HW6

March 8, 2020

[1]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import tqdm
```

[121]:
```
# models
from sklearn.svm import SVC, SVR
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split, GridSearchCV,␣
↪cross_val_score
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline
from sklearn import preprocessing
```

[17]:
```
np.random.seed(2019)
```

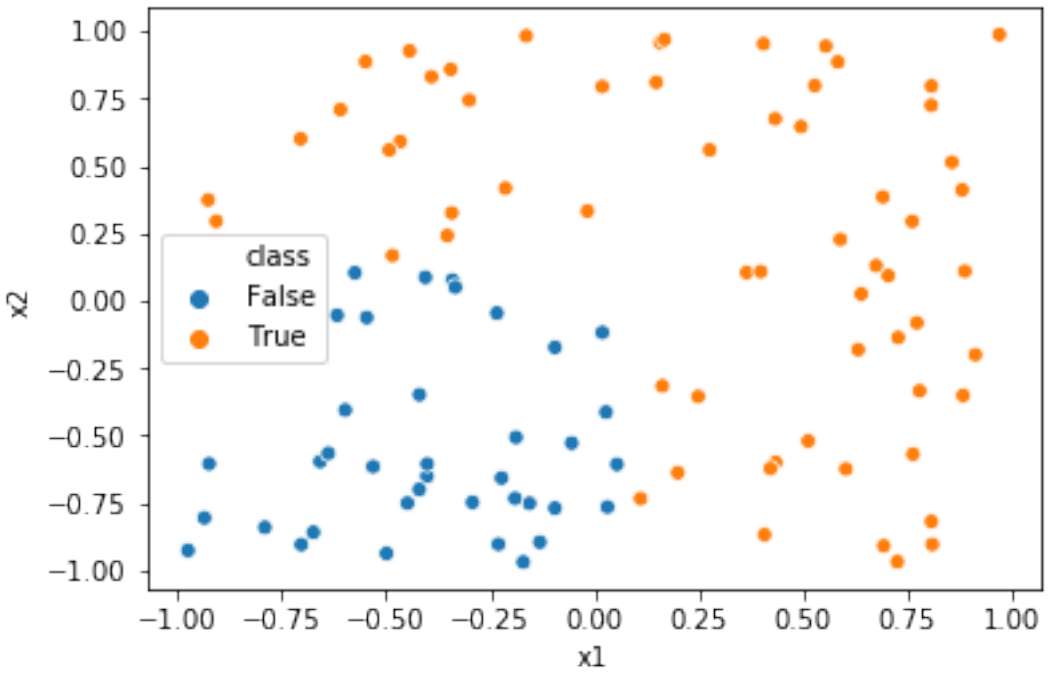
1 Conceptual exercises

1.1 Non-linear separation

[18]:
```
# generate random dataset
x1=np.random.uniform(-1, 1, 100)
x2=np.random.uniform(-1, 1, 100)
err=np.random.normal(0, 0.1, 100)
y= x1+x1**2+x2+x2**2+err
y_class=y>0

total_df=pd.DataFrame({'y': y, "x1":x1, "x2":x2, "error":err, "class":y_class})
```

[19]:
```
# visualize the dataset
sns.scatterplot(x='x1', y="x2", hue="class", data=total_df)
```

[19]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a22c94ed0>
```



[21]:
```
# test train split
x_train, x_test, y_train, y_test = train_test_split(total_df[['x1', 'x2']],␣
 ↪total_df['class'], train_size=0.7)
```

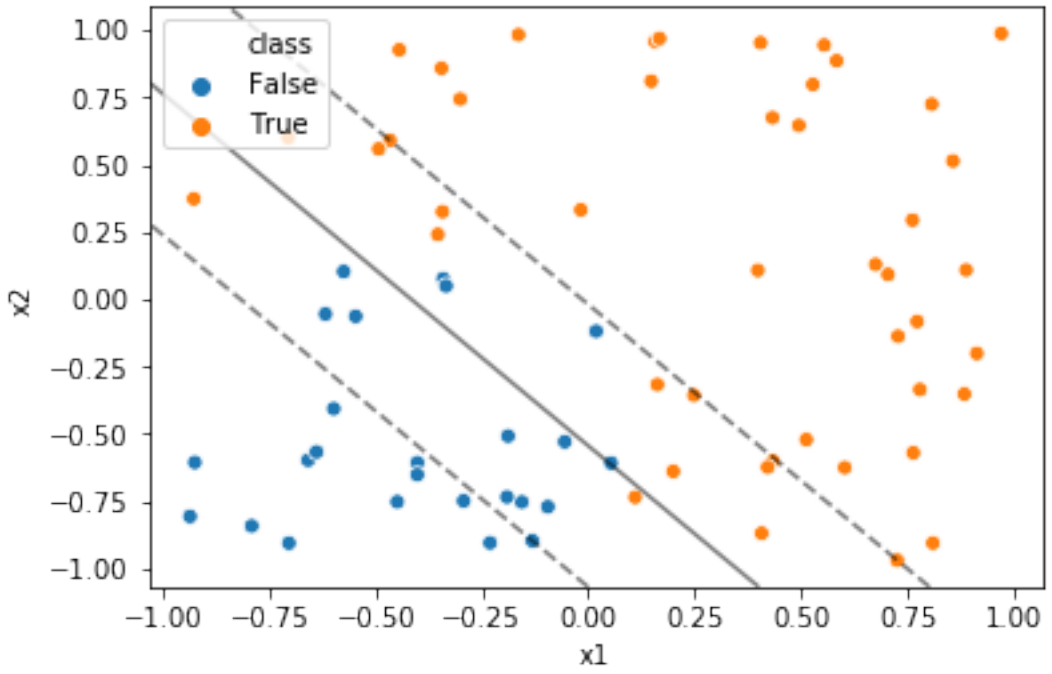
[33]:
```
# train set comparision
svm_linear= SVC(kernel='linear')
svm_radial= SVC(kernel= 'rbf', gamma='scale')

svm_linear.fit(x_train, y_train)
svm_radial.fit(x_train, y_train)

print("Training set error rate (SVM Linear): ", (1-svm_linear.score(x_train,␣
 ↪y_train)))
print("Training set error rate (SVM Radial): ", (1-svm_radial.score(x_train,␣
 ↪y_train)))
```

```
Training set error rate (SVM Linear):  0.08571428571428574
Training set error rate (SVM Radial):  0.02857142857142858
```

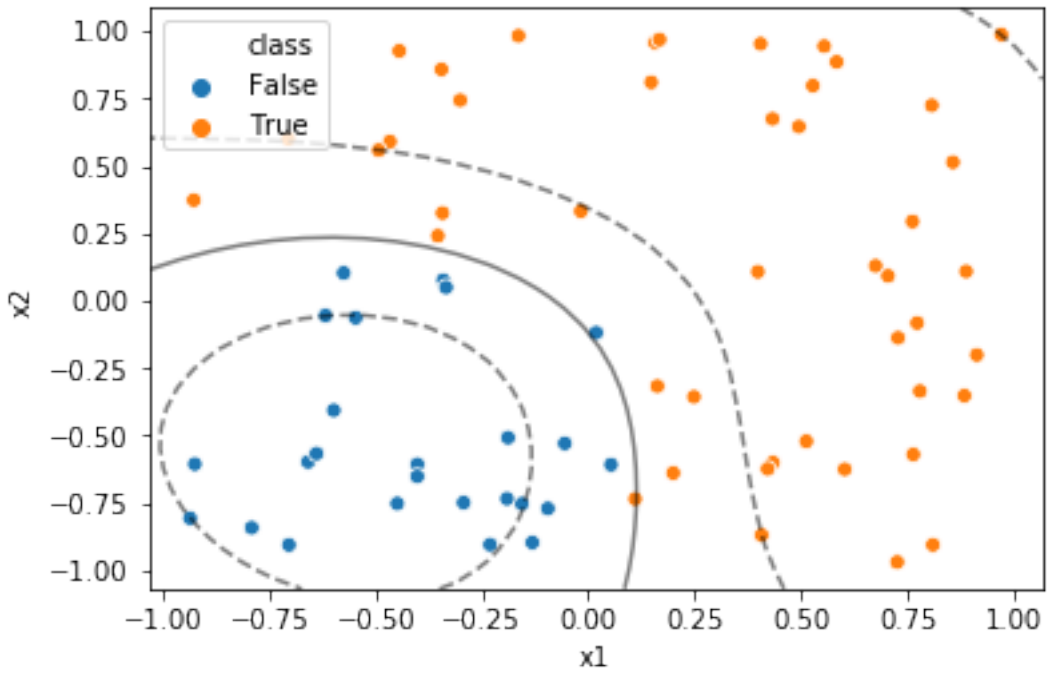
[56]:
```
# draw training set classification
def draw_svm_result(clf, x, y):
    sns.scatterplot(x=x.x1, y=x.x2, hue=y)

    # create grid to evaluate model
```

```
    ax = plt.gca()
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()
    xx = np.linspace(xlim[0], xlim[1], 30)
    yy = np.linspace(ylim[0], ylim[1], 30)
    YY, XX = np.meshgrid(yy, xx)
    xy = np.vstack([XX.ravel(), YY.ravel()]).T
    Z = clf.decision_function(xy).reshape(XX.shape)

    # plot decision boundary and margins
    ax.contour(XX, YY, Z, colors='k', levels=[-1, 0, 1], alpha=0.5,
           linestyles=['--', '-', '--'])

    # plot support vectors
    #ax.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1], s=100,
    #       linewidth=1, facecolors='none', edgecolors='k')

    plt.show()
```

[57]:
```
draw_svm_result(svm_linear, x_train, y_train)
```



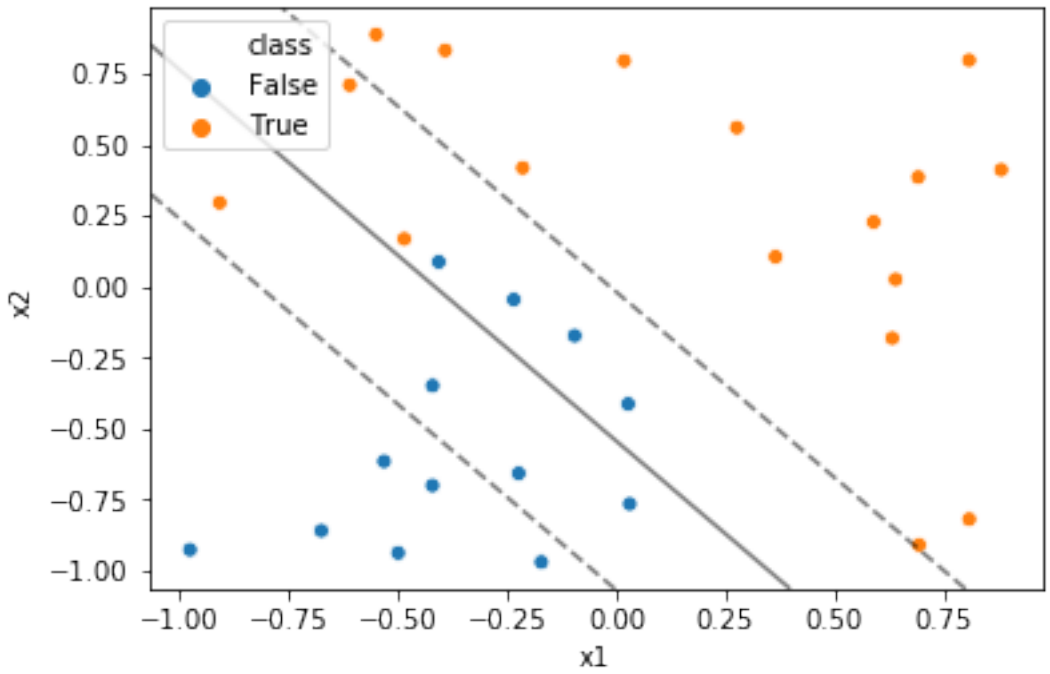
[58]:
```
draw_svm_result(svm_radial, x_train, y_train)
```



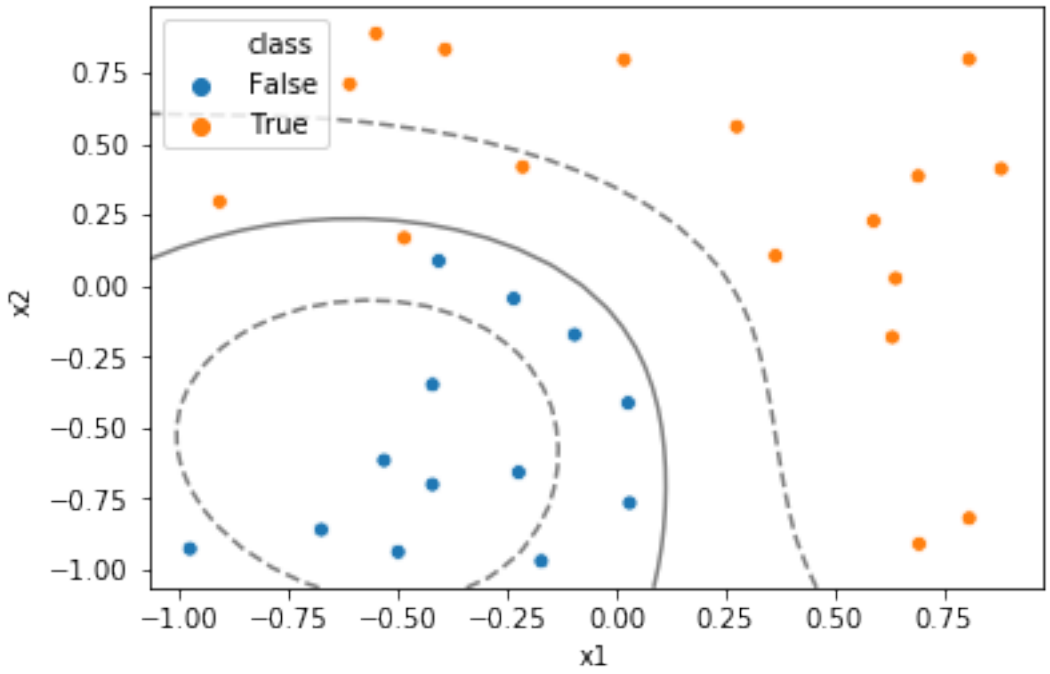
[44]:
```
# test set comparision
print("Test set error rate (SVM Linear): ", (1-svm_linear.score(x_test,␣
 ↪y_test)))
print("Test set error rate (SVM Radial): ", (1-svm_radial.score(x_test,␣
 ↪y_test)))
```

```
Test set error rate (SVM Linear):  0.16666666666666663
Test set error rate (SVM Radial):  0.033333333333333326
```

[59]:
```
draw_svm_result(svm_linear, x_test, y_test)
```



[60]:
```
draw_svm_result(svm_radial, x_test, y_test)
```



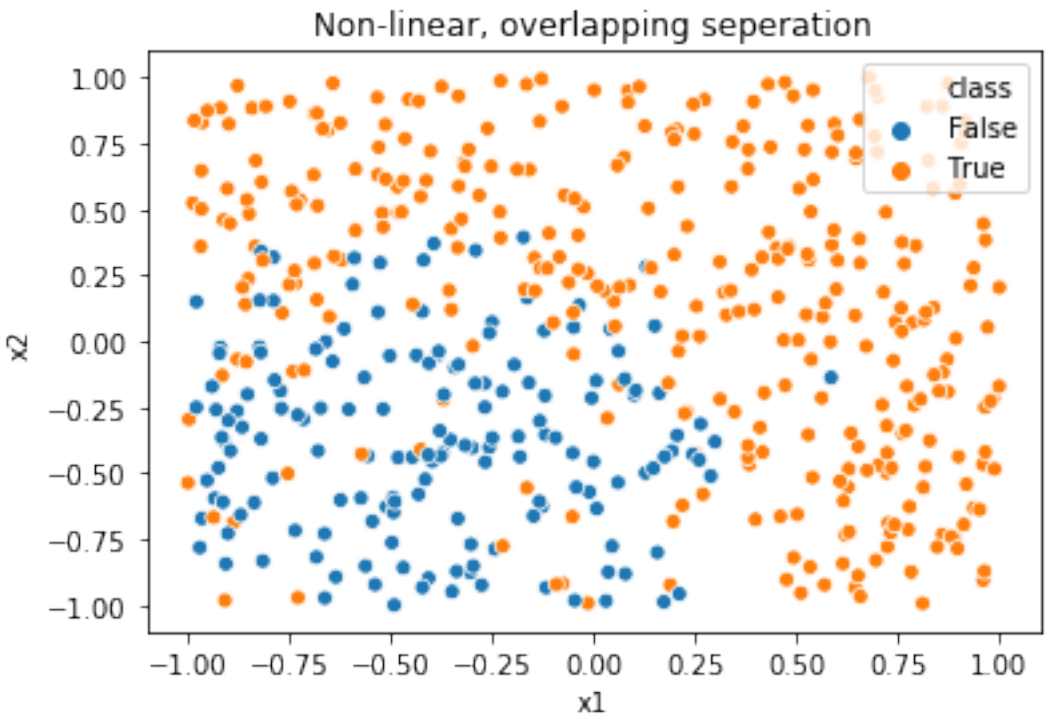
Based on the test error and decision boundary visualization, we could find that SVM with radial kernel outperforms SVC with linear kernel. Moreover, the SVM radial is also better when applying to test set. The test set error rate of SVM radial is much lower than Svm with linear kernel. This conclusion is valid as the data is generated to have a non-linear decsion boundary.

1.2 SVM vs. logistic regression

[65]:
```
# generate random dataset
x1=np.random.uniform(-1, 1, 500)
x2=np.random.uniform(-1, 1, 500)
err=np.random.normal(0, 0.3, 500)
y= x1+x1**2+x2+x2**2+err
y_class=y>0

total_df_2=pd.DataFrame({'y': y, "x1":x1, "x2":x2, "error":err, "class":
 ↪y_class})
```

[67]:
```
#plot the data
sns.scatterplot(x='x1', y="x2", hue="class", data=total_df_2)
plt.title("Non-linear, overlapping seperation")
```

[67]:
```
Text(0.5,1,'Non-linear, overlapping seperation')
```

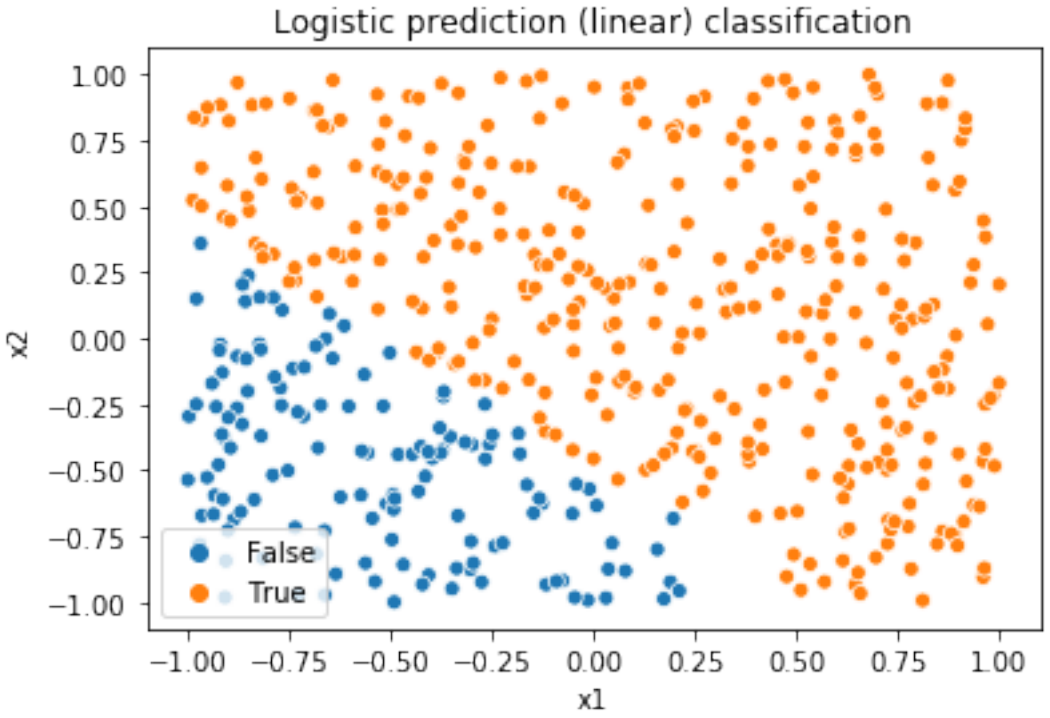


[69]:
```
# logistic regression
log=LogisticRegression(solver ='lbfgs')
log.fit(total_df_2[['x1','x2']], total_df_2['class'])
```

[69]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

[71]:
```
# draw linear classification
y_pred=log.predict(total_df_2[['x1','x2']])
sns.scatterplot(total_df_2.x1, total_df_2.x2, hue=y_pred)
plt.title("Logistic prediction (linear) classification")
```

[71]:
```
Text(0.5,1,'Logistic prediction (linear) classification')
```

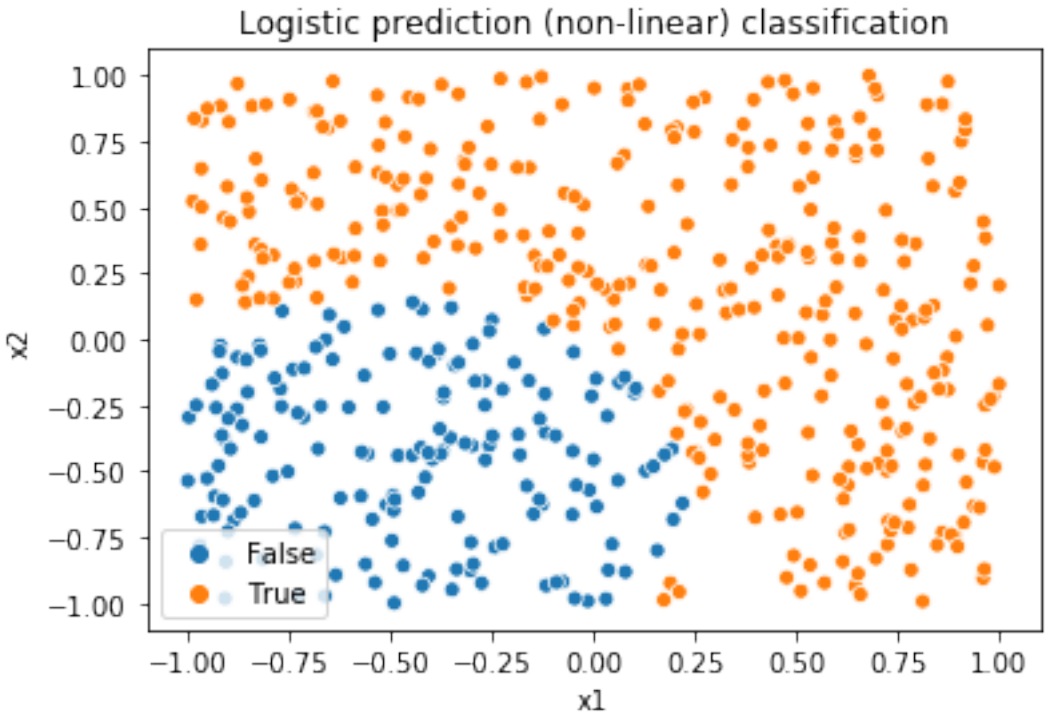


[74]:
```
# non linear logistic regression
log_poly = Pipeline([('poly',PolynomialFeatures(degree = 2)),
                     ('log',LogisticRegression(solver ='lbfgs'))])

log_poly.fit(total_df_2[['x1','x2']], total_df_2['class'])
```

[74]:
```
Pipeline(memory=None,
         steps=[('poly',
                 PolynomialFeatures(degree=2, include_bias=True,
                                    interaction_only=False, order='C')),
                ('log',
                 LogisticRegression(C=1.0, class_weight=None, dual=False,
                                    fit_intercept=True, intercept_scaling=1,
                                    l1_ratio=None, max_iter=100,
                                    multi_class='warn', n_jobs=None,
                                    penalty='l2', random_state=None,
                                    solver='lbfgs', tol=0.0001, verbose=0,
                                    warm_start=False))],
         verbose=False)
```

[75]:
```
# draw non-linear classification
y_pred_poly=log_poly.predict(total_df_2[['x1','x2']])
sns.scatterplot(total_df_2.x1, total_df_2.x2, hue=y_pred_poly)
plt.title("Logistic prediction (non-linear) classification")
```

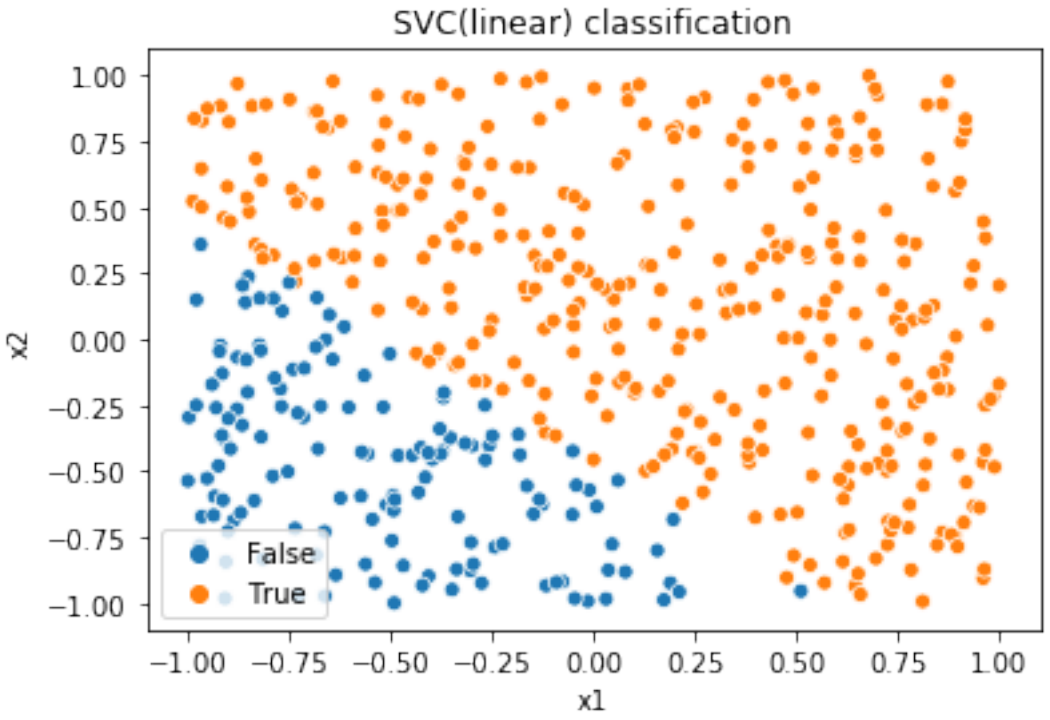
[75]:
```
Text(0.5,1,'Logistic prediction (non-linear) classification')
```



[76]:
```
# support vector classifier (linear kernel)
svm_linear.fit(total_df_2[['x1','x2']], total_df_2['class'])
```

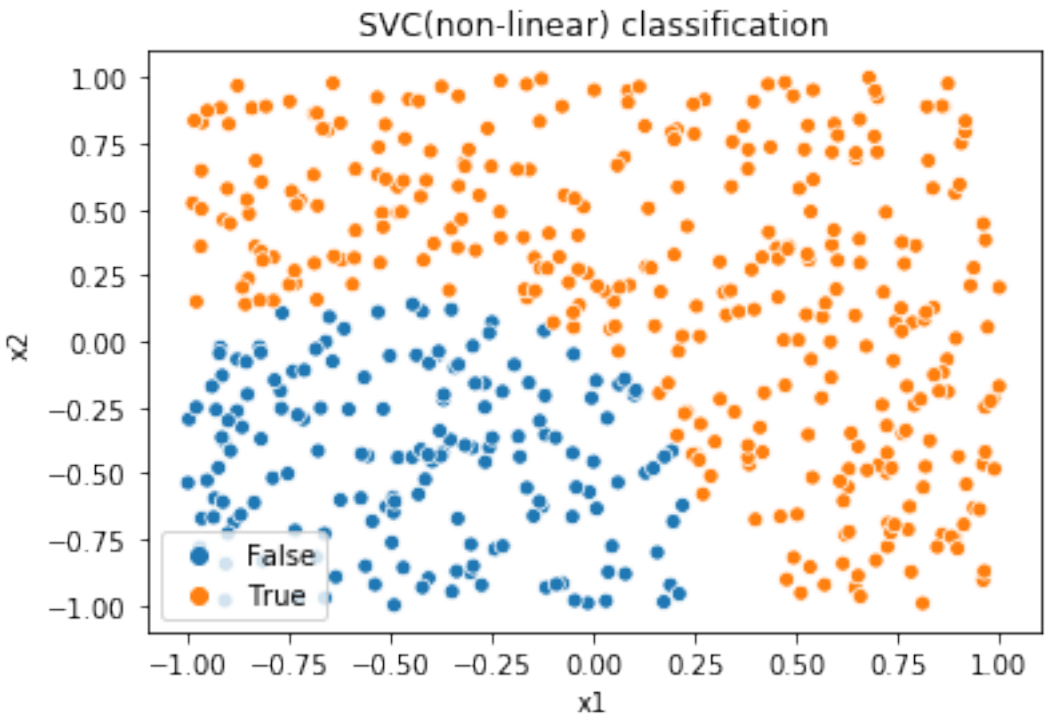
```
y_pred_svc_l=svm_linear.predict(total_df_2[['x1','x2']])
sns.scatterplot(total_df_2.x1, total_df_2.x2, hue=y_pred_svc_l)
plt.title("SVC(linear) classification")
```

[76]: Text(0.5,1,'SVC(linear) classification')



[77]:
```
# SVM using a non-linear kernel
svm_radial.fit(total_df_2[['x1','x2']], total_df_2['class'])
y_pred_svc_R=svm_radial.predict(total_df_2[['x1','x2']])
sns.scatterplot(total_df_2.x1, total_df_2.x2, hue=y_pred_svc_R)
plt.title("SVC(non-linear) classification")
```

[77]: Text(0.5,1,'SVC(non-linear) classification')



[78]:
```
# test set comparision
print("Error rate (Logistic regression Linear): ", (1-log.
 ↪score(total_df_2[['x1','x2']], total_df_2['class'])))
print("Error rate (Logistic regression non-linear): ", (1-log_poly.
 ↪score(total_df_2[['x1','x2']], total_df_2['class'])))
print("Error rate (SVM Linear): ", (1-svm_linear.score(total_df_2[['x1','x2']],␣
 ↪total_df_2['class'])))
print("Error rate (SVM non-linear): ", (1-svm_radial.
 ↪score(total_df_2[['x1','x2']], total_df_2['class'])))
```

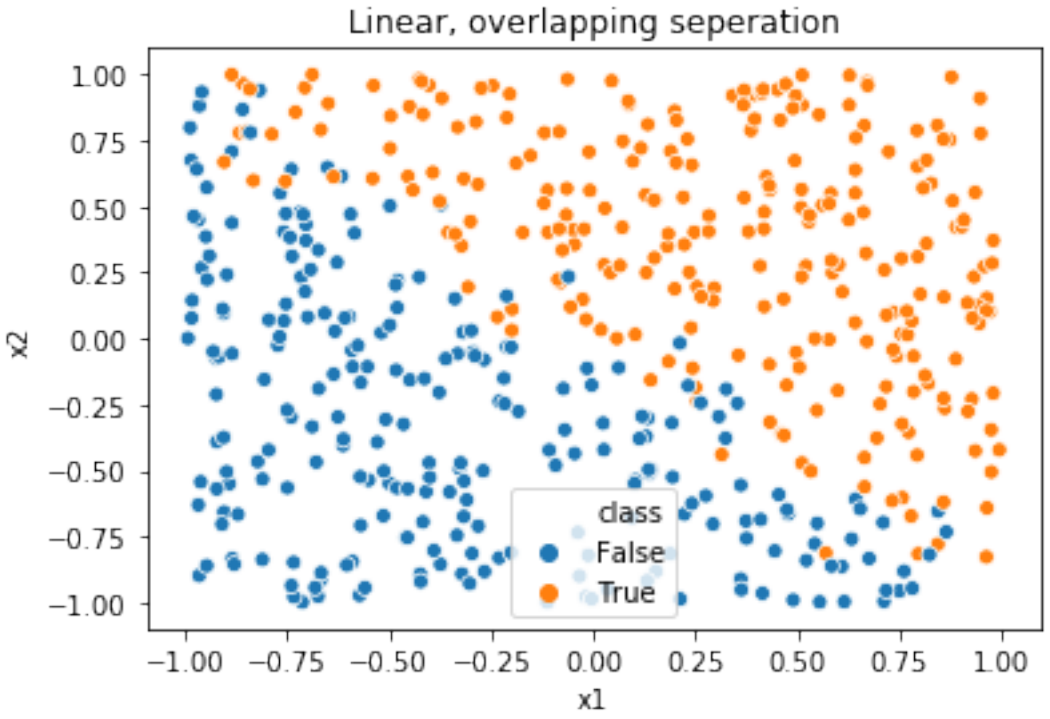
```
Error rate (Logistic regression Linear):  0.17000000000000004
Error rate (Logistic regression non-linear):  0.122
Error rate (SVM Linear):  0.17200000000000004
Error rate (SVM non-linear):  0.12
```

Based on the visualization map and the error rate, we could find that for both Logistic regression and Support vector machine, the non-linear approach performs better than the linear approach. In terms of linear approach, SVM and logistic regression generally have the same performance. However, as for the non-linear approach, SVM with a radial kernel is slightly better than the logistic regression with two degree polynomial features. SVM is also more elegant in terms of non-linear approach: In SVM what we need is just to specify the kernel, but in logistic regression, there might be some uncertainty when building polynomial features.

1.3 Tuning cost

[85]:
```
# generate random dataset
x1=np.random.uniform(-1, 1, 500)
x2=np.random.uniform(-1, 1, 500)
err=np.random.uniform(-0.3, 0.3, 500)
y= x1+x2+err
y_class=y>0

total_df_3=pd.DataFrame({'y': y, "x1":x1, "x2":x2, "class":y_class})

#plot the data
sns.scatterplot(x='x1', y="x2", hue="class", data=total_df_3)
plt.title("Linear, overlapping seperation")
```

[85]:
```
Text(0.5,1,'Linear, overlapping seperation')
```



[86]:
```
# test train split
x_train, x_test, y_train, y_test = train_test_split(total_df_3[['x1', 'x2']],␣
↪total_df_3['class'], train_size=0.7)
```

[107]:
```
costs = [0.01, 0.1, 0.5, 1, 2, 5]
cv_errors=[]
train_errors=[]
```

```
for c in costs:
    clf= SVC(C = c, kernel='linear')
    clf.fit(x_train, y_train)
    cv_err= 1- np.mean(cross_val_score(clf, x_train, y_train, cv=10,␣
 ↪scoring='accuracy'))
    train_err= 1- clf.score(x_train, y_train)
    cv_errors.append(cv_err)
    train_errors.append(train_err)
```

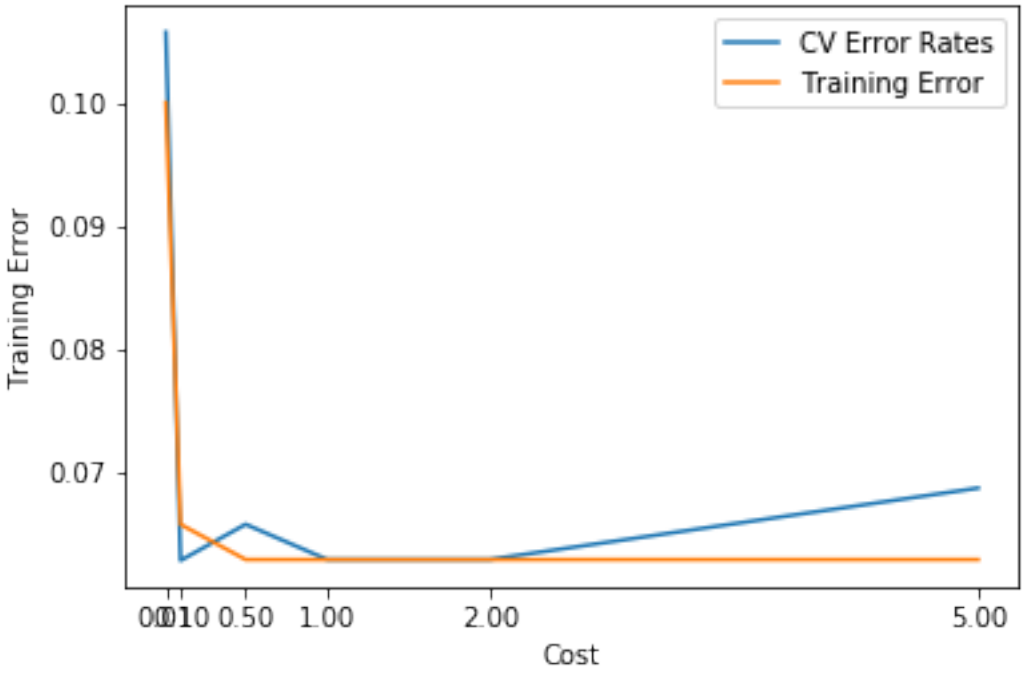
[108]:
```
err_df=pd.DataFrame({"Cost": costs, 'CV Error Rates': cv_errors, "Training␣
 ↪Error": train_errors})
err_df
```

[108]:

	Cost	CV Error Rates	Training Error
0	0.01	0.105728	0.100000
1	0.10	0.062782	0.065714
2	0.50	0.065724	0.062857
3	1.00	0.062866	0.062857
4	2.00	0.062866	0.062857
5	5.00	0.068665	0.062857

[109]:
```
sns.lineplot(x = 'Cost', y='CV Error Rates', data= err_df, label="CV Error␣
 ↪Rates")
sns.lineplot(x = 'Cost', y='Training Error', data= err_df, label="Training␣
 ↪Error")
plt.xticks(costs)
plt.legend()
```

[109]:
```
<matplotlib.legend.Legend at 0x1a2721fa90>
```



Generally, as the cost increases, both the cross validation error rates and the training error rate drops. When the cost reaches certain threshold, the error rates hold still (prbably because at this stage the model made its optimal decsion of classification). The two error rates measures, as could be obeserved from graph and dataframe, is strongly correlated. However, one exception is when cost=0.5, there's a small peak in CV error rates while such peak is not observable in training error.

[111]:
```
costs = [0.01, 0.1, 0.5, 1, 2, 5]
cv_errors=[]
train_errors=[]
test_errors=[]

for c in costs:
    clf= SVC(C = c, kernel='linear')
    clf.fit(x_train, y_train)
    cv_err= 1- np.mean(cross_val_score(clf, x_train, y_train, cv=10,␣
 ↪scoring='accuracy'))
    train_err= 1- clf.score(x_train, y_train)
    test_err= 1- clf.score(x_test, y_test)
    cv_errors.append(cv_err)
    train_errors.append(train_err)
    test_errors.append(test_err)

err_df=pd.DataFrame({"Cost": costs, 'CV Error Rates': cv_errors, "Training␣
 ↪Error": train_errors, "Test Error": test_errors})
```

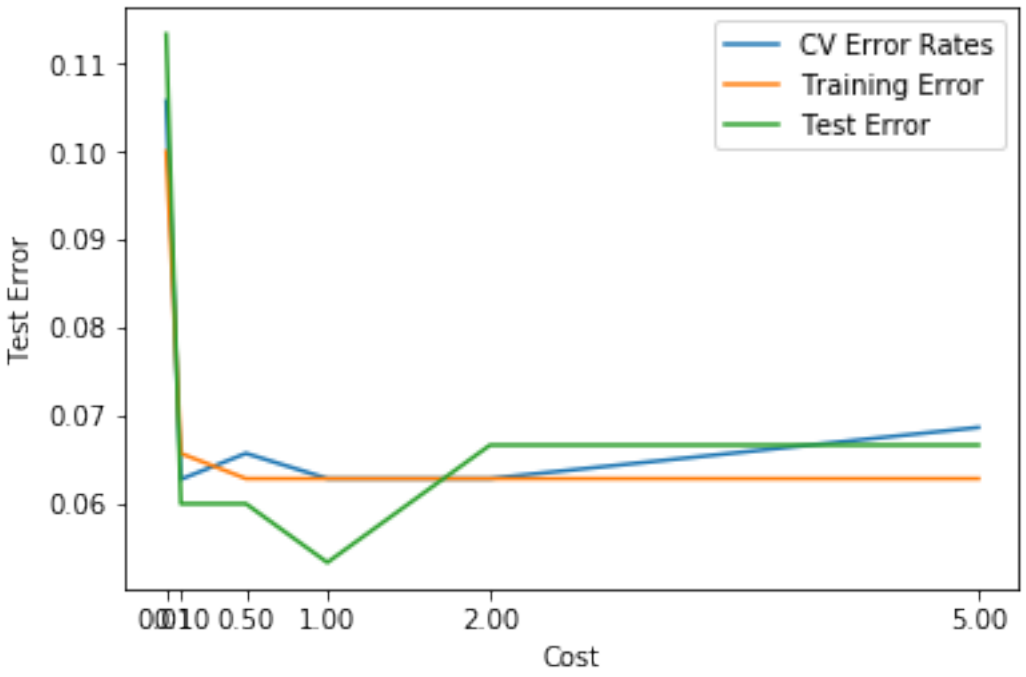
```
err_df
```

[111]:

	Cost	CV Error Rates	Training Error	Test Error
0	0.01	0.105728	0.100000	0.113333
1	0.10	0.062782	0.065714	0.060000
2	0.50	0.065724	0.062857	0.060000
3	1.00	0.062866	0.062857	0.053333
4	2.00	0.062866	0.062857	0.066667
5	5.00	0.068665	0.062857	0.066667

[112]:
```
sns.lineplot(x = 'Cost', y='CV Error Rates', data= err_df, label="CV Error␣
↪Rates")
sns.lineplot(x = 'Cost', y='Training Error', data= err_df, label="Training␣
↪Error")
sns.lineplot(x = 'Cost', y='Test Error', data= err_df, label="Test Error")

plt.xticks(costs)
plt.legend()
```

[112]:
```
<matplotlib.legend.Legend at 0x1a273ee350>
```



When Cost= 1.0 the test error comes to its minimum. For training error, its smallest value is achieved when cost=0.5, for cv error rate, its smallest values is at cost=0.1 and cost=1. Generally, this results show that the optimal cost value should be 1 based on the performance of cross-validation error, training error and test error. As cost increase (>1), the model will likely to be

overfitting (suggested by increasing cv error and test error), and if cost is too small, the model will be underfitting.

2 Application: Predicting attitudes towards racist college professors

[113]:
```
gss_train = pd.read_csv("./data/gss_train.csv")
gss_test=pd.read_csv("./data/gss_test.csv")
x_train =gss_train.drop('colrac', axis=1)
y_train = gss_train.colrac
x_test=gss_test.drop('colrac', axis=1)
y_test = gss_test.colrac
```

[115]:
```
costs = [0.01, 0.1, 0.5, 1, 2, 5, 8, 10]
cv_errors=[]

for c in tqdm.tqdm(costs):
    clf= SVC(C = c, kernel='linear')
    clf.fit(x_train, y_train)
    cv_err= 1- np.mean(cross_val_score(clf, x_train, y_train, cv=10,␣
 ↪scoring='accuracy'))
    cv_errors.append(cv_err)

err_df=pd.DataFrame({"Cost": costs, 'CV Error Rates': cv_errors})
err_df
```

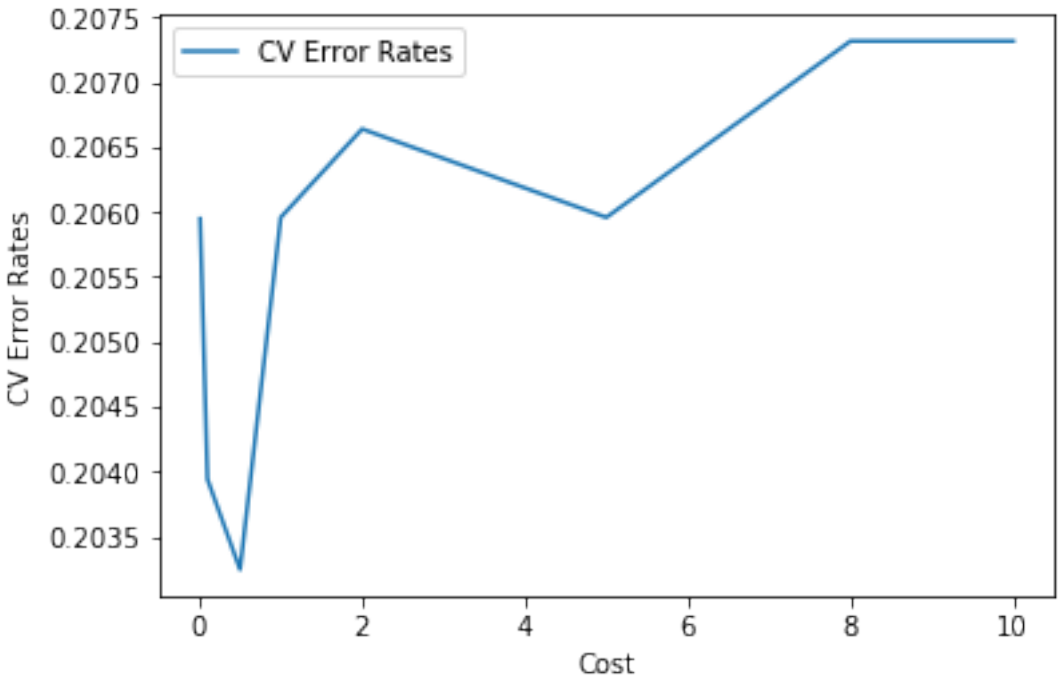
```
100%|      | 8/8 [04:46<00:00, 35.87s/it]
```

[115]:

	Cost	CV Error Rates
0	0.01	0.205949
1	0.10	0.203945
2	0.50	0.203251
3	1.00	0.205958
4	2.00	0.206638
5	5.00	0.205958
6	8.00	0.207314
7	10.00	0.207314

[116]:
```
sns.lineplot(x = 'Cost', y='CV Error Rates', data= err_df, label="CV Error␣
 ↪Rates")
```

[116]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a2794b3d0>
```



Generally, as could be seen from the above graph, the CV error rates reaches its minimum when cost = 0.5. Before the cost reaches its optimal value, the model is still underfitting, thus the error drops when cost increases. However, after passing the opitmial value, the CV error rates increases because of model overfitting. After cost value > 3, the increasing trend of CV error rates generally flattens, probably because the classfication scheme is fixed.

[123]:
```
X_train= preprocessing.scale(x_train)
```

[155]:
```
# SVMs with radial and polynomial basis kernels
params = {'C': [0.01, 0.1, 0.5, 1, 2],
          'kernel': ['poly','rbf'],
          'degree': [2, 3, 4],
          'gamma': [1, 10]
          }

svm_search = GridSearchCV(SVC(), params, refit=True, cv=10, n_jobs = 12,␣
 ↪verbose = 2 )
svm_search.fit(X_train, y_train)
```

```
Fitting 10 folds for each of 60 candidates, totalling 600 fits

[Parallel(n_jobs=12)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=12)]: Done  17 tasks      | elapsed:    0.8s
[Parallel(n_jobs=12)]: Done 138 tasks      | elapsed:    4.2s
[Parallel(n_jobs=12)]: Done 341 tasks      | elapsed:   10.2s
```

```
[Parallel(n_jobs=12)]: Done 600 out of 600 | elapsed:   19.4s finished
```

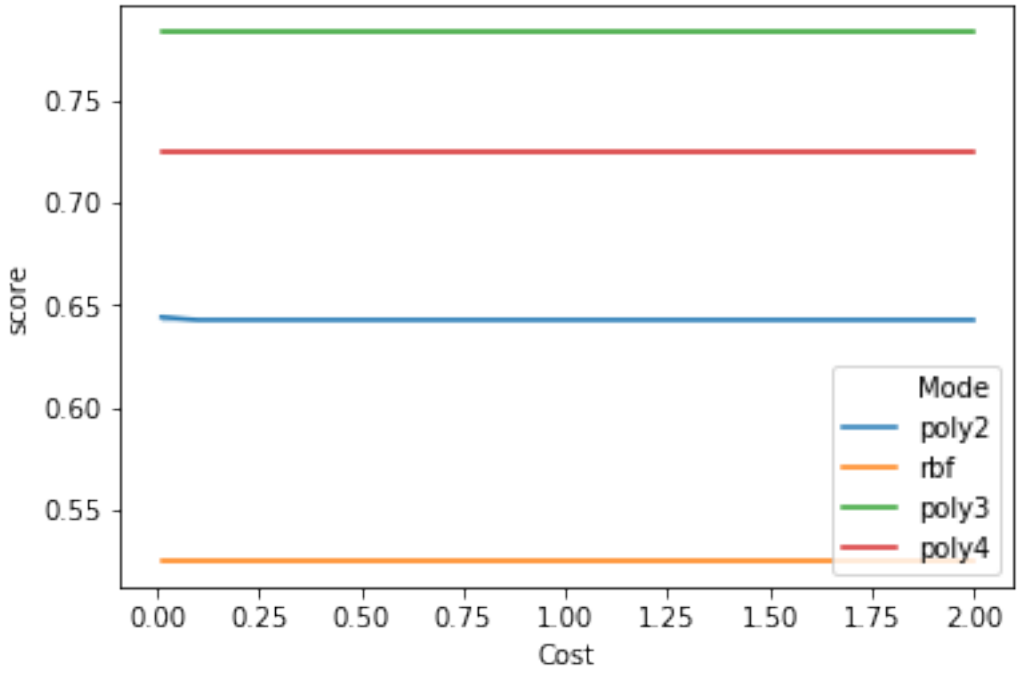
[155]:
```
GridSearchCV(cv=10, error_score='raise-deprecating',
             estimator=SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
                           decision_function_shape='ovr', degree=3,
                           gamma='auto_deprecated', kernel='rbf', max_iter=-1,
                           probability=False, random_state=None, shrinking=True,
                           tol=0.001, verbose=False),
             iid='warn', n_jobs=12,
             param_grid={'C': [0.01, 0.1, 0.5, 1, 2], 'degree': [2, 3, 4],
                         'gamma': [1, 10], 'kernel': ['poly', 'rbf']},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=2)
```

[156]:
```
result= svm_search.cv_results_
result_df= pd.DataFrame({"Cost": result['param_C'],
                         "kernel":result['param_kernel'],
                         "degree": result['param_degree'],
                         "gamma": result['param_gamma'],
                        "score": result['mean_test_score']})
```

[157]:
```
def define_kernel(row):

    degree=row['degree']
    kernel=row['kernel']

    if kernel=="poly":
        row['Mode']= kernel + str(degree)
    else:
        row['Mode'] = kernel
    return row

result_df = result_df.apply(define_kernel, axis=1)
```

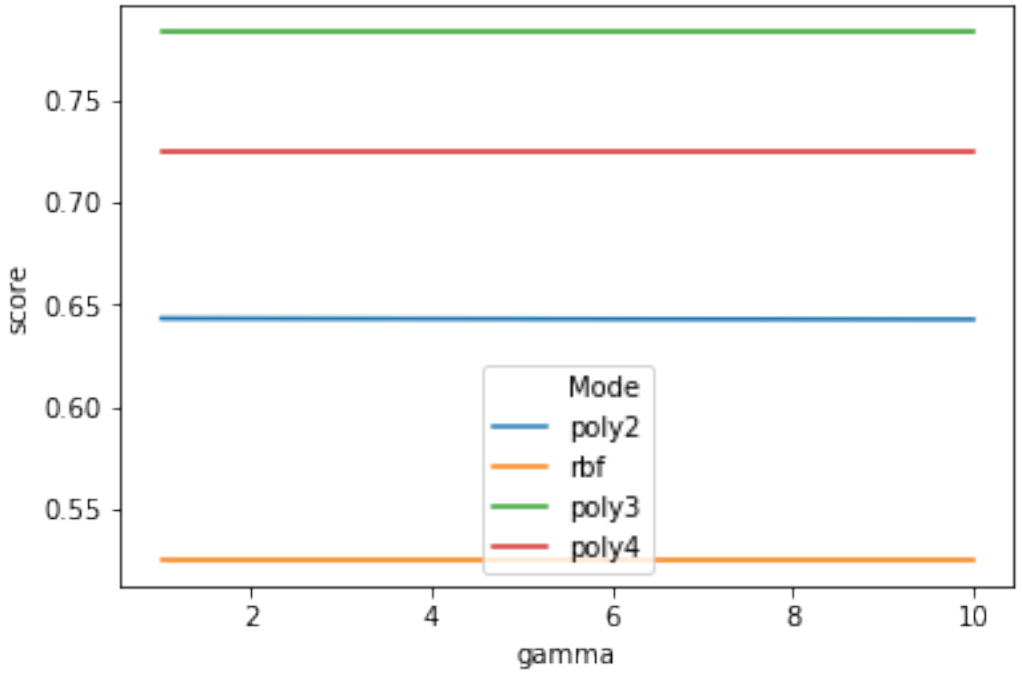
[158]:
```
# different cost and model performance
sns.lineplot(x="Cost", y='score', hue= "Mode", data=result_df)
```

[158]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a29075510>
```



[154]:
```
# different cost and model performance
sns.lineplot(x="gamma", y='score', hue= "Mode", data=result_df)
```

[154]: <matplotlib.axes._subplots.AxesSubplot at 0x1a26a40a10>



[162]:
```
svm_search.best_estimator_
```

[162]:
```
SVC(C=0.01, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma=1, kernel='poly',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

From the above graph, we could see that what makes the model score to be different is the kernel and degree selection in this application. As for the value of cost and gamma, their impact on model performance is very limited. For this problem, polynomial basis kernel is more powerful than the rbf kernel. Moreover, the optimal degree is 2 for polynomial kernel. Overall, the best parameter combination is cost = 0.01, degree=3,gamma=1 and with a polynomial kernel. Again, the choice of cost and gamma do not significantly influence the outcomes.