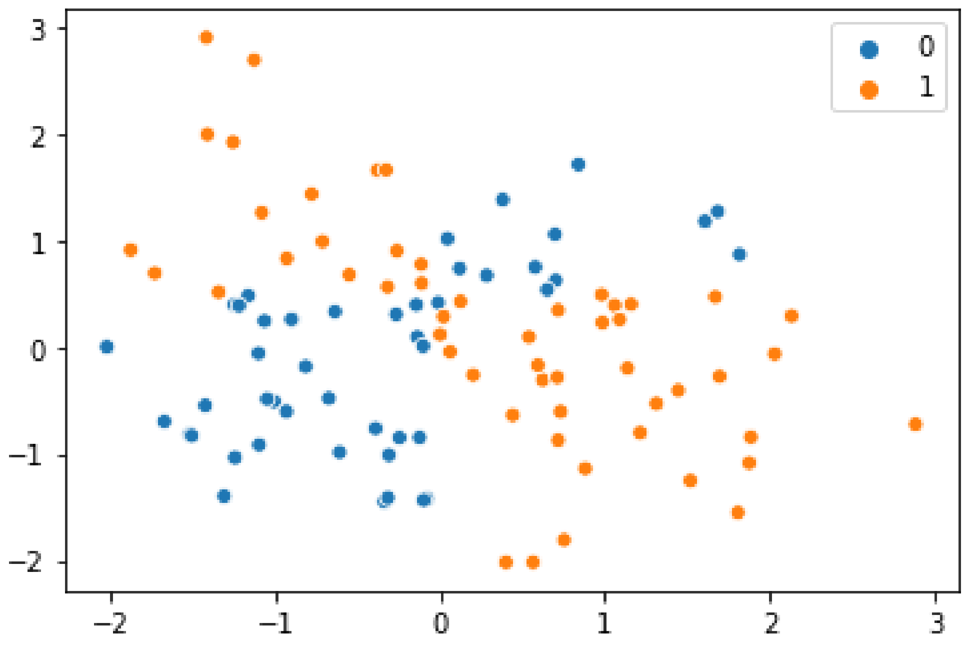
Hu_Anqi_HW6

March 8, 2020

[1]:
```
from sklearn.datasets import make_classification, make_moons
from sklearn import svm, metrics
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split, cross_val_score,␣
 ↪GridSearchCV
from sklearn.datasets import make_blobs
import numpy as np
import pandas as pd
import math
from mlxtend.plotting import plot_decision_regions
import seaborn as sns
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
np.random.RandomState(90210)
x = np.random.randn(100, 2)
y = np.logical_xor(x[:, 0] > 0, x[:, 1] > 0.5)
y = np.array([int(i) for i in y])
sns.scatterplot(x[:, 0], x[:, 1], hue=y);
```



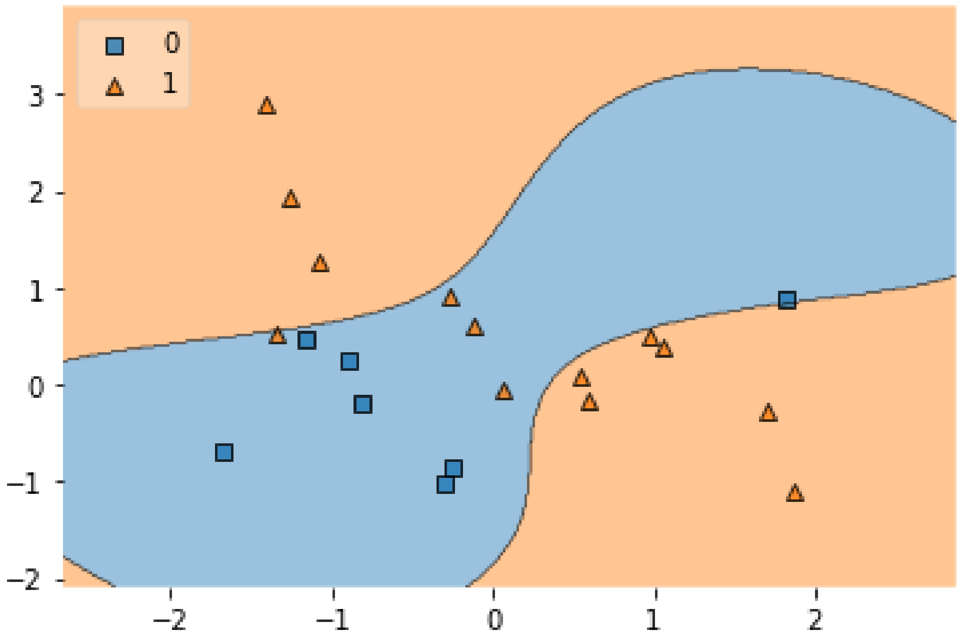
```
[3]: train_x, test_x, train_y, test_y = train_test_split(x, y, test_size=0.2)
```

```
[4]: rbf = svm.SVC(kernel='rbf').fit(train_x, train_y)
     linear = svm.SVC(kernel='linear').fit(train_x, train_y)

     pred1 = rbf.predict(train_x)
     pred2 = linear.predict(train_x)

     err1 = 1 - metrics.accuracy_score(train_y, pred1)
     err2 = 1 - metrics.accuracy_score(train_y, pred2)

     print('The training error of radial kernel is: ', err1)
     print('The training error of linear kernel is: ', err2)

     pred1 = rbf.predict(test_x)
     pred2 = linear.predict(test_x)

     err1 = 1 - metrics.accuracy_score(test_y, pred1)
     err2 = 1 - metrics.accuracy_score(test_y, pred2)

     print('The testing error of radial kernel is: ', err1)
     print('The testing error of linear kernel is: ', err2)
```

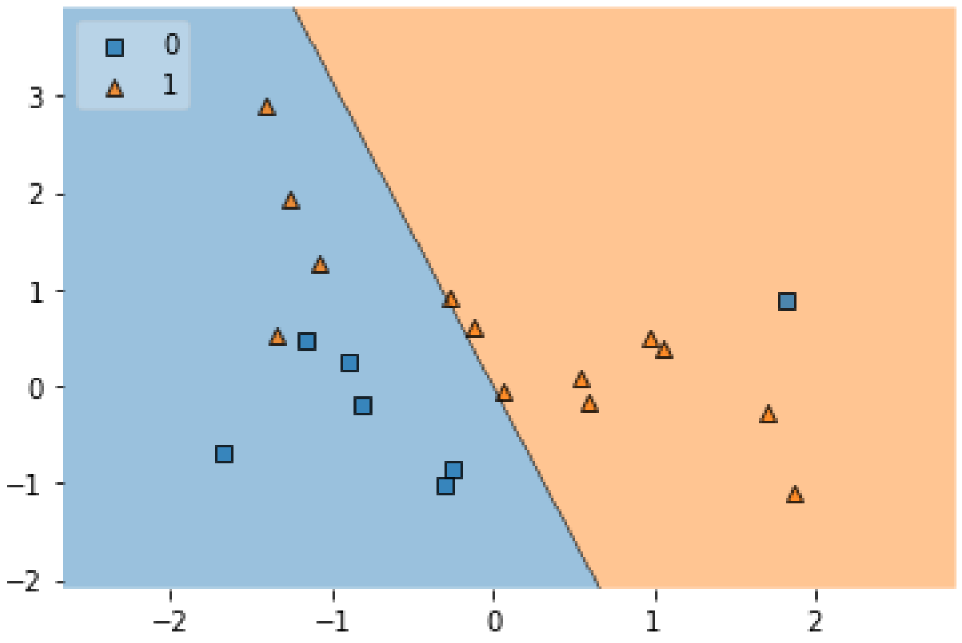
```
The training error of radial kernel is:  0.08750000000000002
The training error of linear kernel is:  0.2875
The testing error of radial kernel is:  0.19999999999999996
The testing error of linear kernel is:  0.25
```

From the training error, it seems like radial kernel is performing better than linear kernel. The same applies to the testing errors. The difference in testing errors is even more pronounced, as the testing error was low.

```
[5]: plot_decision_regions(X=test_x, y=test_y, clf=rbf, legend=2);
```

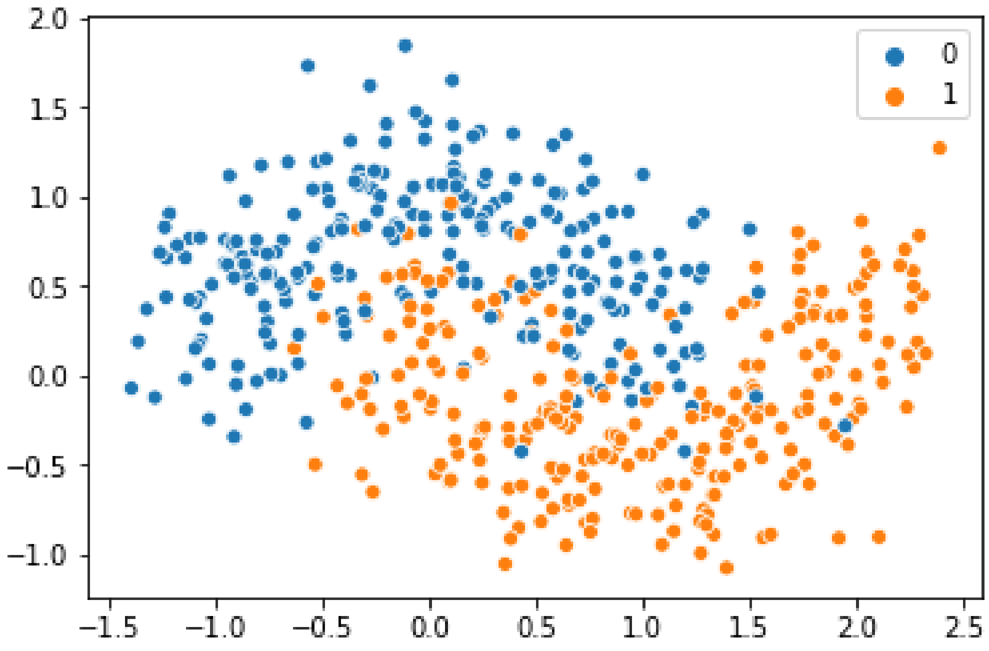


```
[6]: plot_decision_regions(X=test_x, y=test_y, clf=linear, legend=2);
```

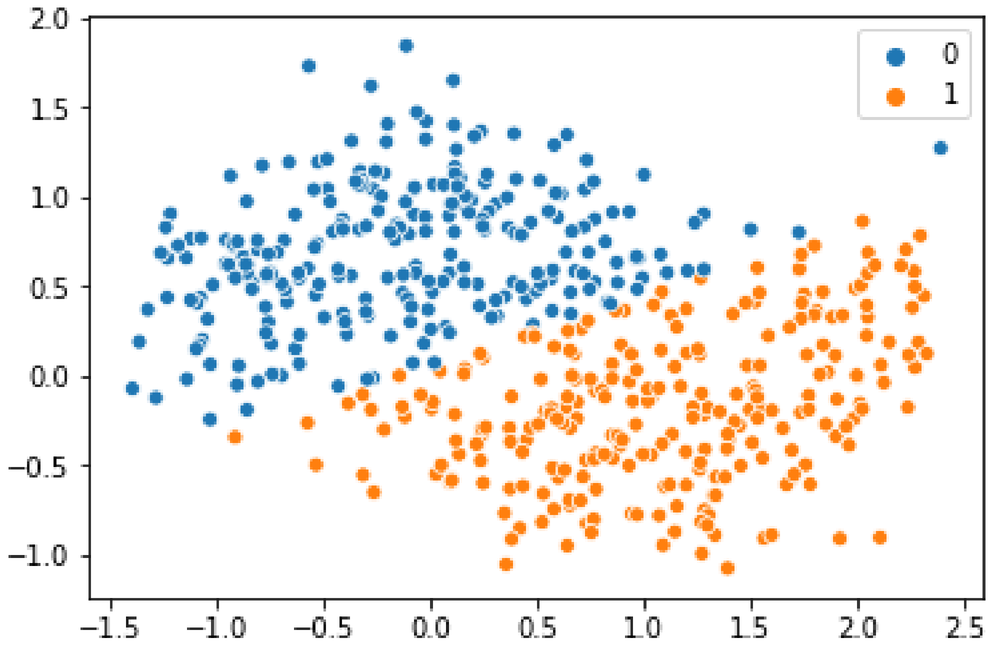


Seeing the decision boundaries with both radial and linear kernels, we can tell that radial kernel is performing a lot better separating the test data into the correct classes. This confirmed the conclusion that radial kernel does a better job when the data is not linearly separable.

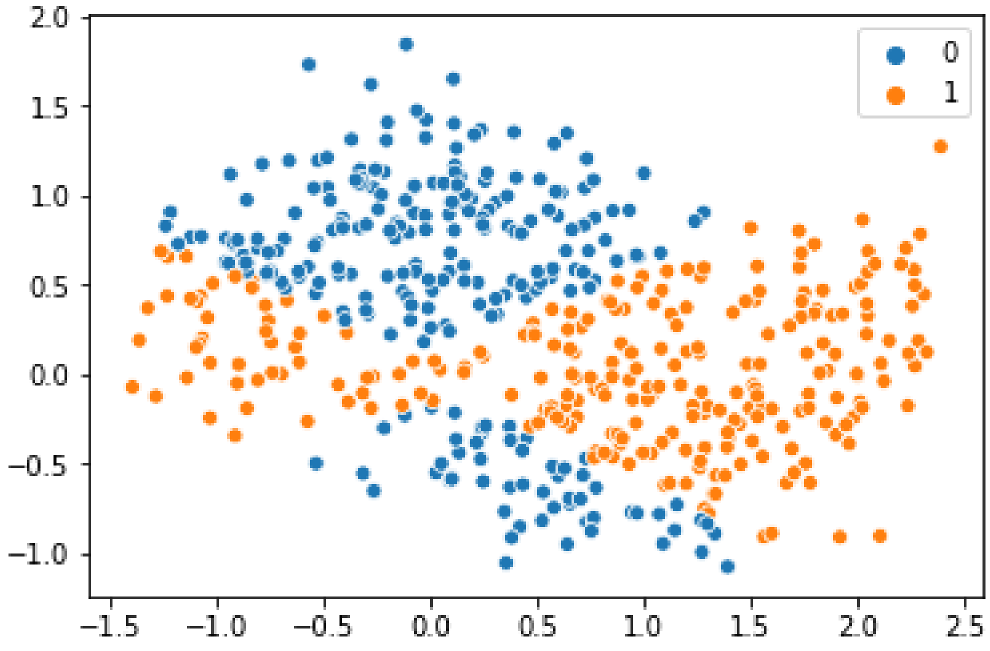
[7]:
```
# part 2
x, y = make_moons(n_samples=500, noise=0.3)
sns.scatterplot(x[:, 0], x[:, 1], hue=y);
```



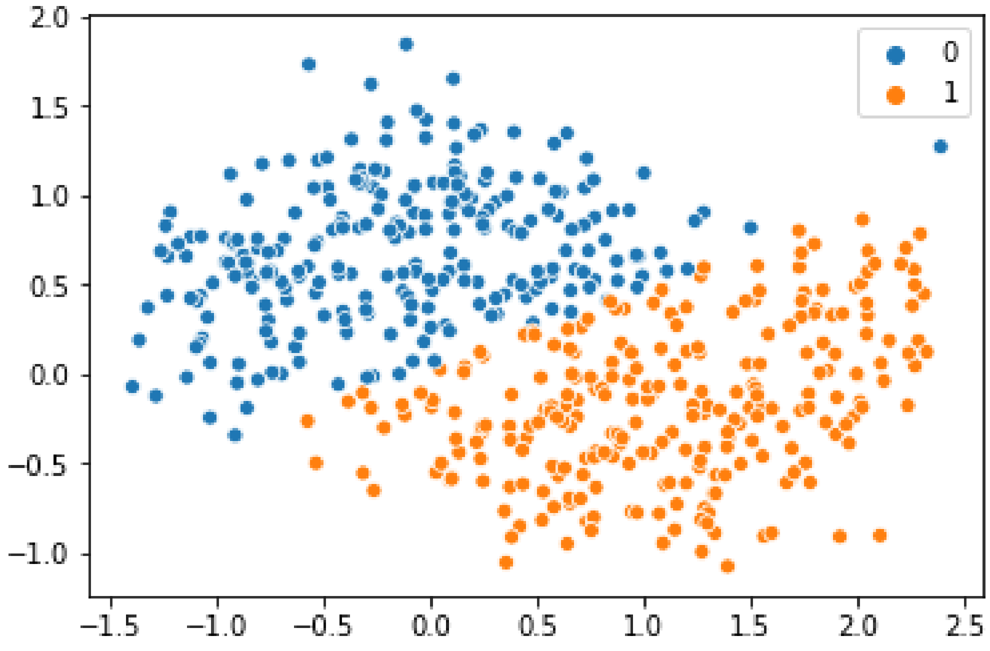
```
[8]: # linear logit
     logit = LogisticRegression().fit(x, y)
     pred = logit.predict(x)
     sns.scatterplot(x[:, 0], x[:, 1], hue=pred);
```



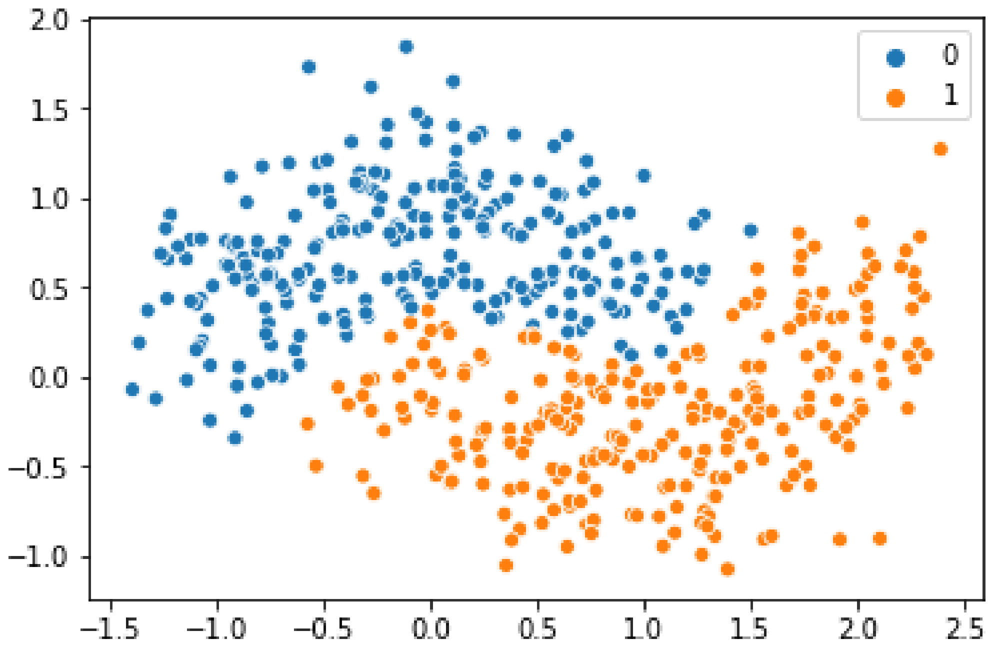
```
[9]: # non-linear logit
     x_new = np.array([x[:, 0] ** 2, x[:, 1] ** 2])
     x_new = x_new.T
     logit = LogisticRegression().fit(x_new, y)
     pred = logit.predict(x_new)
     sns.scatterplot(x[:, 0], x[:, 1], hue=pred);
```



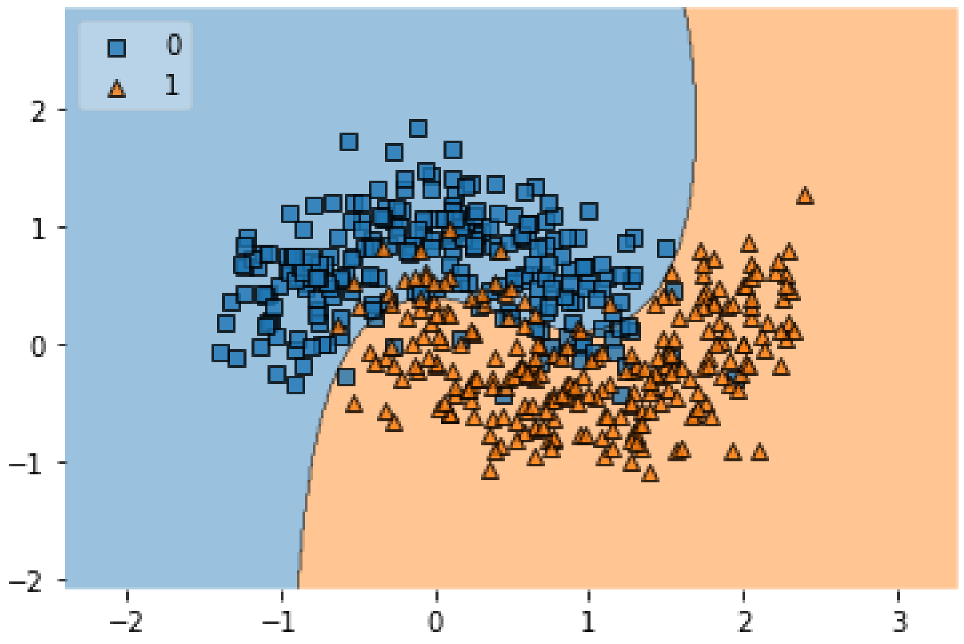
```
[10]: # linear SVC
      linear = svm.SVC(kernel='linear').fit(x, y)
      pred = linear.predict(x)
      sns.scatterplot(x[:, 0], x[:, 1], hue=pred);
```



[11]:
```
# non-linear SVC
rbf = svm.SVC(kernel='rbf').fit(x, y)
pred = rbf.predict(x)
sns.scatterplot(x[:, 0], x[:, 1], hue=pred);
```



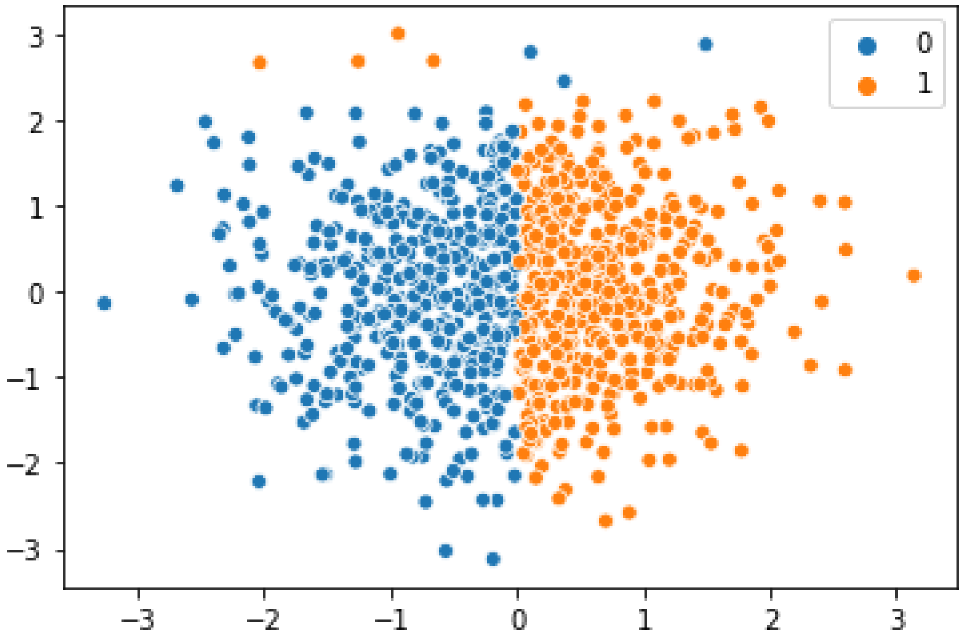
[12]:
```
plot_decision_regions(X=x, y=y, clf=rbf, legend=2);
```



Obviously, using logistic regression for non-linear decision boundaries is not very effective, compared to a non-linear SVM. The decision boundaries for both linear logistic regression and linear support vector classifier are highly similar. Logistic regression captures the statistical relationship between the variables better, whereas SVM focuses on the geometric patterns and is thus better when the data points are less structured and overlapping.

0.0.1 Tuning cost

[13]:
```
x = np.random.randn(900, 2)
y = np.logical_xor(x[:, 0] > 0, x[:, 1] > 2.3)
y = np.array([int(i) for i in y])
sns.scatterplot(x[:, 0], x[:, 1], hue=y);
```



[14]:
```
# SVC, tuning cost, training error
c = [1, 10, 100, 1000]
num_err = []
tr_err = []

for value in c:
    linear = svm.SVC(kernel='linear', C=value).fit(x, y)
    pred = linear.predict(x)
    num_train_err = len(y) - np.sum(pred==y)
    num_err.append(num_train_err)
    train_err = 1 - np.mean(cross_val_score(linear, x, y, cv=10))
    tr_err.append(train_err)
```

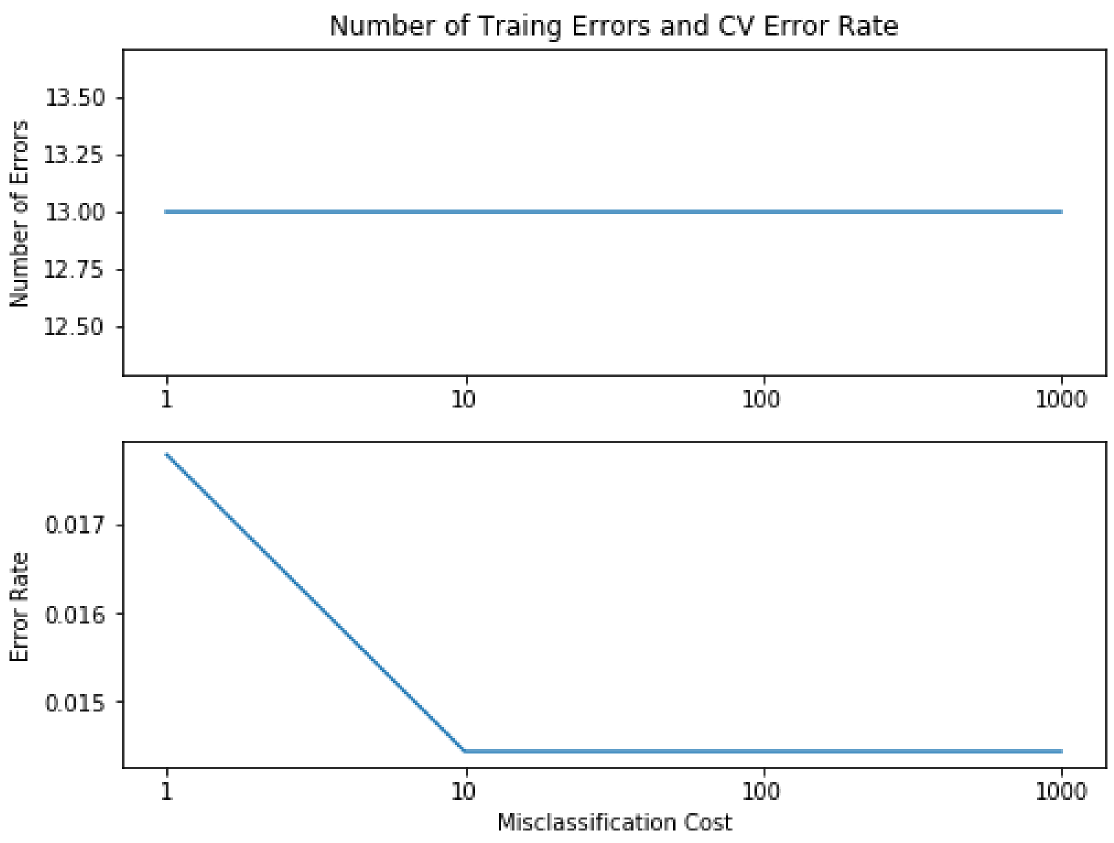
[15]:
```
num_err
```

[15]:
```
[13, 13, 13, 13]
```

[16]:
```
tr_err
```

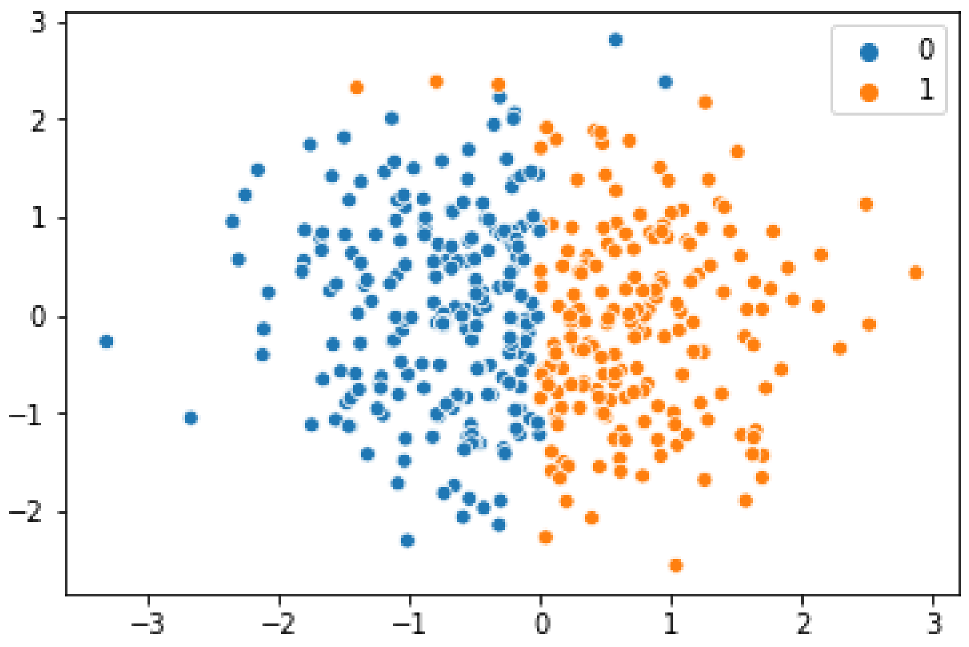
[16]:
```
[0.017779149689262086,
 0.014433331961421714,
 0.014433331961421714,
 0.014433331961421714]
```

[17]:
```
fig, axes = plt.subplots(2, figsize=(8, 6))
axes[0].set_title('Number of Traing Errors and CV Error Rate')
axes[0].plot(num_err)
axes[0].set_xticks([0, 1, 2, 3])
axes[0].set_xticklabels(c)
axes[0].set_ylabel('Number of Errors')
axes[1].plot(tr_err)
axes[1].set_xticks([0, 1, 2, 3])
axes[1].set_xticklabels(c)
axes[1].set_xlabel('Misclassification Cost')
axes[1].set_ylabel('Error Rate');
```



From the graph above we can see that the numbers of errors in the training data are the same across all costs. The overall error rate is the highest when the cost is 1. The cross-validated errors do not seem to be directly related to the number of training errors, though.

[18]:
```
# test data
test_x = np.random.randn(400, 2)
test_y = np.logical_xor(test_x[:, 0] > 0, test_x[:, 1] > 2.3)
test_y = np.array([int(i) for i in test_y])
sns.scatterplot(test_x[:, 0], test_x[:, 1], hue=test_y);
```



```
[19]: # SVC, tuning cost, testing error
      c = [1, 10, 100, 1000]
      num_err = []
      te_err = []

      for value in c:
          linear = svm.SVC(kernel='linear', C=value).fit(x, y)
          pred = linear.predict(test_x)
          num_test_err = len(test_y) - np.sum(pred==test_y)
          num_err.append(num_test_err)
          test_err = 1 - np.mean(cross_val_score(linear, test_x, test_y, cv=10))
          te_err.append(test_err)
```

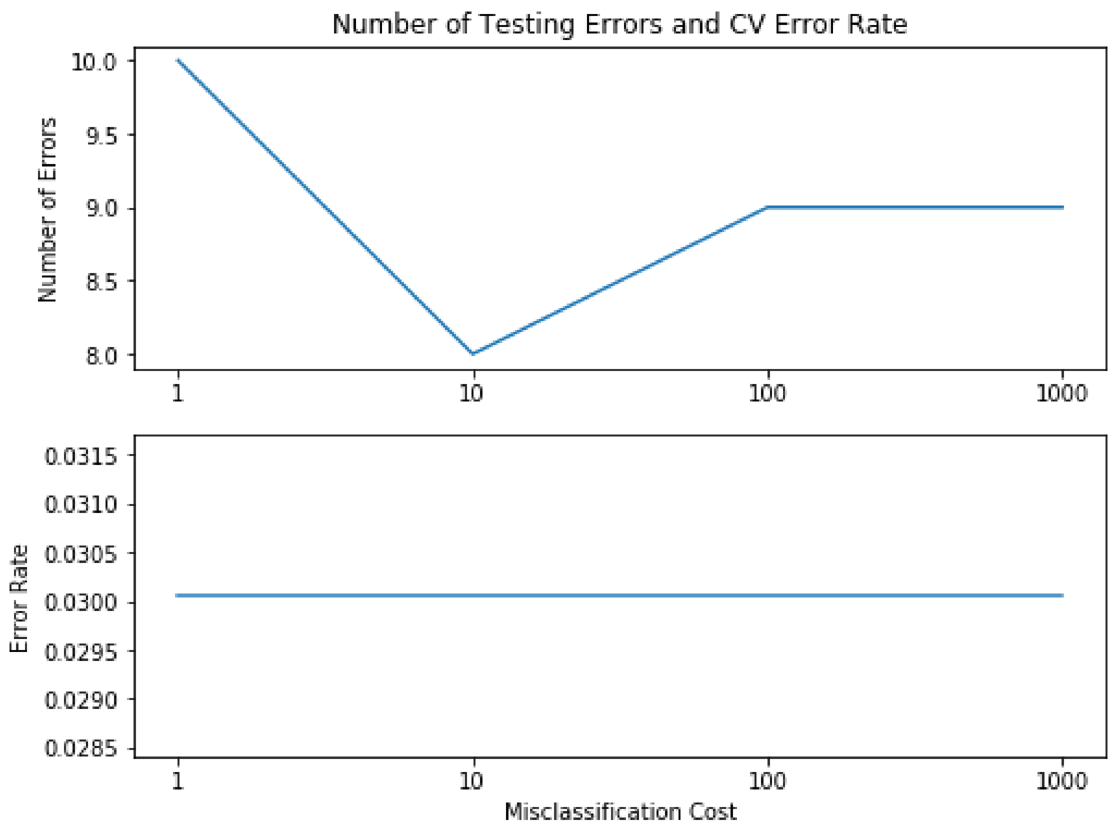
```
[20]: num_err
```

```
[20]: [10, 8, 9, 9]
```

```
[21]: te_err
```

```
[21]: [0.030064102564102635,
       0.030064102564102635,
       0.030064102564102635,
       0.030064102564102635]
```

[22]:
```
fig, axes = plt.subplots(2, figsize=(8, 6))
axes[0].set_title('Number of Testing Errors and CV Error Rate')
axes[0].plot(num_err)
axes[0].set_xticks([0, 1, 2, 3])
axes[0].set_xticklabels(c)
axes[0].set_ylabel('Number of Errors')
axes[1].plot(te_err)
axes[1].set_xticks([0, 1, 2, 3])
axes[1].set_xticklabels(c)
axes[1].set_xlabel('Misclassification Cost')
axes[1].set_ylabel('Error Rate');
```



In the testing data, the number of test errors is the lowest when cost is 10. Relatively speaking, on the other hand, the error rates are the same across all four cost values. Judging by the number of errors, 10 is the best performing cost. Different from the case with training errors, the best performing cost should be 10.

0.0.2 Applications

[23]:
```
train = pd.read_csv('data/gss_train.csv')
test = pd.read_csv('data/gss_test.csv')
```

[24]:
```
train_x = train.loc[:, train.columns != 'colrac']
train_y = train['colrac']

test_x = test.loc[:, test.columns != 'colrac']
test_y = test['colrac']
```

[25]:
```
# SVC
c = [1, 10, 100, 1000]
te_error = {}

for value in c:
    svc = svm.SVC(kernel='linear', C=value).fit(train_x, train_y)
    test_err = 1 - np.mean(cross_val_score(svc, test_x, test_y, cv=10))
    te_error[value] = test_err

te_error
```

[25]:
```
{1: 0.23180612244897958,
 10: 0.23809863945578225,
 100: 0.23801530612244892,
 1000: 0.23805612244897956}
```

For the linear kernel, the performances of the models with four different cost levels are very similar to one another. However, it seems that the CV error is the lowest when the cost is 1.

[26]:
```
# non-linear SVM - radial
param_grid = {'gamma': [0.1, 1, 10, 100],
              'C': [1, 10, 100, 1000]}

rbf = svm.SVC(kernel='rbf').fit(train_x, train_y)
rbf_grid = GridSearchCV(rbf, param_grid=param_grid, cv=10)

fit = rbf_grid.fit(test_x, test_y)

rbf_score = fit.best_score_
rbf_best = fit.best_estimator_
```

[27]:
```
# non-linear SVM - polynomial
param_grid = {'gamma': [0.1, 1, 10, 100],
              'degree': [2, 3, 4, 5],
              'C': [1, 10, 100, 1000]}
```

```
poly = svm.SVC(kernel='poly').fit(train_x, train_y)
grid = GridSearchCV(poly, param_grid=param_grid, cv=10)

fit = grid.fit(test_x, test_y)

poly_score = fit.best_score_

poly_best = fit.best_estimator_
```

[28]:
```
print(rbf_score, poly_score)
```

```
0.537525354969574 0.7342799188640974
```

[29]:
```
rbf_best
```

[29]:
```
SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma=0.1, kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

[30]:
```
poly_best
```

[30]:
```
SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=4, gamma=0.1, kernel='poly',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

With the best performance scores of radial and polynomial kernels, the radial kernel is performing better when fitting the test data. Looking further at the best estimators using both kernels, both optimize when the cost is 1 and gamma is 0.1. C measures the cost of misclassification. A C at value 1 means that the cost of misclassifying data points is fairly low. Since gamma represents the extent to which the model is trying to fit the training data, the optimal gamma being 0.1 indicates that the training data had large influence on the training fitting. Degree was only tuned for the polynomial kernel because it is not applicable to the radial kernel. The optimal degree for the polynomial kernel is 4, meaning that it was the fourth degree polynomial that was used to find the hyperplane to split the data.

[ ]: