediction for each observation. Plot the observations, colored according to the predicted class labels.

I will use the radial kernel, which is non-linear, to fit the data.

[15]:
```
# svm with radial kernel fitting
svm_rk2 = SVC(kernel='rbf').fit(X,y)
svm_rk2_pred = svm_rk2.predict(X)

plt.figure(figsize=(7,7))
sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=svm_rk2_pred)
XX, YY = np.mgrid[-5:5:200j, -5:5:200j]
Z = svm_rk2.decision_function(np.c_[XX.ravel(), YY.ravel()])
Z = Z.reshape(XX.shape)
plt.contour(XX, YY, Z, colors=['k', 'k', 'k'], linestyles=['--', '-', '--'],
             levels=[-.5, 0, .5])
plt.legend(loc=1)
plt.xlabel('$X_1$')
plt.ylabel('$X_2$');
```



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

[16]:
```
print("logistic regression error rate:", sum(lr_pred != y) / len(y))
print("logistic regression with polynomial term error rate:", sum(lr_sq_pred !=␣
 ↪y) / len(y))
print("logistic regression with interaction term error rate:",␣
 ↪sum(lr_intact_pred != y) / len(y))
print("svc error rate:", sum(svm_lr2_pred != y) / len(y))
```

```python
print("svc error rate:", sum(svm_rk2_pred != y) / len(y))
```

```
logistic regression error rate: 0.4
logistic regression with polynomial term error rate: 0.162
logistic regression with interaction term error rate: 0.346
svc error rate: 0.372
svc error rate: 0.132
```

We can see that logistic regression and SVC performed quite similarly. Their decision boundaries are almost identical, although they both did not classify the classes well. In addition, logistic regression using polynomial terms showed similar performances and decision boundaries with the SVM with a radial kernel. Their performance was much better than the rest of the methods. On the contrary, logistic regression using interaction term ($X_1 \times X_2$) showed a non-linear boundary that is quite different from any other boundaries, which did not reflect the original boundary correctly.

One thing to note is that although the decision boundary looked quite similar for the two cases mentioned above (logistic regression-SVC, logistic regression with polynomial term-SVM with a radial kernel), the error rate seems to suggest that the SVC and SVM are doing slightly better than logistic regression. This might be suggesting that SVC and SVM perform slightly better when there is an overlap in the boundaries.

Another thing to note is that logistic regression performed quite badly even if we used a non-linear function (log) if the function is different from the true function behind the data. Also, I think logistic regression would have performed similar or better than SVM with radial kernel if it was given the true function behind the data ($X_1 + X_1^2 + X_2 + X_2^2$). This suggests that although we can create a non-linear boundary using logistic regression, the performance of that boundary will be quite dependent on the non-linear function we feed it.

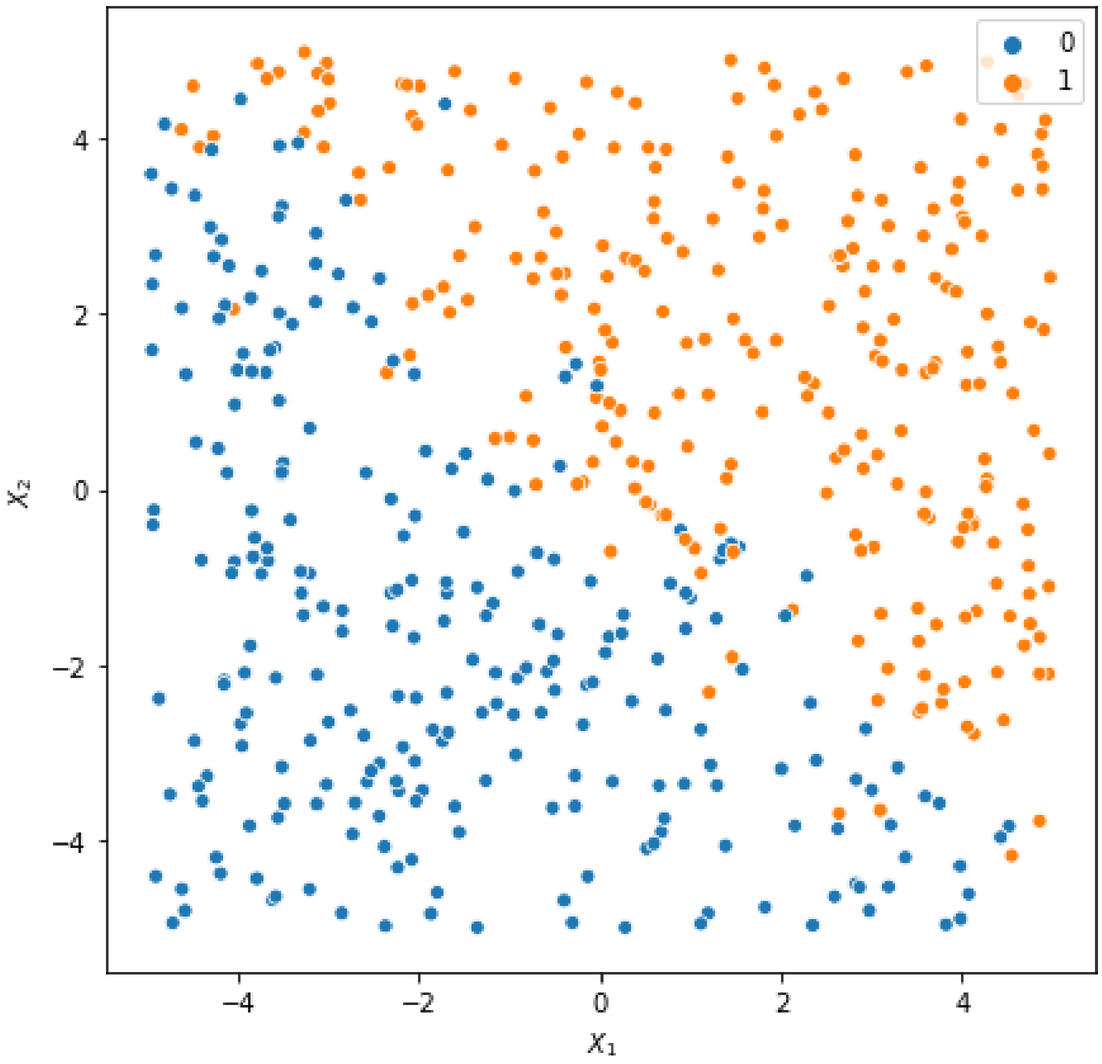
In terms of the trade-off between estimating non-linear decision boundaries and linear decision boundaries is roughly trade-off between accuracy and difficulty. Needless to say, non-linear methods will perform much better when we are dealing with classification problems with non-linear decision boundaries. However, this comes with a cost - non-linear methods might be harder to optimize or computationally more expensive. For logistic regression, we already discussed that they perform very well when fed in the proper functions (2nd-degree polynomial in the above example) but perform terribly when fed in non-linear functions that are not similar to the true structure (interaction terms in the above example). Therefore, the logistic regression using non-linear functions will be very hard to use in the real-world setting, since we almost always do not know the true structure in a real-world setting. This difficulty in finding out the proper non-linear function could be a trade-off in using a non-linear method and using a linear method in logistic regression. SVM and SVC share the same trade-off. SVM with non-linear kernel will perform better than SVC in non-linear decision boundary classification problems, but we have to choose the proper kernel and optimize the hyperparameters in this case, too. This will be easier in this case since we can use cross-validation to tune the parameters rather than having to feed in random non-linear functions to the model, but this will be much more computationally expensive. SVC indeed has some hyperparameters, such as C, but the number of hyperparameters will increase as we choose non-linear kernels. Therefore, the trade-off in SVC/SVM will be that SVM with non-linear methods will perform better in classification problems with non-linear decision boundaries, but it will become more computationally expensive since we have to tune a lot of hyperparameters. Another thing to note is that linear methods are easier to interpret in most cases, which could be another trade-off between non-linear methods and linear methods.

In terms of the trade-off between SVC/SVM and logistic regression, the main trade-off is that SVC/SVM is computationally more expensive, especially considering the tuning of hyperparameters using cross-validation. However, logistic regression might be more difficult to use when dealing with non-linear decision boundary problems, since we have to find the appropriate non-linear function. Therefore, in a non-linear decision boundary classification problem, I think it will be better to use SVM with a non-linear kernel if we do not know the true structure behind it. If we do know the true structure, it will be better to use logistic regression since it is computationally cheaper. When we are dealing with classification problem with linear decision boundary, I think it will be better to use logistic regression, since it is computationally cheaper and show similar performance with SVC.

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

[17]:
```
np.random.seed(26)
x11_1 = np.random.uniform(-5, 5, 500)
x11_2 = np.random.uniform(-5, 5, 500)
epsilon = np.random.normal(0, 1, 500)
X11 = np.column_stack([x11_1, x11_2])
y11 = np.int64((x11_1 + x11_2 + epsilon) > 0)
```

[18]:
```
plt.figure(figsize=(7,7))
sns.scatterplot(x=X11[:, 0], y=X11[:, 1], hue=y11)
plt.legend(loc=1)
plt.xlabel('$X_1$')
plt.ylabel('$X_2$');
```



As we can see clearly from the above plots, the decision boundary is linear but the boundary does not separate the classes perfectly. Thus, the decision boundary is just barely linearly separable.

12. 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?

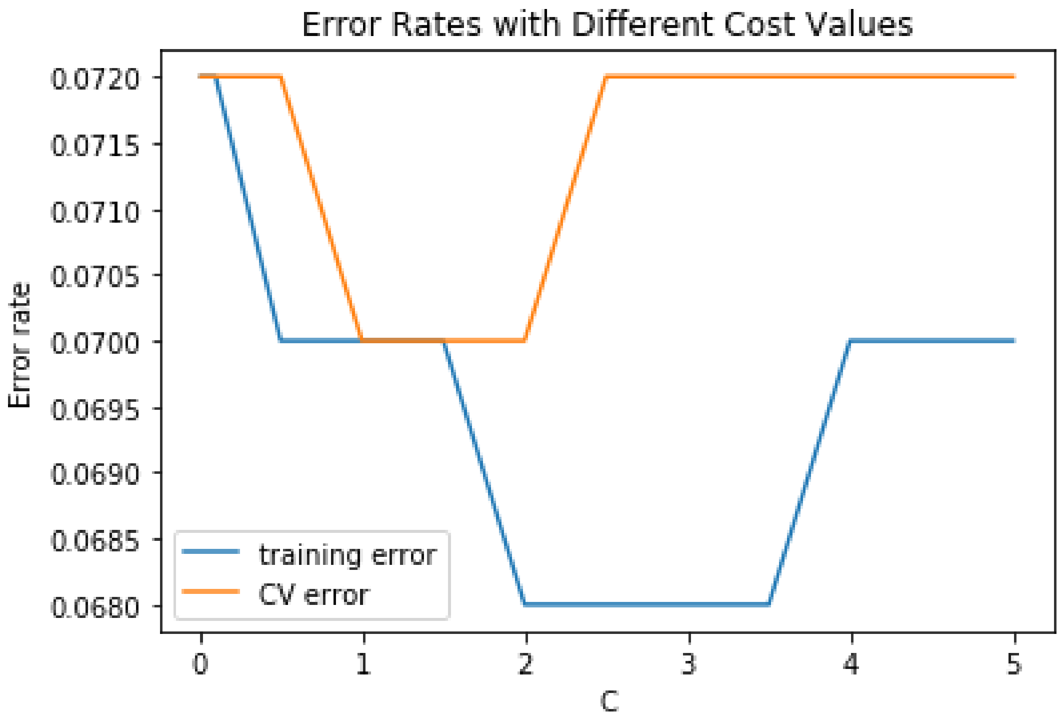
The C parameter is the regularization parameter (cost parameter) in SKLearn SVMs. The regularization strength is inversely proportional to C in SKLearn, so a huge value of C is roughly equivalent to a small value of `cost` discussed in the problem.

[19]:
```
costs = np.append(np.array([0.01, 0.05, 0.1, 0.5]), np.linspace(1, 5, 9))
best_cv_err = 1
best_tr_err = 1
cv_errs = []
```

```
tr_errs = []
for c in costs:
    svm_12 = SVC(C=c, kernel='linear', random_state=178)
    cv_err = np.mean(1 - cross_val_score(svm_12, X11, y11, cv=10))
    cv_errs.append(cv_err)
    if np.round(cv_err, decimals=4) < np.round(best_cv_err, decimals=4):
        best_cv_err = cv_err
        best_cv_c = c
    tr_err = 1 - svm_12.fit(X11, y11).score(X11, y11)
    tr_errs.append(tr_err)
    if np.round(tr_err, decimals=4) < np.round(best_tr_err, decimals=4):
        best_tr_err = tr_err
        best_tr_c = c
```

[20]:

```
sns.lineplot(costs, tr_errs)
sns.lineplot(costs, cv_errs)
plt.legend(['training error', 'CV error'])
plt.xlabel('C')
plt.ylabel('Error rate')
plt.title('Error Rates with Different Cost Values');
```



[21]:

```
for idx, c in enumerate(costs):
    print("C=", c, "Training Errors:", int(np.round(tr_errs[idx] * 500)),
          "CV Errors", np.round(cv_errs[idx] * 50, decimals=1))
```

```
C= 0.01 Training Errors: 36 CV Errors 3.6
C= 0.05 Training Errors: 36 CV Errors 3.6
C= 0.1 Training Errors: 36 CV Errors 3.6
C= 0.5 Training Errors: 35 CV Errors 3.6
C= 1.0 Training Errors: 35 CV Errors 3.5
C= 1.5 Training Errors: 35 CV Errors 3.5
C= 2.0 Training Errors: 34 CV Errors 3.5
C= 2.5 Training Errors: 34 CV Errors 3.6
C= 3.0 Training Errors: 34 CV Errors 3.6
C= 3.5 Training Errors: 34 CV Errors 3.6
C= 4.0 Training Errors: 35 CV Errors 3.6
C= 4.5 Training Errors: 35 CV Errors 3.6
C= 5.0 Training Errors: 35 CV Errors 3.6
```

We can see that training error shows a tendency to decrease as C becomes larger until C = 3.5, then increases as C become larger. Cross-validation error shows a similar pattern, but the C value in the transition point is a little bit smaller than that of training error (C = 2). Overall, I think training errors are closely related to the cross-validation errors since the difference between the number of errors is less than 2 in all C. Also, they express the same pattern of decreasing and increasing as C becomes larger.

13. 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?

[22]:
```
# generating test data set
np.random.seed(32)
x13_1 = np.random.uniform(-5, 5, 500)
x13_2 = np.random.uniform(-5, 5, 500)
epsilon = np.random.normal(0, 1, 500)
X13 = np.column_stack([x13_1, x13_2])
y13 = np.int64((x13_1 + x13_2 + epsilon) > 0)
```

[23]:
```
best_te_err = 1
te_errs = []
for idx, c in enumerate(costs):
    svm_13 = SVC(C=c, kernel='linear', random_state=178)
    te_err = 1 - svm_13.fit(X11, y11).score(X13, y13)
    print("C=", c, "Training Errors:", int(np.round(tr_errs[idx] * 500)),
          "CV Errors", np.round(cv_errs[idx] * 50, decimals=1),
          "Test Errors:", int(np.round(te_err * 500)))
    te_errs.append(te_err)
    if np.round(te_err, decimals=4) < np.round(best_te_err, decimals=4):
        best_te_err = te_err
        best_te_c = c
```

```
print("Best C for Training Error:", best_tr_c, "CV error:", best_cv_c, "Test␣
↪Error:", best_te_c)
```

```
C= 0.01 Training Errors: 36 CV Errors 3.6 Test Errors: 38
C= 0.05 Training Errors: 36 CV Errors 3.6 Test Errors: 37
C= 0.1 Training Errors: 36 CV Errors 3.6 Test Errors: 37
C= 0.5 Training Errors: 35 CV Errors 3.6 Test Errors: 37
C= 1.0 Training Errors: 35 CV Errors 3.5 Test Errors: 36
C= 1.5 Training Errors: 35 CV Errors 3.5 Test Errors: 36
C= 2.0 Training Errors: 34 CV Errors 3.5 Test Errors: 39
C= 2.5 Training Errors: 34 CV Errors 3.6 Test Errors: 39
C= 3.0 Training Errors: 34 CV Errors 3.6 Test Errors: 39
C= 3.5 Training Errors: 34 CV Errors 3.6 Test Errors: 39
C= 4.0 Training Errors: 35 CV Errors 3.6 Test Errors: 39
C= 4.5 Training Errors: 35 CV Errors 3.6 Test Errors: 39
C= 5.0 Training Errors: 35 CV Errors 3.6 Test Errors: 39
Best C for Training Error: 2.0 CV error: 1.0 Test Error: 1.0
```

Parameters C that showed the least training errors were 2.0, 2.5, 3.0, and 3.5. Parameters C that showed the least cross-validation errors were 1.0, 1.5, and 2.0. Parameters C that showed the least test errors were 1.0, and 1.5. Roughly, parameters C that showed the least cross-validation were similar to parameters C that showed the least test errors. This shows that cross-validation could be a good way to tune the cost parameter.

14. Discuss your results.

Since the data is not linearly separable (barely linearly separable), having a very low C, such as 0.01 in the above example, will give a relatively high error rate. This could be interpreted that we should allow some amount of error in the decision boundary to find a good separating hyperplane. However, we could see that if we allow too much error (by having a high C parameter), it will also show a relatively high error rate. Therefore, choosing an appropriate cost parameter (C value) is an essential part of using SVC/SVM successfully.

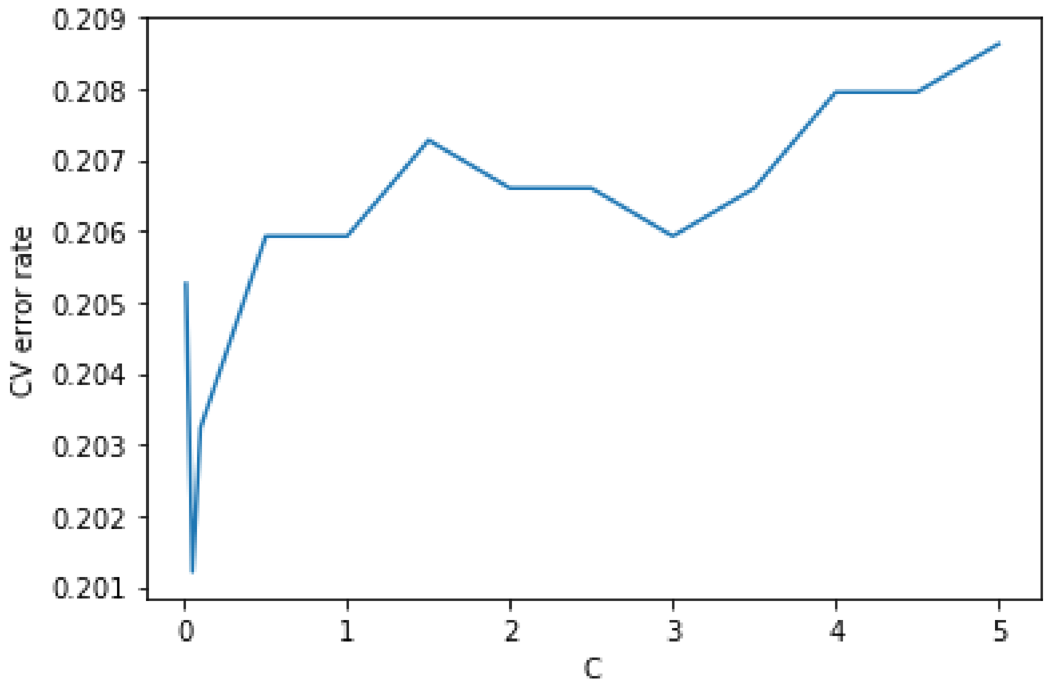
We also saw that cross-validation error can be predictive of test error. Therefore, similar to the other models we have discussed so far, cross-validation is a good way to tune the cost parameter (C value). Combined with the need to choose the appropriate cost parameter discussed above, this leads to the conclusion that using cross-validation to determine the cost parameter will be important in using SVC/SVM successfully.

15. 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.

[24]:
```
# Loading the data
gss_train = pd.read_csv('data/gss_train.csv')
gss_test = pd.read_csv('data/gss_test.csv')
train_X = gss_train.drop('colrac', axis=1)
train_Y = gss_train['colrac']
test_X = gss_test.drop('colrac', axis=1)
test_Y = gss_test['colrac']
```

```
[25]: costs = np.append(np.array([0.01, 0.05, 0.1, 0.5]), np.linspace(1, 5, 9))
      best_cv_err = 1
      cv_errs = []
      for c in costs:
          svm_15 = SVC(C=c, kernel='linear', random_state=533)
          cv_err = np.mean(1 - cross_val_score(svm_15, train_X, train_Y, cv=10))
          cv_errs.append(cv_err)
          if cv_err < best_cv_err:
              best_cv_err = cv_err
              best_cv_c = c
              best_cv_mod = svm_15
```

```
[26]: sns.lineplot(costs, cv_errs)
      plt.xlabel('C')
      plt.ylabel('CV error rate');
```



```
[27]: for idx, c in enumerate(costs):
          print('C value:', c, 'CV errors:', cv_errs[idx])
      print('Best C value:', best_cv_c, 'Best CV error:', best_cv_err)
```

```
C value: 0.01 CV errors: 0.20526482858697626
C value: 0.05 CV errors: 0.20121530926900055
C value: 0.1 CV errors: 0.2032378015599492
C value: 0.5 CV errors: 0.20593143479049525
C value: 1.0 CV errors: 0.20593143479049517
```

```
C value: 1.5 CV errors: 0.20728278614184656
C value: 2.0 CV errors: 0.20660711046617086
C value: 2.5 CV errors: 0.20660711046617086
C value: 3.0 CV errors: 0.2059314347904952
C value: 3.5 CV errors: 0.20660711046617086
C value: 4.0 CV errors: 0.20795846181752223
C value: 4.5 CV errors: 0.20795846181752223
C value: 5.0 CV errors: 0.20863413749319792
Best C value: 0.05 Best CV error: 0.20121530926900055
```

[28]:
```
print('Test Error of best model by CV:', sum(test_Y != best_cv_mod.fit(train_X,␣
↪train_Y).predict(test_X)) / len(test_Y))
```

```
Test Error of best model by CV: 0.20689655172413793
```

The cross-validation error rate showed a similar pattern with the pattern discussed in question 12 and 14 - it showed a tendency to decrease as C increase until a certain point, then showed a tendency to increase. This further shows that using cross-validation to get the optimal cost parameter is important in using SVC/SVM.

In terms of performance of the best model, I think SVC is showing decent performance especially considering that the null model error rate (predicting everything to be 0) is about 0.48 (discussed in the last problem set). The test error rate was very similar to the cross-validation error rate suggesting that the cross-validation performed well.

16. 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.).

16.1. SVM with radial kernel I note that I did not tune the degree parameter for the radial kernel since radial kernel does not really have a degree parameter.

[29]:
```
rbf_grid = {'kernel':['rbf'], 'gamma': np.logspace(-3,1,5),
            'C':np.append(np.array([0.01, 0.05, 0.1, 0.5]), np.linspace(1, 3,␣
↪5))}
svm16_rbf = GridSearchCV(estimator=SVC(), param_grid=rbf_grid, cv=10,␣
↪refit=True, n_jobs=4).fit(train_X, train_Y)
```

[30]:
```
for idx, param  in enumerate(svm16_rbf.cv_results_['params']):
    print("parameter", param, "CV Accuracy", svm16_rbf.
↪cv_results_['mean_test_score'][idx])
```

```
parameter {'C': 0.01, 'gamma': 0.001, 'kernel': 'rbf'} CV Accuracy
0.5253219662615635
parameter {'C': 0.01, 'gamma': 0.01, 'kernel': 'rbf'} CV Accuracy
0.5253219662615635
parameter {'C': 0.01, 'gamma': 0.1, 'kernel': 'rbf'} CV Accuracy
0.5253219662615635
parameter {'C': 0.01, 'gamma': 1.0, 'kernel': 'rbf'} CV Accuracy
0.5253219662615635
```

```
parameter {'C': 0.01, 'gamma': 10.0, 'kernel': 'rbf'} CV Accuracy
0.5253219662615635
parameter {'C': 0.05, 'gamma': 0.001, 'kernel': 'rbf'} CV Accuracy
0.6678079085797206
parameter {'C': 0.05, 'gamma': 0.01, 'kernel': 'rbf'} CV Accuracy
0.5313985126065662
parameter {'C': 0.05, 'gamma': 0.1, 'kernel': 'rbf'} CV Accuracy
0.5253219662615635
parameter {'C': 0.05, 'gamma': 1.0, 'kernel': 'rbf'} CV Accuracy
0.5253219662615635
parameter {'C': 0.05, 'gamma': 10.0, 'kernel': 'rbf'} CV Accuracy
0.5253219662615635
parameter {'C': 0.1, 'gamma': 0.001, 'kernel': 'rbf'} CV Accuracy
0.7076455650281154
parameter {'C': 0.1, 'gamma': 0.01, 'kernel': 'rbf'} CV Accuracy
0.655663885361872
parameter {'C': 0.1, 'gamma': 0.1, 'kernel': 'rbf'} CV Accuracy
0.5253219662615635
parameter {'C': 0.1, 'gamma': 1.0, 'kernel': 'rbf'} CV Accuracy
0.5253219662615635
parameter {'C': 0.1, 'gamma': 10.0, 'kernel': 'rbf'} CV Accuracy
0.5253219662615635
parameter {'C': 0.5, 'gamma': 0.001, 'kernel': 'rbf'} CV Accuracy
0.7542127698167966
parameter {'C': 0.5, 'gamma': 0.01, 'kernel': 'rbf'} CV Accuracy
0.7400462543079993
parameter {'C': 0.5, 'gamma': 0.1, 'kernel': 'rbf'} CV Accuracy
0.5253219662615635
parameter {'C': 0.5, 'gamma': 1.0, 'kernel': 'rbf'} CV Accuracy
0.5253219662615635
parameter {'C': 0.5, 'gamma': 10.0, 'kernel': 'rbf'} CV Accuracy
0.5253219662615635
parameter {'C': 1.0, 'gamma': 0.001, 'kernel': 'rbf'} CV Accuracy
0.7636586250680211
parameter {'C': 1.0, 'gamma': 0.01, 'kernel': 'rbf'} CV Accuracy
0.7481316887357157
parameter {'C': 1.0, 'gamma': 0.1, 'kernel': 'rbf'} CV Accuracy
0.5307319064030473
parameter {'C': 1.0, 'gamma': 1.0, 'kernel': 'rbf'} CV Accuracy
0.5253219662615635
parameter {'C': 1.0, 'gamma': 10.0, 'kernel': 'rbf'} CV Accuracy
0.5253219662615635
parameter {'C': 1.5, 'gamma': 0.001, 'kernel': 'rbf'} CV Accuracy
0.7724288046435698
parameter {'C': 1.5, 'gamma': 0.01, 'kernel': 'rbf'} CV Accuracy
0.7494830400870669
parameter {'C': 1.5, 'gamma': 0.1, 'kernel': 'rbf'} CV Accuracy
0.5381461998911664
```

```
parameter {'C': 1.5, 'gamma': 1.0, 'kernel': 'rbf'} CV Accuracy
0.5253219662615635
parameter {'C': 1.5, 'gamma': 10.0, 'kernel': 'rbf'} CV Accuracy
0.5253219662615635
parameter {'C': 2.0, 'gamma': 0.001, 'kernel': 'rbf'} CV Accuracy
0.772419735171413
parameter {'C': 2.0, 'gamma': 0.01, 'kernel': 'rbf'} CV Accuracy
0.7521857427897698
parameter {'C': 2.0, 'gamma': 0.1, 'kernel': 'rbf'} CV Accuracy
0.5381461998911664
parameter {'C': 2.0, 'gamma': 1.0, 'kernel': 'rbf'} CV Accuracy
0.5253219662615635
parameter {'C': 2.0, 'gamma': 10.0, 'kernel': 'rbf'} CV Accuracy
0.5253219662615635
parameter {'C': 2.5, 'gamma': 0.001, 'kernel': 'rbf'} CV Accuracy
0.7758026482858698
parameter {'C': 2.5, 'gamma': 0.01, 'kernel': 'rbf'} CV Accuracy
0.7521857427897697
parameter {'C': 2.5, 'gamma': 0.1, 'kernel': 'rbf'} CV Accuracy
0.5381461998911664
parameter {'C': 2.5, 'gamma': 1.0, 'kernel': 'rbf'} CV Accuracy
0.5253219662615635
parameter {'C': 2.5, 'gamma': 10.0, 'kernel': 'rbf'} CV Accuracy
0.5253219662615635
parameter {'C': 3.0, 'gamma': 0.001, 'kernel': 'rbf'} CV Accuracy
0.7764873934337022
parameter {'C': 3.0, 'gamma': 0.01, 'kernel': 'rbf'} CV Accuracy
0.7481362234717939
parameter {'C': 3.0, 'gamma': 0.1, 'kernel': 'rbf'} CV Accuracy
0.5381461998911664
parameter {'C': 3.0, 'gamma': 1.0, 'kernel': 'rbf'} CV Accuracy
0.5253219662615635
parameter {'C': 3.0, 'gamma': 10.0, 'kernel': 'rbf'} CV Accuracy
0.5253219662615635
```

```
[31]: print("best parameters:", svm16_rbf.best_params_, "best CV Accuracy", svm16_rbf.
      ↪best_score_)
```

```
best parameters: {'C': 3.0, 'gamma': 0.001, 'kernel': 'rbf'} best CV Accuracy
0.7764873934337022
```

```
[32]: print('Test Accuracy of best model by CV:', sum(test_Y == svm16_rbf.
      ↪predict(test_X)) / len(test_Y))
```

```
Test Accuracy of best model by CV: 0.7890466531440162
```

Examining the results presented above, we can see that having too small C value (smaller than 0.1) will show poor performance(lower than 0.6 accuracies except one) regardless of the gamma parameter. If the cost parameter is sufficiently high, I think the fit is more sensitive to the gamma parameter, since having appropriate gamma value (0.001 and 0.01) shows good performance (accuracy higher than 0.7 except one) while having less appropriate gamma value (larger than 0.01) shows worse performance (accuracy lower than 0.6). This shows that both parameters have to be properly tuned to make SVM function well.

In terms of the performance of the best model, I think it is showing decent performance. Again, the null model accuracy is about 0.52 in the data set, so this model is performing quite well. Also, The test accuracy was very similar to cross-validation accuracy suggesting that the cross-validation performed well.

16.2. SVMs with polynomial basis kernel

[33]:
```
poly_grid = {'kernel':['poly'], 'gamma': np.logspace(-3,1,5), 'degree':np.
↪linspace(1,3,3),
             'C':np.append(np.array([0.01, 0.05, 0.1, 0.5]), np.linspace(1, 3,␣
↪5))}
svm16_poly = GridSearchCV(estimator=SVC(), param_grid=poly_grid, cv=10,␣
↪refit=True, n_jobs=4).fit(train_X, train_Y)
```

[34]:
```
for idx, param  in enumerate(svm16_poly.cv_results_['params']):
    print("parameter", param, "CV Accuracy", svm16_poly.
↪cv_results_['mean_test_score'][idx])
```

```
parameter {'C': 0.01, 'degree': 1.0, 'gamma': 0.001, 'kernel': 'poly'} CV
Accuracy 0.5253219662615635
parameter {'C': 0.01, 'degree': 1.0, 'gamma': 0.01, 'kernel': 'poly'} CV
Accuracy 0.6968619626337746
parameter {'C': 0.01, 'degree': 1.0, 'gamma': 0.1, 'kernel': 'poly'} CV Accuracy
0.7602983856339562
parameter {'C': 0.01, 'degree': 1.0, 'gamma': 1.0, 'kernel': 'poly'} CV Accuracy
0.7947351714130237
parameter {'C': 0.01, 'degree': 1.0, 'gamma': 10.0, 'kernel': 'poly'} CV
Accuracy 0.7974378741157265
parameter {'C': 0.01, 'degree': 2.0, 'gamma': 0.001, 'kernel': 'poly'} CV
Accuracy 0.699546526392164
parameter {'C': 0.01, 'degree': 2.0, 'gamma': 0.01, 'kernel': 'poly'} CV
Accuracy 0.7886450208597859
parameter {'C': 0.01, 'degree': 2.0, 'gamma': 0.1, 'kernel': 'poly'} CV Accuracy
0.7785280246689642
parameter {'C': 0.01, 'degree': 2.0, 'gamma': 1.0, 'kernel': 'poly'} CV Accuracy
0.7339470342826048
parameter {'C': 0.01, 'degree': 2.0, 'gamma': 10.0, 'kernel': 'poly'} CV
Accuracy 0.7211364048612371
parameter {'C': 0.01, 'degree': 3.0, 'gamma': 0.001, 'kernel': 'poly'} CV
Accuracy 0.7332985670233991
parameter {'C': 0.01, 'degree': 3.0, 'gamma': 0.01, 'kernel': 'poly'} CV
Accuracy 0.7879738799201886
parameter {'C': 0.01, 'degree': 3.0, 'gamma': 0.1, 'kernel': 'poly'} CV Accuracy
```

```
0.7393433702158534
parameter {'C': 0.01, 'degree': 3.0, 'gamma': 1.0, 'kernel': 'poly'} CV Accuracy
0.7386676945401778
parameter {'C': 0.01, 'degree': 3.0, 'gamma': 10.0, 'kernel': 'poly'} CV
Accuracy 0.7379965536005806
parameter {'C': 0.05, 'degree': 1.0, 'gamma': 0.001, 'kernel': 'poly'} CV
Accuracy 0.69549247233811
parameter {'C': 0.05, 'degree': 1.0, 'gamma': 0.01, 'kernel': 'poly'} CV
Accuracy 0.7474650825321966
parameter {'C': 0.05, 'degree': 1.0, 'gamma': 0.1, 'kernel': 'poly'} CV Accuracy
0.7893206965354616
parameter {'C': 0.05, 'degree': 1.0, 'gamma': 1.0, 'kernel': 'poly'} CV Accuracy
0.7987846907309993
parameter {'C': 0.05, 'degree': 1.0, 'gamma': 10.0, 'kernel': 'poly'} CV
Accuracy 0.7940685652095048
parameter {'C': 0.05, 'degree': 2.0, 'gamma': 0.001, 'kernel': 'poly'} CV
Accuracy 0.7407264647197532
parameter {'C': 0.05, 'degree': 2.0, 'gamma': 0.01, 'kernel': 'poly'} CV
Accuracy 0.797433339379648
parameter {'C': 0.05, 'degree': 2.0, 'gamma': 0.1, 'kernel': 'poly'} CV Accuracy
0.7643206965354616
parameter {'C': 0.05, 'degree': 2.0, 'gamma': 1.0, 'kernel': 'poly'} CV Accuracy
0.7238436423000182
parameter {'C': 0.05, 'degree': 2.0, 'gamma': 10.0, 'kernel': 'poly'} CV
Accuracy 0.7211364048612371
parameter {'C': 0.05, 'degree': 3.0, 'gamma': 0.001, 'kernel': 'poly'} CV
Accuracy 0.7616406675131507
parameter {'C': 0.05, 'degree': 3.0, 'gamma': 0.01, 'kernel': 'poly'} CV
Accuracy 0.7751269726101941
parameter {'C': 0.05, 'degree': 3.0, 'gamma': 0.1, 'kernel': 'poly'} CV Accuracy
0.7393433702158534
parameter {'C': 0.05, 'degree': 3.0, 'gamma': 1.0, 'kernel': 'poly'} CV Accuracy
0.7386676945401778
parameter {'C': 0.05, 'degree': 3.0, 'gamma': 10.0, 'kernel': 'poly'} CV
Accuracy 0.7379965536005806
parameter {'C': 0.1, 'degree': 1.0, 'gamma': 0.001, 'kernel': 'poly'} CV
Accuracy 0.6968619626337746
parameter {'C': 0.1, 'degree': 1.0, 'gamma': 0.01, 'kernel': 'poly'} CV Accuracy
0.7602983856339562
parameter {'C': 0.1, 'degree': 1.0, 'gamma': 0.1, 'kernel': 'poly'} CV Accuracy
0.7954108470886995
parameter {'C': 0.1, 'degree': 1.0, 'gamma': 1.0, 'kernel': 'poly'} CV Accuracy
0.7967621984400509
parameter {'C': 0.1, 'degree': 1.0, 'gamma': 10.0, 'kernel': 'poly'} CV Accuracy
0.7940685652095048
parameter {'C': 0.1, 'degree': 2.0, 'gamma': 0.001, 'kernel': 'poly'} CV
Accuracy 0.7548884454924724
parameter {'C': 0.1, 'degree': 2.0, 'gamma': 0.01, 'kernel': 'poly'} CV Accuracy
```

```
0.7981044803192454
parameter {'C': 0.1, 'degree': 2.0, 'gamma': 0.1, 'kernel': 'poly'} CV Accuracy
0.7650009069472156
parameter {'C': 0.1, 'degree': 2.0, 'gamma': 1.0, 'kernel': 'poly'} CV Accuracy
0.7238436423000182
parameter {'C': 0.1, 'degree': 2.0, 'gamma': 10.0, 'kernel': 'poly'} CV Accuracy
0.7211364048612371
parameter {'C': 0.1, 'degree': 3.0, 'gamma': 0.001, 'kernel': 'poly'} CV
Accuracy 0.7778432795211319
parameter {'C': 0.1, 'degree': 3.0, 'gamma': 0.01, 'kernel': 'poly'} CV Accuracy
0.7771539996372211
parameter {'C': 0.1, 'degree': 3.0, 'gamma': 0.1, 'kernel': 'poly'} CV Accuracy
0.7393433702158534
parameter {'C': 0.1, 'degree': 3.0, 'gamma': 1.0, 'kernel': 'poly'} CV Accuracy
0.7386676945401778
parameter {'C': 0.1, 'degree': 3.0, 'gamma': 10.0, 'kernel': 'poly'} CV Accuracy
0.7379965536005806
parameter {'C': 0.5, 'degree': 1.0, 'gamma': 0.001, 'kernel': 'poly'} CV
Accuracy 0.7481407582078723
parameter {'C': 0.5, 'degree': 1.0, 'gamma': 0.01, 'kernel': 'poly'} CV Accuracy
0.7893206965354616
parameter {'C': 0.5, 'degree': 1.0, 'gamma': 0.1, 'kernel': 'poly'} CV Accuracy
0.7987846907309993
parameter {'C': 0.5, 'degree': 1.0, 'gamma': 1.0, 'kernel': 'poly'} CV Accuracy
0.7940685652095048
parameter {'C': 0.5, 'degree': 1.0, 'gamma': 10.0, 'kernel': 'poly'} CV Accuracy
0.7913658625068021
parameter {'C': 0.5, 'degree': 2.0, 'gamma': 0.001, 'kernel': 'poly'} CV
Accuracy 0.7798703065481588
parameter {'C': 0.5, 'degree': 2.0, 'gamma': 0.01, 'kernel': 'poly'} CV Accuracy
0.7812216578995103
parameter {'C': 0.5, 'degree': 2.0, 'gamma': 0.1, 'kernel': 'poly'} CV Accuracy
0.7407083257754399
parameter {'C': 0.5, 'degree': 2.0, 'gamma': 1.0, 'kernel': 'poly'} CV Accuracy
0.7238436423000182
parameter {'C': 0.5, 'degree': 2.0, 'gamma': 10.0, 'kernel': 'poly'} CV Accuracy
0.7211364048612371
parameter {'C': 0.5, 'degree': 3.0, 'gamma': 0.001, 'kernel': 'poly'} CV
Accuracy 0.7920097950299292
parameter {'C': 0.5, 'degree': 3.0, 'gamma': 0.01, 'kernel': 'poly'} CV Accuracy
0.7481135497914022
parameter {'C': 0.5, 'degree': 3.0, 'gamma': 0.1, 'kernel': 'poly'} CV Accuracy
0.7393433702158534
parameter {'C': 0.5, 'degree': 3.0, 'gamma': 1.0, 'kernel': 'poly'} CV Accuracy
0.7386676945401778
parameter {'C': 0.5, 'degree': 3.0, 'gamma': 10.0, 'kernel': 'poly'} CV Accuracy
0.7379965536005806
parameter {'C': 1.0, 'degree': 1.0, 'gamma': 0.001, 'kernel': 'poly'} CV
```

```
Accuracy 0.7602983856339562
parameter {'C': 1.0, 'degree': 1.0, 'gamma': 0.01, 'kernel': 'poly'} CV Accuracy
0.7947351714130237
parameter {'C': 1.0, 'degree': 1.0, 'gamma': 0.1, 'kernel': 'poly'} CV Accuracy
0.7967621984400509
parameter {'C': 1.0, 'degree': 1.0, 'gamma': 1.0, 'kernel': 'poly'} CV Accuracy
0.7940685652095048
parameter {'C': 1.0, 'degree': 1.0, 'gamma': 10.0, 'kernel': 'poly'} CV Accuracy
0.7906901868311265
parameter {'C': 1.0, 'degree': 2.0, 'gamma': 0.001, 'kernel': 'poly'} CV
Accuracy 0.7886450208597859
parameter {'C': 1.0, 'degree': 2.0, 'gamma': 0.01, 'kernel': 'poly'} CV Accuracy
0.7785280246689642
parameter {'C': 1.0, 'degree': 2.0, 'gamma': 0.1, 'kernel': 'poly'} CV Accuracy
0.7332804280790858
parameter {'C': 1.0, 'degree': 2.0, 'gamma': 1.0, 'kernel': 'poly'} CV Accuracy
0.7238436423000182
parameter {'C': 1.0, 'degree': 2.0, 'gamma': 10.0, 'kernel': 'poly'} CV Accuracy
0.7211364048612371
parameter {'C': 1.0, 'degree': 3.0, 'gamma': 0.001, 'kernel': 'poly'} CV
Accuracy 0.7947261019408669
parameter {'C': 1.0, 'degree': 3.0, 'gamma': 0.01, 'kernel': 'poly'} CV Accuracy
0.7380010883366588
parameter {'C': 1.0, 'degree': 3.0, 'gamma': 0.1, 'kernel': 'poly'} CV Accuracy
0.7393433702158534
parameter {'C': 1.0, 'degree': 3.0, 'gamma': 1.0, 'kernel': 'poly'} CV Accuracy
0.7386676945401778
parameter {'C': 1.0, 'degree': 3.0, 'gamma': 10.0, 'kernel': 'poly'} CV Accuracy
0.7379965536005806
parameter {'C': 1.5, 'degree': 1.0, 'gamma': 0.001, 'kernel': 'poly'} CV
Accuracy 0.7697533103573372
parameter {'C': 1.5, 'degree': 1.0, 'gamma': 0.01, 'kernel': 'poly'} CV Accuracy
0.7960819880282967
parameter {'C': 1.5, 'degree': 1.0, 'gamma': 0.1, 'kernel': 'poly'} CV Accuracy
0.794739706149102
parameter {'C': 1.5, 'degree': 1.0, 'gamma': 1.0, 'kernel': 'poly'} CV Accuracy
0.7927172138581535
parameter {'C': 1.5, 'degree': 1.0, 'gamma': 10.0, 'kernel': 'poly'} CV Accuracy
0.7906901868311265
parameter {'C': 1.5, 'degree': 2.0, 'gamma': 0.001, 'kernel': 'poly'} CV
Accuracy 0.7879602757119535
parameter {'C': 1.5, 'degree': 2.0, 'gamma': 0.01, 'kernel': 'poly'} CV Accuracy
0.7677081443859967
parameter {'C': 1.5, 'degree': 2.0, 'gamma': 0.1, 'kernel': 'poly'} CV Accuracy
0.7258479956466534
parameter {'C': 1.5, 'degree': 2.0, 'gamma': 1.0, 'kernel': 'poly'} CV Accuracy
0.7238436423000182
parameter {'C': 1.5, 'degree': 2.0, 'gamma': 10.0, 'kernel': 'poly'} CV Accuracy
```

```
0.7211364048612371
parameter {'C': 1.5, 'degree': 3.0, 'gamma': 0.001, 'kernel': 'poly'} CV
Accuracy 0.7954017776165427
parameter {'C': 1.5, 'degree': 3.0, 'gamma': 0.01, 'kernel': 'poly'} CV Accuracy
0.7379920188645022
parameter {'C': 1.5, 'degree': 3.0, 'gamma': 0.1, 'kernel': 'poly'} CV Accuracy
0.7393433702158534
parameter {'C': 1.5, 'degree': 3.0, 'gamma': 1.0, 'kernel': 'poly'} CV Accuracy
0.7386676945401778
parameter {'C': 1.5, 'degree': 3.0, 'gamma': 10.0, 'kernel': 'poly'} CV Accuracy
0.7379965536005806
parameter {'C': 2.0, 'degree': 1.0, 'gamma': 0.001, 'kernel': 'poly'} CV
Accuracy 0.7704199165608562
parameter {'C': 2.0, 'degree': 1.0, 'gamma': 0.01, 'kernel': 'poly'} CV Accuracy
0.7981044803192454
parameter {'C': 2.0, 'degree': 1.0, 'gamma': 0.1, 'kernel': 'poly'} CV Accuracy
0.7981180845274805
parameter {'C': 2.0, 'degree': 1.0, 'gamma': 1.0, 'kernel': 'poly'} CV Accuracy
0.7933928895338291
parameter {'C': 2.0, 'degree': 1.0, 'gamma': 10.0, 'kernel': 'poly'} CV Accuracy
0.7906901868311265
parameter {'C': 2.0, 'degree': 2.0, 'gamma': 0.001, 'kernel': 'poly'} CV
Accuracy 0.7920188645020859
parameter {'C': 2.0, 'degree': 2.0, 'gamma': 0.01, 'kernel': 'poly'} CV Accuracy
0.7731135497914021
parameter {'C': 2.0, 'degree': 2.0, 'gamma': 0.1, 'kernel': 'poly'} CV Accuracy
0.7298975149646291
parameter {'C': 2.0, 'degree': 2.0, 'gamma': 1.0, 'kernel': 'poly'} CV Accuracy
0.7238436423000182
parameter {'C': 2.0, 'degree': 2.0, 'gamma': 10.0, 'kernel': 'poly'} CV Accuracy
0.7211364048612371
parameter {'C': 2.0, 'degree': 3.0, 'gamma': 0.001, 'kernel': 'poly'} CV
Accuracy 0.7960865227643751
parameter {'C': 2.0, 'degree': 3.0, 'gamma': 0.01, 'kernel': 'poly'} CV Accuracy
0.7400235806276074
parameter {'C': 2.0, 'degree': 3.0, 'gamma': 0.1, 'kernel': 'poly'} CV Accuracy
0.7393433702158534
parameter {'C': 2.0, 'degree': 3.0, 'gamma': 1.0, 'kernel': 'poly'} CV Accuracy
0.7386676945401778
parameter {'C': 2.0, 'degree': 3.0, 'gamma': 10.0, 'kernel': 'poly'} CV Accuracy
0.7379965536005806
parameter {'C': 2.5, 'degree': 1.0, 'gamma': 0.001, 'kernel': 'poly'} CV
Accuracy 0.7778523489932885
parameter {'C': 2.5, 'degree': 1.0, 'gamma': 0.01, 'kernel': 'poly'} CV Accuracy
0.7967576637039724
parameter {'C': 2.5, 'degree': 1.0, 'gamma': 0.1, 'kernel': 'poly'} CV Accuracy
0.7954153818247778
parameter {'C': 2.5, 'degree': 1.0, 'gamma': 1.0, 'kernel': 'poly'} CV Accuracy
```

```
0.7933928895338291
parameter {'C': 2.5, 'degree': 1.0, 'gamma': 10.0, 'kernel': 'poly'} CV Accuracy
0.7913658625068021
parameter {'C': 2.5, 'degree': 2.0, 'gamma': 0.001, 'kernel': 'poly'} CV
Accuracy 0.7920233992381643
parameter {'C': 2.5, 'degree': 2.0, 'gamma': 0.01, 'kernel': 'poly'} CV Accuracy
0.7717667331761292
parameter {'C': 2.5, 'degree': 2.0, 'gamma': 0.1, 'kernel': 'poly'} CV Accuracy
0.7299111191728641
parameter {'C': 2.5, 'degree': 2.0, 'gamma': 1.0, 'kernel': 'poly'} CV Accuracy
0.7238436423000182
parameter {'C': 2.5, 'degree': 2.0, 'gamma': 10.0, 'kernel': 'poly'} CV Accuracy
0.7211364048612371
parameter {'C': 2.5, 'degree': 3.0, 'gamma': 0.001, 'kernel': 'poly'} CV
Accuracy 0.795406312352621
parameter {'C': 2.5, 'degree': 3.0, 'gamma': 0.01, 'kernel': 'poly'} CV Accuracy
0.7379920188645022
parameter {'C': 2.5, 'degree': 3.0, 'gamma': 0.1, 'kernel': 'poly'} CV Accuracy
0.7393433702158534
parameter {'C': 2.5, 'degree': 3.0, 'gamma': 1.0, 'kernel': 'poly'} CV Accuracy
0.7386676945401778
parameter {'C': 2.5, 'degree': 3.0, 'gamma': 10.0, 'kernel': 'poly'} CV Accuracy
0.7379965536005806
parameter {'C': 3.0, 'degree': 1.0, 'gamma': 0.001, 'kernel': 'poly'} CV
Accuracy 0.78595138762924
parameter {'C': 3.0, 'degree': 1.0, 'gamma': 0.01, 'kernel': 'poly'} CV Accuracy
0.8001360420823509
parameter {'C': 3.0, 'degree': 1.0, 'gamma': 0.1, 'kernel': 'poly'} CV Accuracy
0.7947397061491022
parameter {'C': 3.0, 'degree': 1.0, 'gamma': 1.0, 'kernel': 'poly'} CV Accuracy
0.7940685652095048
parameter {'C': 3.0, 'degree': 1.0, 'gamma': 10.0, 'kernel': 'poly'} CV Accuracy
0.7927172138581533
parameter {'C': 3.0, 'degree': 2.0, 'gamma': 0.001, 'kernel': 'poly'} CV
Accuracy 0.7927036096499184
parameter {'C': 3.0, 'degree': 2.0, 'gamma': 0.01, 'kernel': 'poly'} CV Accuracy
0.7724378741157264
parameter {'C': 3.0, 'degree': 2.0, 'gamma': 0.1, 'kernel': 'poly'} CV Accuracy
0.7258661345909668
parameter {'C': 3.0, 'degree': 2.0, 'gamma': 1.0, 'kernel': 'poly'} CV Accuracy
0.7238436423000182
parameter {'C': 3.0, 'degree': 2.0, 'gamma': 10.0, 'kernel': 'poly'} CV Accuracy
0.7211364048612371
parameter {'C': 3.0, 'degree': 3.0, 'gamma': 0.001, 'kernel': 'poly'} CV
Accuracy 0.7900009069472157
parameter {'C': 3.0, 'degree': 3.0, 'gamma': 0.01, 'kernel': 'poly'} CV Accuracy
0.7386676945401778
parameter {'C': 3.0, 'degree': 3.0, 'gamma': 0.1, 'kernel': 'poly'} CV Accuracy
```

```
0.7393433702158534
parameter {'C': 3.0, 'degree': 3.0, 'gamma': 1.0, 'kernel': 'poly'} CV Accuracy
0.7386676945401778
parameter {'C': 3.0, 'degree': 3.0, 'gamma': 10.0, 'kernel': 'poly'} CV Accuracy
0.7379965536005806
```

```
[35]: print("best parameters:", svm16_poly.best_params_, "best CV score", svm16_poly.
       ↪best_score_)
```

```
best parameters: {'C': 3.0, 'degree': 1.0, 'gamma': 0.01, 'kernel': 'poly'} best
CV score 0.8001360420823509
```

```
[36]: print('Test Accuracy of best model by CV:', sum(test_Y == svm16_poly.
       ↪predict(test_X)) / len(test_Y))
```

```
Test Accuracy of best model by CV: 0.7890466531440162
```

SVM with polynomial basis kernel showed decent performance overall, showing more than 0.7 accuracies in almost all hyperparameter conditions. Since they performed relatively well in all combinations, it is relatively hard to see which parameter affected the performance the most.

The best performing model shows slightly better result than best performing SVM with a radial kernel since its cross-validation is slightly better than that of SVM with a radial kernel. (Although I note that they show the identical test accuracy.) Also, the fact that SVM using polynomial kernel performed quite well in almost all hyperparameter combinations suggests that polynomial kernel works better for this problem.

In terms of performance, this model is doing better than SVM with a radial kernel, showing good performance overall. However, it does not perform significantly better than SVC, implied by the fact that the best degree was 1 (SKLearn's implementation of SVM with a polynomial kernel with degree 1 is slightly different from SVC, but it is quite similar). I think it will be better to use SVC in this case, because it shows similar performance and SVC is computationally less expensive (considering fewer parameters to tune).