Xu_Yilun_HW06

March 6, 2020

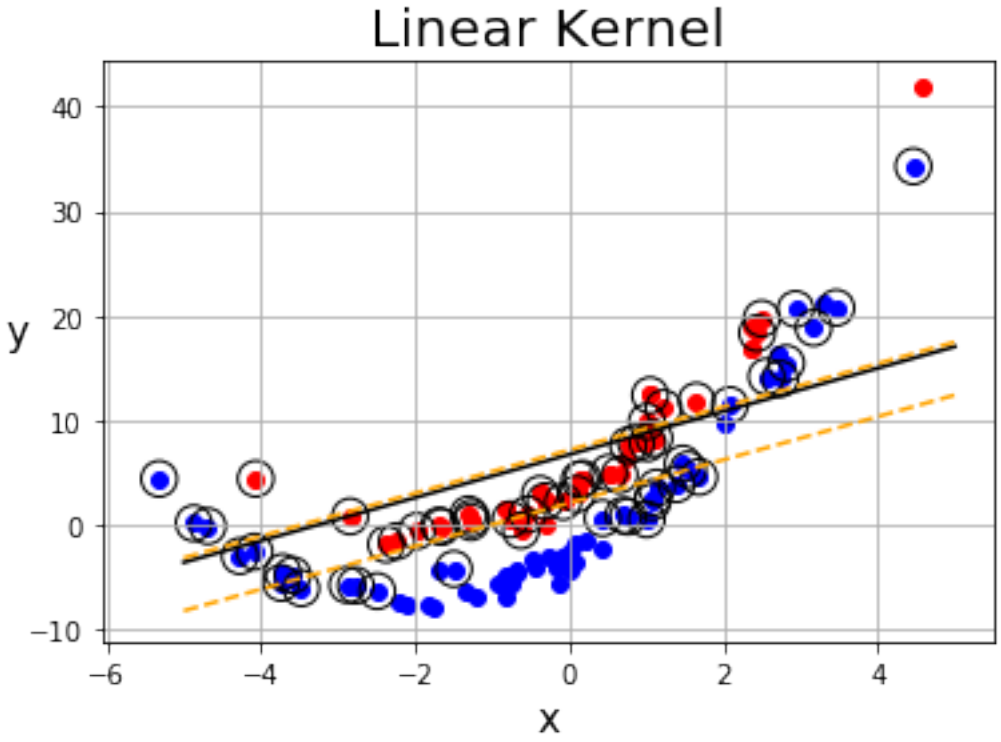
[1]:
```
import numpy as np
from sklearn.model_selection import *
from sklearn import svm
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
import seaborn
import pandas as pd
import warnings
warnings.filterwarnings("ignore")
from mpl_toolkits.mplot3d import Axes3D
```

1 Non-linear seperation

[2]:
```
np.random.seed(2)
x = np.random.normal(0, 2, 100)
y = x*4 + x**2 + np.random.normal(0, 1, 100)
index = list(range(100))
index_a = np.random.randint(0, 100, size = 50)
index_b = [i for i in index if i not in index_a]
y[index_a] += 3
y[index_b] -= 3

labels = np.asarray([1]*100)
labels[index_b] -= 2

data = np.column_stack((x,y))
x_train, x_test, y_train, y_test = train_test_split(data, labels, test_size = 0.
↪1)
```

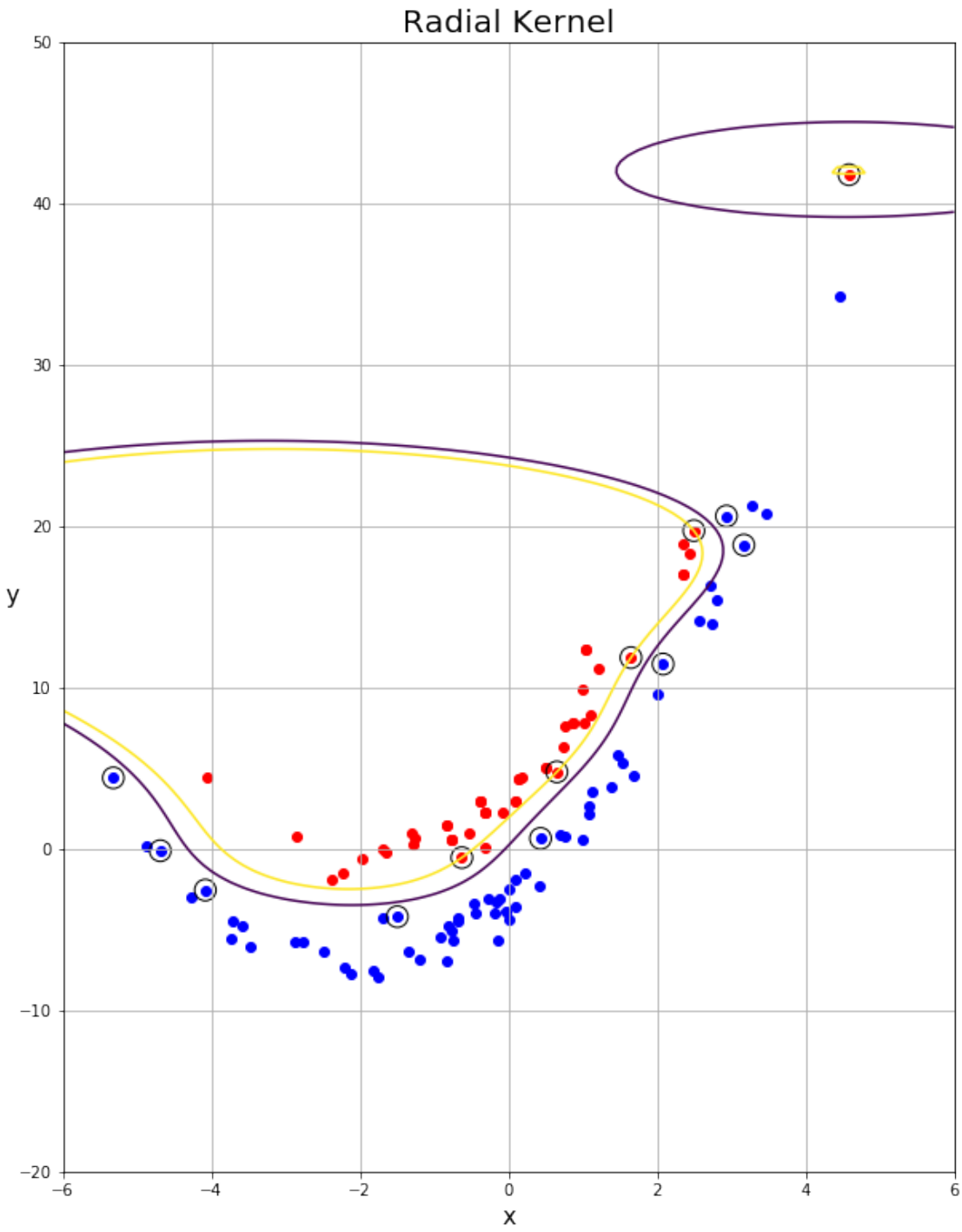
[3]:
```
svc_1 = svm.SVC(kernel = 'linear', C = 10000)
svc_1.fit(x_train, y_train)
ratio = -svc_1.coef_[0][0] / svc_1.coef_[0][1]
x_range = np.linspace(-5, 5)
y_range = ratio * x_range - (svc_1.intercept_[0]) / svc_1.coef_[0][1]
v1, v_last = svc_1.support_vectors_[0], svc_1.support_vectors_[-1]
```

```
y_range_lower = ratio * x_range + (v1[1] - ratio * v1[0])
y_range_higher = ratio * x_range + (v_last[1] - ratio * v_last[0])

plt.scatter(x[index_a], y[index_a], color='r')
plt.scatter(x[index_b], y[index_b], color='b')
plt.scatter(svc_1.support_vectors_[:, 0], svc_1.support_vectors_[:, 1],s=180,␣
 ↪facecolors='none', edgecolors='black')
plt.plot(x_range, y_range, 'k-')
plt.plot(x_range, y_range_lower, 'k--', color='orange')
plt.plot(x_range, y_range_higher, 'k--', color='orange')

plt.rcParams["figure.figsize"] = (10, 13)
plt.xlabel("x",size = 15)
plt.ylabel("y", rotation = 0,size = 15)
plt.grid()
plt.title("Linear Kernel",size = 20)
plt.show()
```



[4]:
```
rad_1 = svm.SVC(kernel='rbf', C = 10000)
rad_1.fit(x_train, y_train)
x_range_rad, y_range_rad = np.mgrid[-6:6:200j, -20:50:200j]
t = rad_1.decision_function(np.c_[x_range_rad.ravel(), y_range_rad.ravel()])
```

```
t = t.reshape(x_range_rad.shape)

plt.scatter(x[index_a], y[index_a], color='r')
plt.scatter(x[index_b], y[index_b], color='b')
plt.scatter(rad_1.support_vectors_[:, 0], rad_1.support_vectors_[:, 1], s =␣
 ↪180, facecolors='none', edgecolors='black')
plt.contour(x_range_rad, y_range_rad, t, levels=[0, 1])

plt.xlabel("x", size = 15)
plt.ylabel("y", rotation = 0, size = 15)
plt.grid()
plt.title("Radial Kernel", size = 20)
plt.rcParams["figure.figsize"] = (10,13)
plt.show()
```



[5]:
```
linear_kernal_train_error = 1 - accuracy_score(y_train, svc_1.predict(x_train))
linear_kernal_test_error = 1 - accuracy_score(y_test, svc_1.predict(x_test))
radial_kernal_train_error = 1 - accuracy_score(y_train, rad_1.predict(x_train))
radial_kernal_test_error = 1 - accuracy_score(y_test, rad_1.predict(x_test))
```

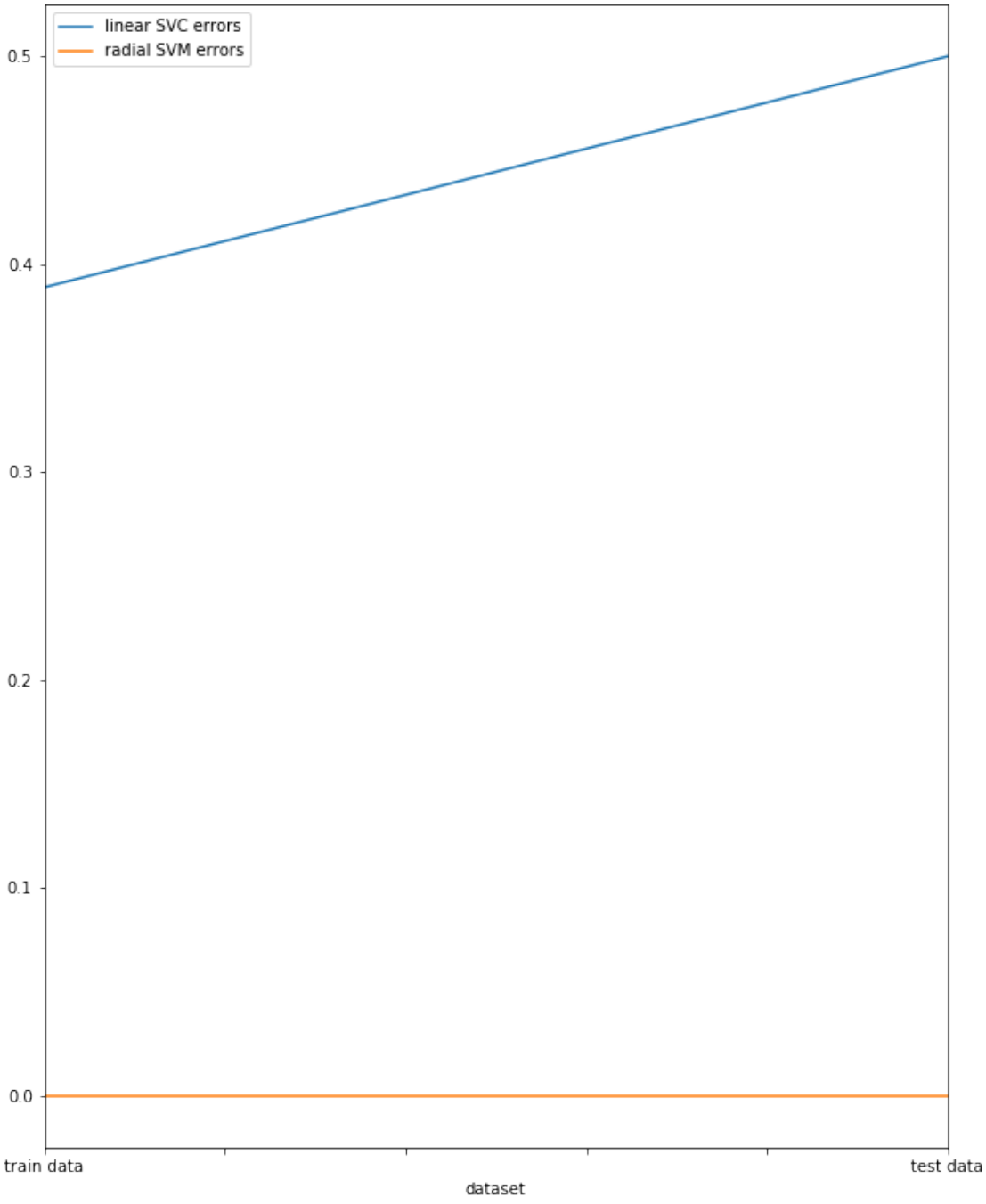
```
linear = [linear_kernal_train_error, linear_kernal_test_error]
radial = [radial_kernal_train_error, radial_kernal_test_error]
dataset = ['train data', 'test data']
data = {'dataset': dataset, 'linear SVC errors': linear, 'radial SVM errors':␣
↪radial}
linear_radial_comparison = pd.DataFrame(data)
linear_radial_comparison
```

[5]:

	dataset	linear SVC errors	radial SVM errors
0	train data	0.388889	0.0
1	test data	0.500000	0.0

[6]:
```
linear_radial_comparison.plot(x = 'dataset',y = linear_radial_comparison.
↪columns[1:])
```

[6]: <matplotlib.axes._subplots.AxesSubplot at 0x1662e649288>



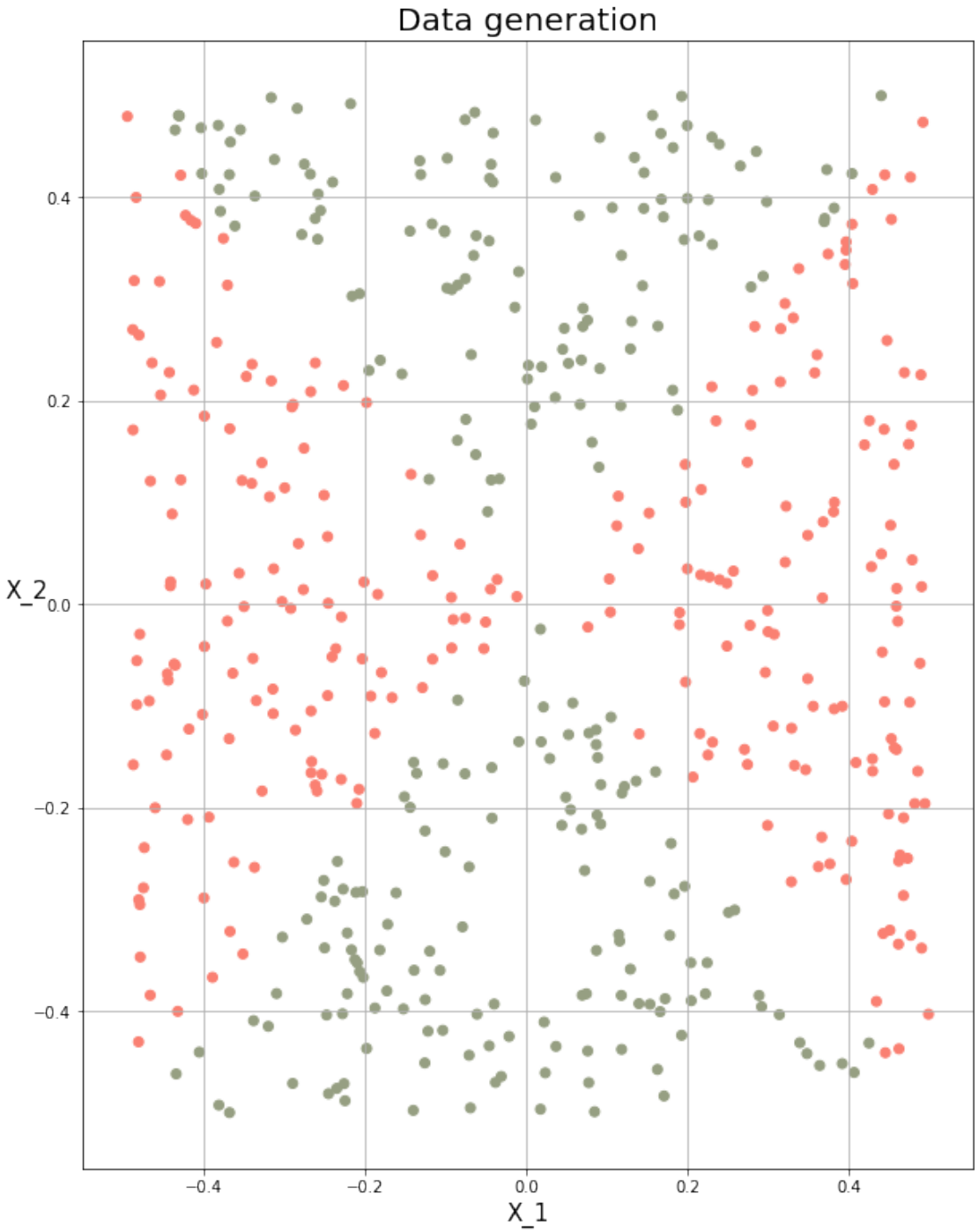
According to the above analysis, we can see that a supporting vector machine with a radial kernel will outperfom a support vector classifier (a linear kernel) both on the train and test dataset.

2 SVM vs. logistic regression

2.1 2 & 3

[7]:
```
np.random.seed(0)
x_1 = np.random.uniform(0, 1, 500) - 0.5
x_2 = np.random.uniform(0, 1, 500) - 0.5
y = ((x_1**2 - x_2**2) > 0).astype(int)
color= ['#969f82' if l == 0 else 'salmon' for l in y]

plt.xlabel("X_1", size = 15)
plt.ylabel("X_2", rotation = 0, size = 15)
plt.grid()
plt.scatter(x_1, x_2, color=color)
plt.title("Data generation", size = 20)
```

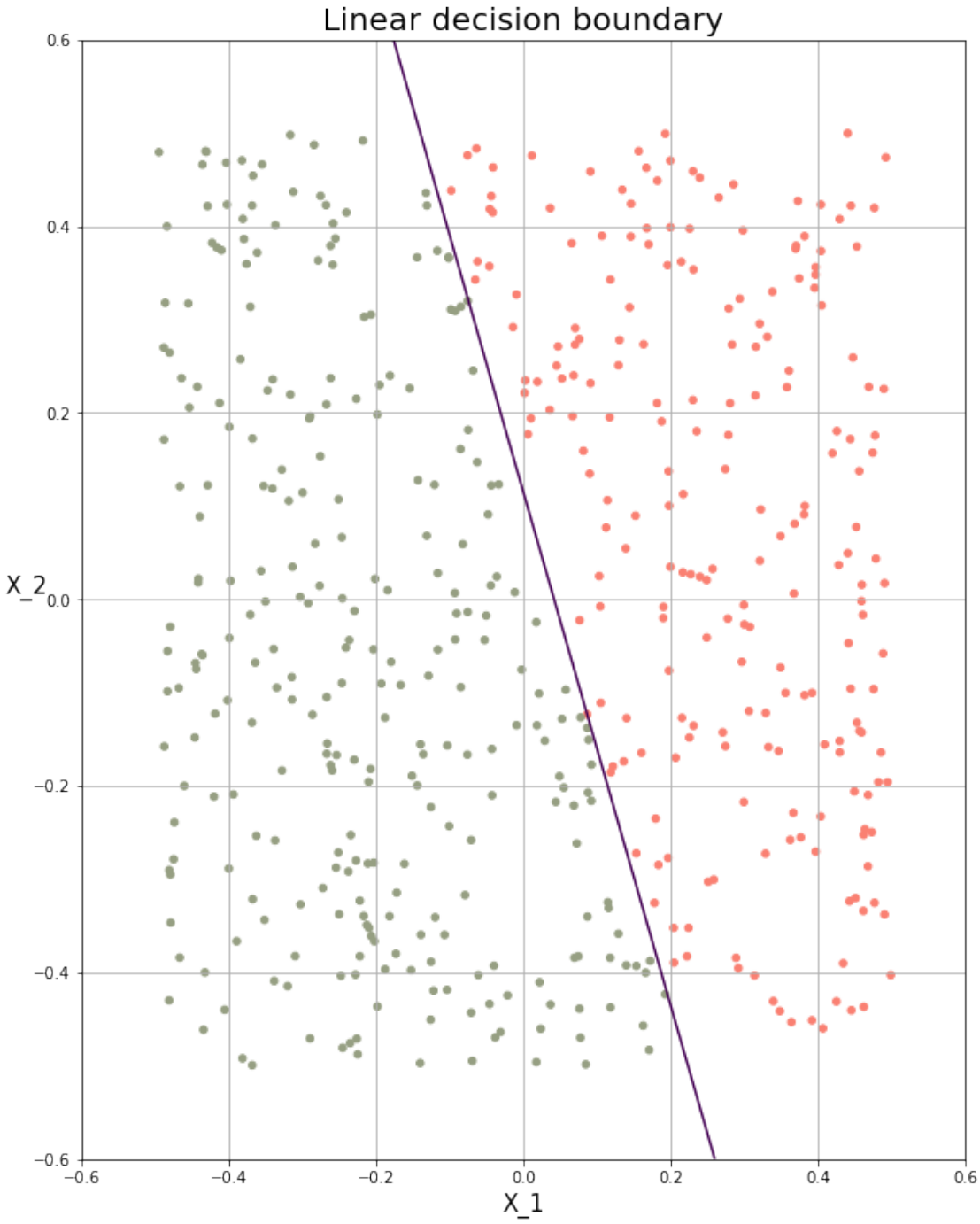
[7]:
```
Text(0.5, 1.0, 'Data generation')
```


Data generation
0.4
0.2
0.0
−0.2
−0.4
X_2
−0.4
−0.2
0.0
0.2
0.4
X_1

2.2 4 & 5

[8]:
```
xs = np.column_stack((x_1,x_2))
lr = LogisticRegression(fit_intercept=True,random_state=0)
lr.fit(xs, y)
y_pred_train = lr.predict(xs)
cols = {'x_1':x_1,'x_2':x_2,'y_pred_train':y_pred_train}
df = pd.DataFrame(cols)
df.plot.scatter(x='x_1',y='x_2',c = ['#969f82' if l == 0 else 'salmon' for l in␣
↪y_pred_train])

x_range_lr, y_range_lr = np.mgrid[-.6:.6:200j, -.6:.6:200j]
t = lr.decision_function(np.c_[x_range_lr.ravel(), y_range_lr.ravel()])
t = t.reshape(x_range_lr.shape)
plt.contour(x_range_lr, y_range_lr, t, levels=[0, 1])

plt.xlabel("X_1", size = 15)
plt.ylabel("X_2", rotation = 0, size = 15)
plt.grid()
plt.title("Linear decision boundary", size = 20)
```

[8]:
```
Text(0.5, 1.0, 'Linear decision boundary')
```

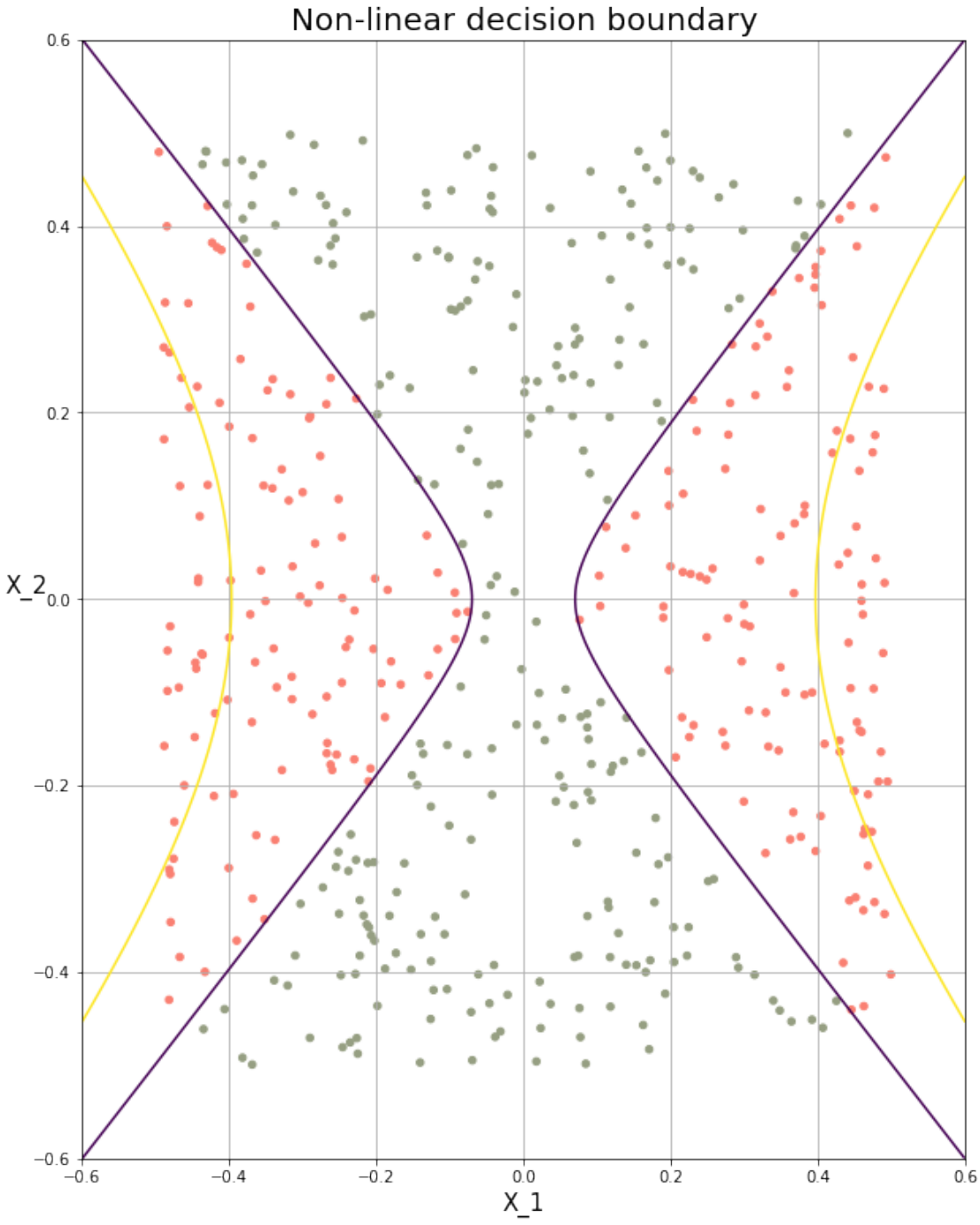


According to the above figure, we can see that the predicted decision boundary looks like a linear one.

2.3 6 & 7

[9]:
```
xs_2 = np.column_stack((x_1**2,x_2**2))
lr_2 = LogisticRegression(random_state=0, fit_intercept=True)
lr_2.fit(xs_2, y)
y_pred_train_2 = lr_2.predict(xs_2)

cols = {'x_1':x_1,'x_2':x_2,'y_pred_train_2':y_pred_train_2}
df2 = pd.DataFrame(cols)
df2.plot.scatter(x = 'x_1',y = 'x_2',c = ['#969f82' if l == 0 else 'salmon' for␣
↪l in y_pred_train_2])
x_range_lr_2, y_range_lr_2 = np.mgrid[-.6:.6:200j, -.6:.6:200j]
t_2 = lr_2.decision_function(np.c_[x_range_lr_2.ravel()**2, y_range_lr_2.
↪ravel()**2])
t_2 = t_2.reshape(x_range_lr_2.shape)
plt.contour(x_range_lr_2, y_range_lr_2, t_2, levels=[0, 1])

plt.xlabel("X_1", size = 15)
plt.ylabel("X_2", rotation = 0, size = 15)
plt.grid()
plt.title("Non-linear decision boundary", size = 20)
```

[9]:
```
Text(0.5, 1.0, 'Non-linear decision boundary')
```



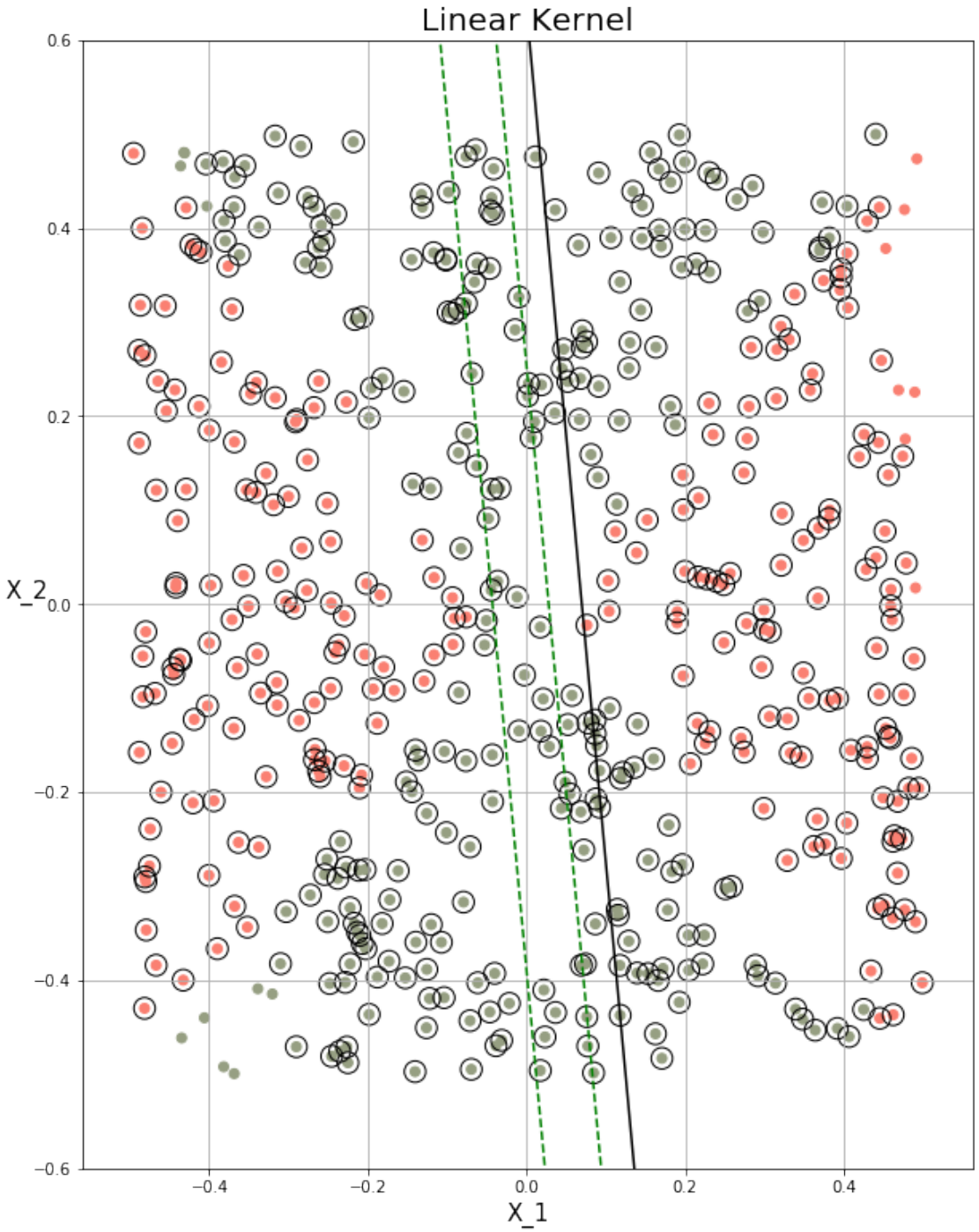
According to the above figure, we can see that the decision boundary is non-linear.

2.4 8

[10]:
```
y = ((x_1**2 - x_2**2) > 0).astype(int)
xs = np.column_stack((x_1,x_2))
clf = svm.SVC(kernel='linear', C = 1000)
clf.fit(xs, y)
y_pred_linear = clf.predict(xs)
color= ['red' if l == 0 else 'green' for l in y_pred_linear]
coef = clf.coef_[0]
ratio = -coef[0] / coef[1]
xx = np.linspace(-0.5, 0.5)
yy = ratio * xx - (clf.intercept_[0]) / coef[1]
```

[11]:
```
v1,v_last = clf.support_vectors_[0], clf.support_vectors_[-1]
y_low = ratio * xx + (v1[1] - ratio * v1[0])
y_high = ratio * xx + (v_last[1] - ratio * v_last[0])
plt.scatter(x_1, x_2, color = ['#969f82' if l == 0 else 'salmon' for l in␣
 ↪y_pred_train_2])
plt.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1],s = 180,␣
 ↪facecolors = 'none', edgecolors = 'black')
plt.xlabel("X_1", size = 15)
plt.ylabel("X_2", rotation = 0, size = 15)
plt.ylim(-0.6,0.6)
plt.grid()
plt.title("Linear Kernel", size = 20)
plt.plot(xx, yy, 'k-')
plt.plot(xx, y_low, 'k--', color='g')
plt.plot(xx, y_high, 'k--', color='g')
```

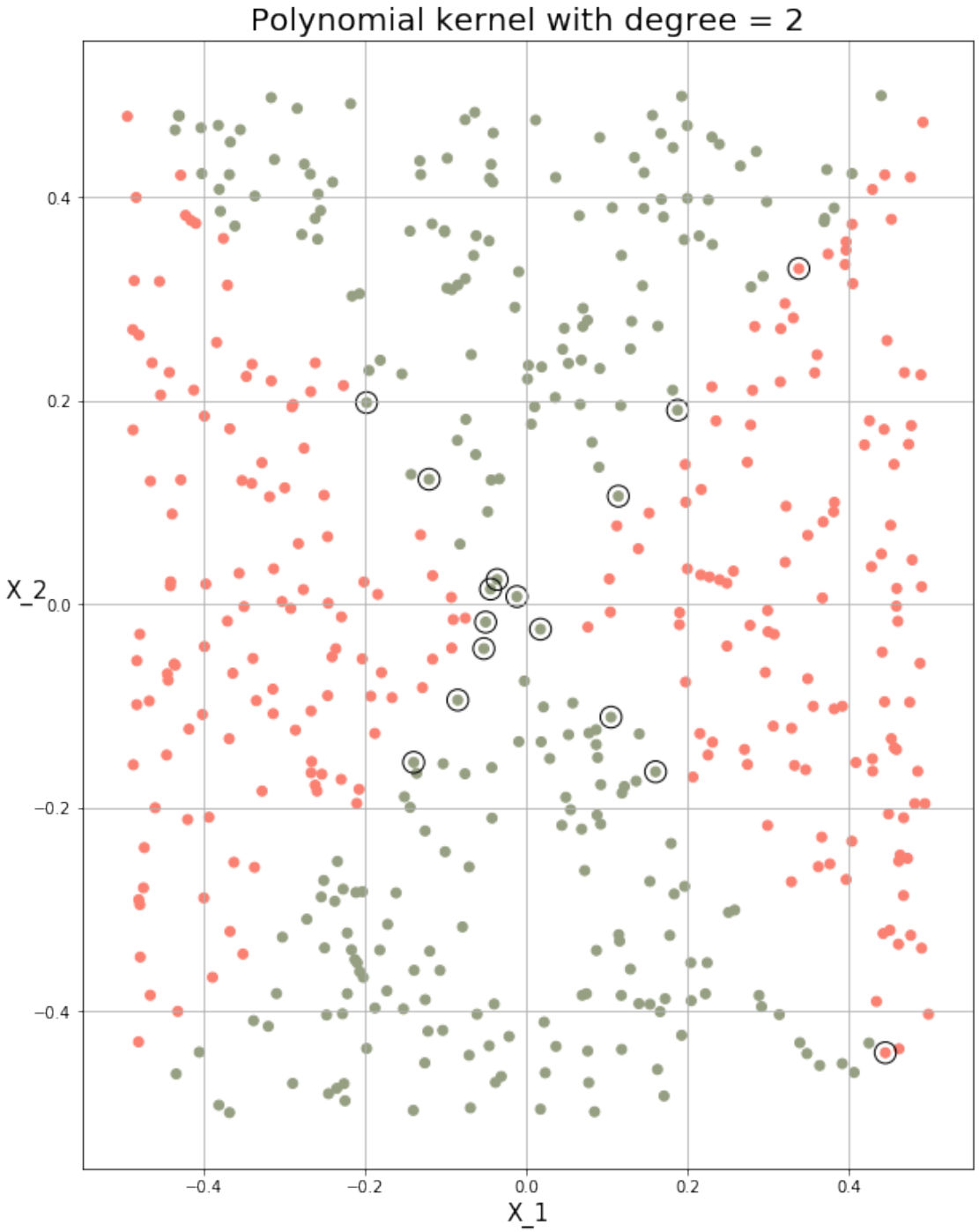
[11]:
```
[<matplotlib.lines.Line2D at 0x1662ed88508>]
```


Linear Kernel
X_1
X_2
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.4
−0.2
0.0
0.2
0.4

2.5 9

[12]:
```
np.random.seed(0)
poly_clf = svm.SVC(kernel = 'poly', degree = 2, C = 1000)
poly_clf.fit(xs, y)
y_pred_poly = poly_clf.predict(xs)

plt.scatter(x_1, x_2, color=['#969f82' if l == 0 else 'salmon' for l in␣
↪y_pred_train_2])
plt.scatter(poly_clf.support_vectors_[:, 0], poly_clf.support_vectors_[:, 1],s␣
↪= 180, facecolors = 'none', edgecolors = 'black')

plt.xlabel("X_1", size = 15)
plt.ylabel("X_2", rotation = 0, size = 15)
plt.grid()
plt.title("Polynomial kernel with degree = 2", size = 20)
```

[12]:
```
Text(0.5, 1.0, 'Polynomial kernel with degree = 2')
```


Polynomial kernel with degree = 2
X_2
X_1
0.4
0.2
0.0
−0.2
−0.4
−0.4
−0.2
0.0
0.2
0.4

2.6 10

[13]:
```
##linear logistic regression
llr_score = accuracy_score(y_pred_train, y)
##non-linear logstic regression
nlr_score = accuracy_score(y_pred_train_2, y)
##linear kernel
lk_score = accuracy_score(y_pred_linear, y)
##non-linear kernel
nlk_score = accuracy_score(y_pred_poly, y)
print('Accuracy scores :')
print('linear logistic regression :', llr_score)
print('non-linear logistic regression :', nlr_score)
print('linear kernel SVM :', lk_score)
print('non-linear kernel SVM :', nlk_score)
```

```
Accuracy scores :
linear logistic regression : 0.558
non-linear logistic regression : 0.98
linear kernel SVM : 0.566
non-linear kernel SVM : 0.994
```

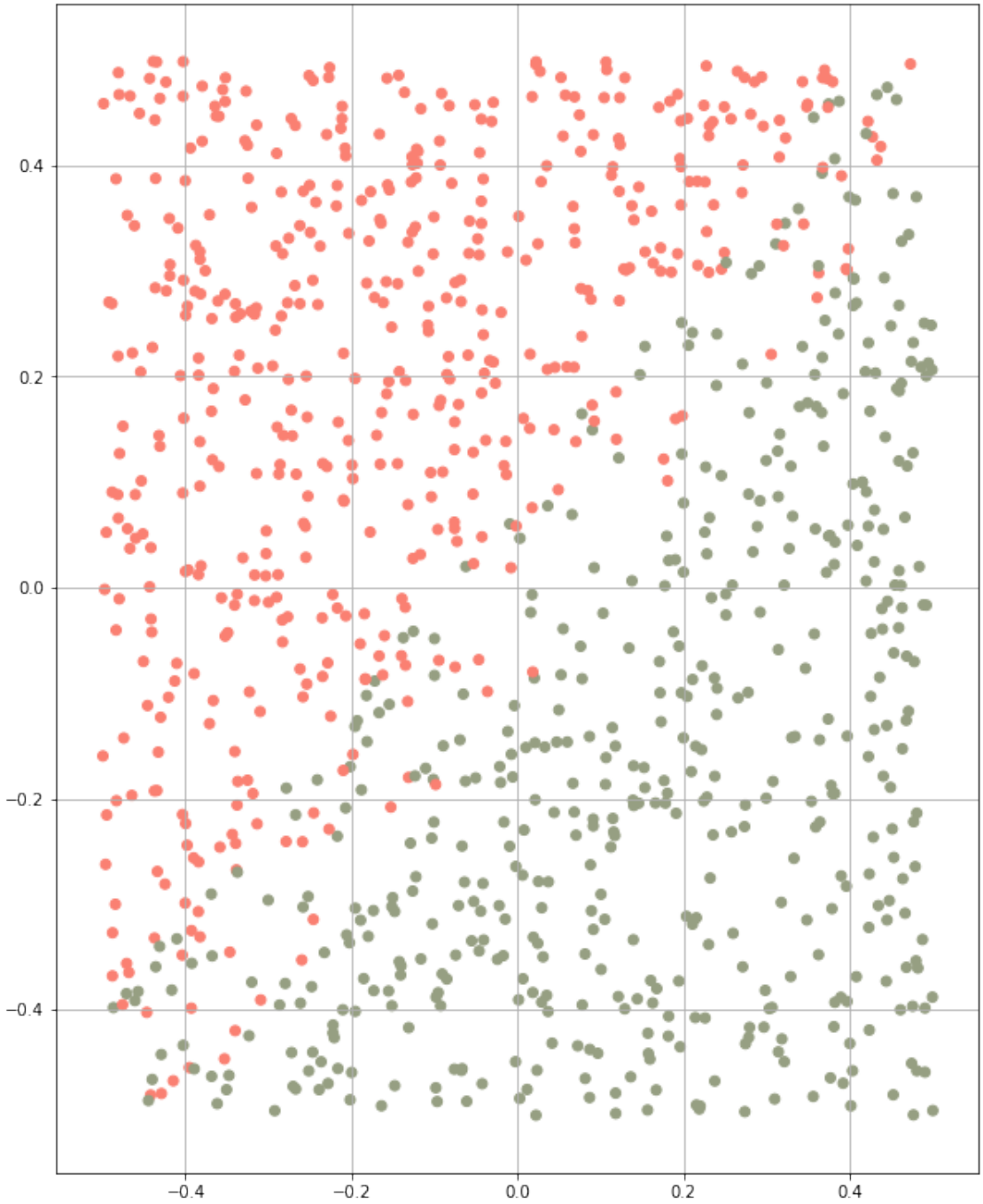
According to the above data, we can see that in this case, non-linear decision boundaries both get higher accuracy score compared to linear ones. This means that a non-linear model will be better to make the prediction in this case. However, the accuracy score of non-linear kernel SVM is higher than the one of non-linear logistic regression, although they are both high enough. To achive a higher accuracy score, we should pick non-linear kernel SVM. However, SVM tries to maximize the margin between the closest support vectors, while logistic regression wants to maximize the posterior class probability. Therefore, the selection between these two models depends on our goal of building this model.

3 Tuning Cost

3.1 11

[14]:
```
np.random.seed(0)
x1 = np.random.uniform(0, 1, 1000) - 0.5
x2 = np.random.uniform(0, 1, 1000) - 0.5
x = np.column_stack((x1,x2))
diff = x2-x1
y = [None] * len(diff)

for i, num in enumerate(diff):
    if num > 0.1:
        y[i] = 1
    elif num < -0.1:
        y[i] = 0
    else:
```

```
        y[i] = np.random.uniform(0, 1) > 0.5

plt.scatter(x1, x2, color=['#969f82' if l == 0 else 'salmon' for l in y])
plt.grid()
plt.rcParams["figure.figsize"] = (10,13)
plt.show()
```



3.2 12 & 13

[15]:
```
np.random.seed(0)
x_train, x_valid, y_train, y_valid = train_test_split(x, y, test_size=0.5)

x1_test = np.random.uniform(0, 1, 500) - 0.5
x2_test = np.random.uniform(0, 1, 500) - 0.5
x_test = np.column_stack((x1_test,x2_test))

diff_test = x2_test - x1_test
y_test = [None] * len(diff_test)

for i, num in enumerate(diff_test):
    if num > 0.1:
        y_test[i] = 1
    elif num < -0.1:
        y_test[i] = 0
    else:
        y_test[i] = np.random.uniform(0, 1) > 0.5
```

[16]:
```
cost = [0.0001, 0.001, 0.01, 0.1, 1, 10, 20, 100, 500, 1000, 10000]
cv_error, train_error, test_error = [], [], []

for c in cost:
    clf = svm.SVC(kernel='linear', C = c)
    clf.fit(x_train, y_train)
    pred_train, pred_valid, pred_test = clf.predict(x_train), clf.
 ↪predict(x_valid), clf.predict(x_test)
    train_error.append(1 - accuracy_score(y_train, pred_train))
    cv_error.append(1 - accuracy_score(y_valid, pred_valid))
    test_error.append(1 - accuracy_score(y_test, pred_test))
data = {'train_error': train_error,'cv_error': cv_error, 'test_error':␣
 ↪test_error,'cost': cost}
errors = pd.DataFrame(data)
```

[17]:
```
errors.sort_values(by = "train_error" , ascending = True)
```

[17]:

	train_error	cv_error	test_error	cost
6	0.088	0.088	0.078	20.0000
4	0.090	0.086	0.086	1.0000
5	0.090	0.086	0.080	10.0000
8	0.092	0.088	0.078	500.0000
9	0.092	0.084	0.080	1000.0000
10	0.092	0.084	0.080	10000.0000
3	0.094	0.072	0.076	0.1000
7	0.094	0.084	0.080	100.0000
2	0.250	0.302	0.252	0.0100

0	0.496	0.494	0.500	0.0001
1	0.496	0.494	0.500	0.0010

[18]:
```
errors.sort_values(by = "cv_error" , ascending = True)
```

[18]:

	train_error	cv_error	test_error	cost
3	0.094	0.072	0.076	0.1000
7	0.094	0.084	0.080	100.0000
9	0.092	0.084	0.080	1000.0000
10	0.092	0.084	0.080	10000.0000
4	0.090	0.086	0.086	1.0000
5	0.090	0.086	0.080	10.0000
6	0.088	0.088	0.078	20.0000
8	0.092	0.088	0.078	500.0000
2	0.250	0.302	0.252	0.0100
0	0.496	0.494	0.500	0.0001
1	0.496	0.494	0.500	0.0010

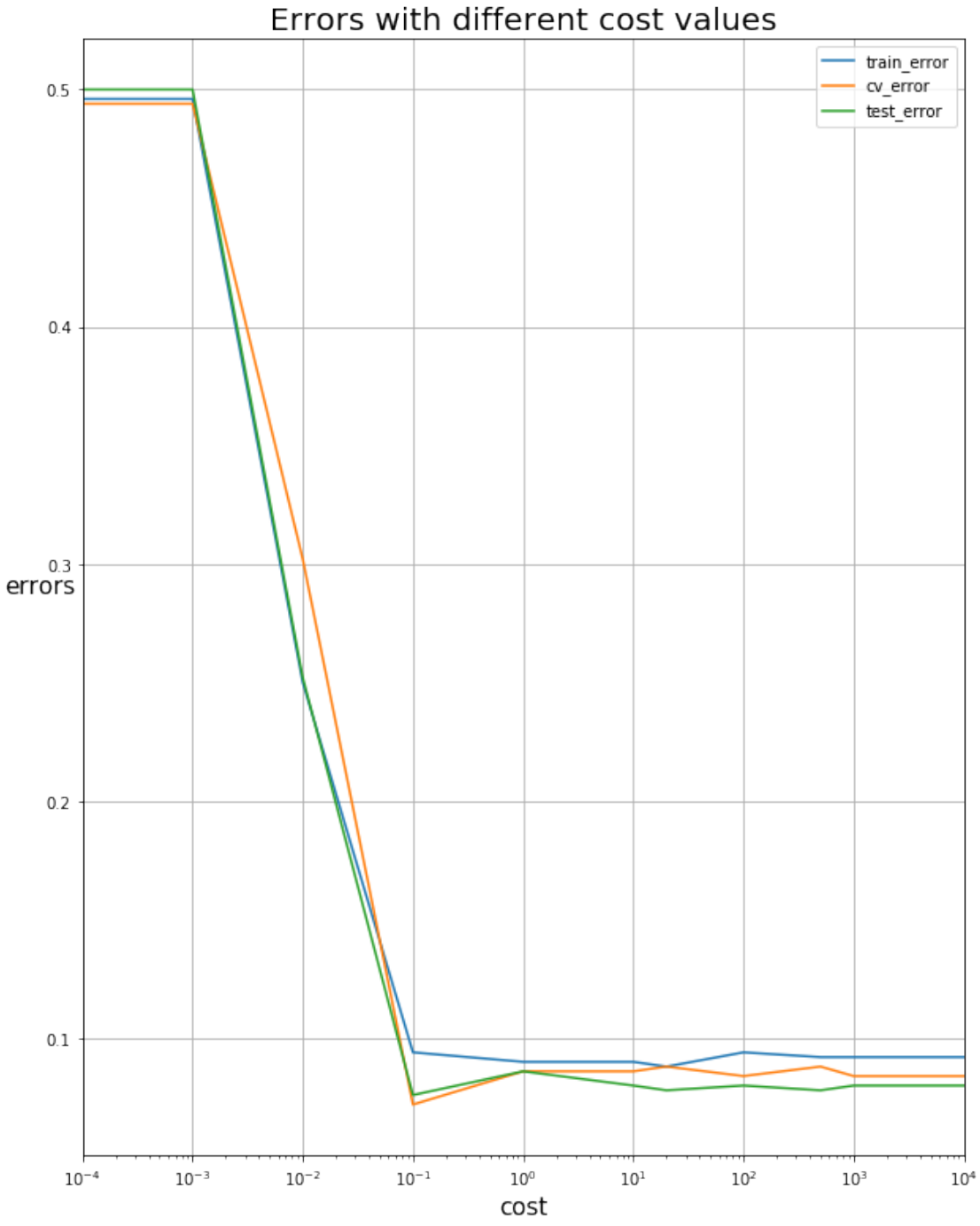
[19]:
```
errors.sort_values(by = "test_error" , ascending = True)
```

[19]:

	train_error	cv_error	test_error	cost
3	0.094	0.072	0.076	0.1000
6	0.088	0.088	0.078	20.0000
8	0.092	0.088	0.078	500.0000
5	0.090	0.086	0.080	10.0000
7	0.094	0.084	0.080	100.0000
9	0.092	0.084	0.080	1000.0000
10	0.092	0.084	0.080	10000.0000
4	0.090	0.086	0.086	1.0000
2	0.250	0.302	0.252	0.0100
0	0.496	0.494	0.500	0.0001
1	0.496	0.494	0.500	0.0010

For train data, the best cost value is 20. For cross-validation data, the best cost value is 0.1. For test data, the best cost value is 0.1.

[20]:
```
errors.plot(x = "cost", y = errors.columns[:-1])
plt.xscale('log')
plt.xlabel("cost", size = 15)
plt.ylabel("errors", rotation = 0, size = 15)
plt.title("Errors with different cost values", size = 20)
plt.grid()
plt.rcParams["figure.figsize"] = (10,13)
```



3.3 14

According to the above analysis, we can see that when cost value is equal to or bigger than 0.1, the errors are significantly decreased. For train data, the best cost value is 20. For cross-validation data and test data, the best cost value is 0.1.Therefore, if we want to pick a cost value which minimizes the error in train data, we pick 20. If we want to minimize the error in the cross-validation data or

test data, we choose the cost value of 0.1. In addition, we can see from the above figure that when the cost value is equal to or bigger than 0.1, the errors (whether in train data, cross-validation data or test data) are very small. This indicates that linear SVC is a good model in this case to make the prediction.

4 Application: Predicting attitudes towards racist college professors

[21]:
```
train = pd.read_csv('C:/Users/mac/Desktop/temp/problem-set-6-master/data/
↪gss_train.csv')
test = pd.read_csv('C:/Users/mac/Desktop/temp/problem-set-6-master/data/
↪gss_test.csv')
x_cols = [x for x in train.columns if x != 'colrac']
x_train = train[x_cols]
y_train = train[['colrac']]
x_test = test[x_cols]
y_test = test[['colrac']]
```

4.1 15

[22]:
```
param_lr = {'C': [0.01, 0.1, 5, 10, 100]}
svc_grid = GridSearchCV(svm.SVC(kernel='linear'),param_lr, cv = 10, scoring =␣
↪'accuracy')
svc_grid.fit(x_train, y_train)
```

[22]:
```
GridSearchCV(cv=10, error_score=nan,
             estimator=SVC(C=1.0, break_ties=False, cache_size=200,
                           class_weight=None, coef0=0.0,
                           decision_function_shape='ovr', degree=3,
                           gamma='scale', kernel='linear', max_iter=-1,
                           probability=False, random_state=None, shrinking=True,
                           tol=0.001, verbose=False),
             iid='deprecated', n_jobs=None,
             param_grid={'C': [0.01, 0.1, 5, 10, 100]}, pre_dispatch='2*n_jobs',
             refit=True, return_train_score=False, scoring='accuracy',
             verbose=0)
```

- I intended to use the cost list of cost = [0.0001, 0.001, 0.01, 0.1, 1, 10, 20, 100, 500, 1000, 10000]. However, it will take too much time and I decided to cut the list into a smaller one.

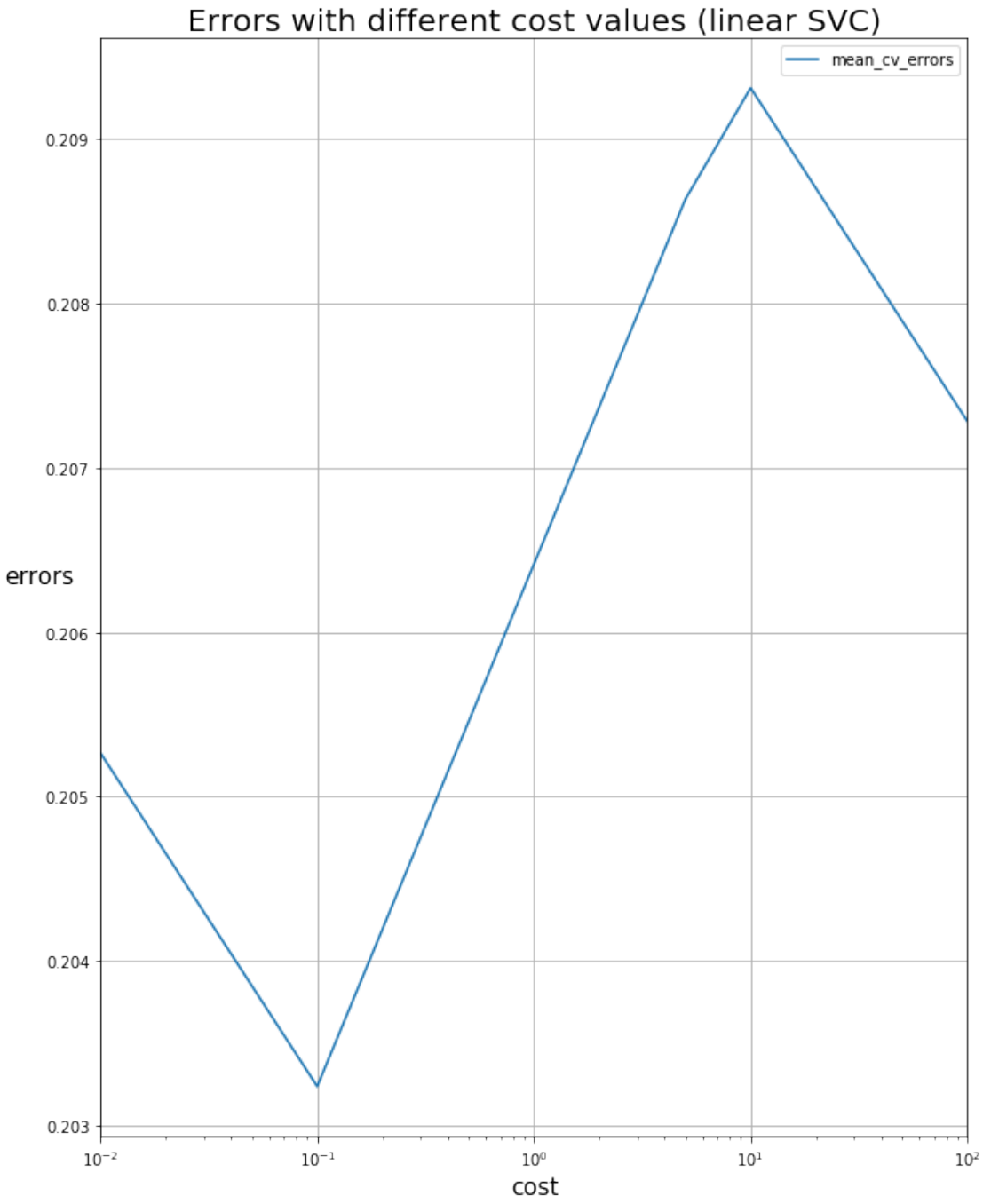
[23]:
```
linear_costs_errors = pd.DataFrame(svc_grid.cv_results_)
linear_costs_errors['mean_cv_errors'] = 1 -␣
↪linear_costs_errors['mean_test_score']
```

[24]:
```
linear_costs_errors.sort_values(by = "rank_test_score" , ascending =␣
↪True)[['param_C', 'rank_test_score','mean_test_score','mean_cv_errors']]
```

[24]:

	param_C	rank_test_score	mean_test_score	mean_cv_errors
1	0.1	1	0.796762	0.203238
0	0.01	2	0.794735	0.205265
4	100	3	0.792717	0.207283
2	5	4	0.791366	0.208634
3	10	5	0.790690	0.209310

[25]:
```
linear_costs_errors.plot(x = "param_C", y = 'mean_cv_errors')
plt.xscale('log')
plt.xlabel("cost", size = 15)
plt.ylabel("errors", rotation = 0, size = 15)
plt.title("Errors with different cost values (linear SVC)", size = 20)
plt.grid()
```



[26]:
```
print(svc_grid.best_estimator_, svc_grid.best_score_)
```

```
SVC(C=0.1, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='scale', kernel='linear',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False) 0.7967621984400509
```

According to the above information, we can know that the best cost value here is 0.1.

4.2 16

[27]:
```
parameters = {'C':[0.01, 0.1, 5, 10, 100], 'gamma':[0.01, 0.1, 5, 10]}
radial_grid = GridSearchCV(svm.SVC(kernel = 'rbf'), parameters, cv = 10,␣
↪scoring = 'accuracy')
radial_grid.fit(x_train,y_train)
```

[27]:
```
GridSearchCV(cv=10, error_score=nan,
             estimator=SVC(C=1.0, break_ties=False, cache_size=200,
                           class_weight=None, coef0=0.0,
                           decision_function_shape='ovr', degree=3,
                           gamma='scale', kernel='rbf', max_iter=-1,
                           probability=False, random_state=None, shrinking=True,
                           tol=0.001, verbose=False),
             iid='deprecated', n_jobs=None,
             param_grid={'C': [0.01, 0.1, 5, 10, 100],
                         'gamma': [0.01, 0.1, 5, 10]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='accuracy', verbose=0)
```

[28]:
```
radial_costs_errors = pd.DataFrame(radial_grid.cv_results_)
radial_costs_errors['mean_cv_errors'] = 1 -␣
↪radial_costs_errors['mean_test_score']
```

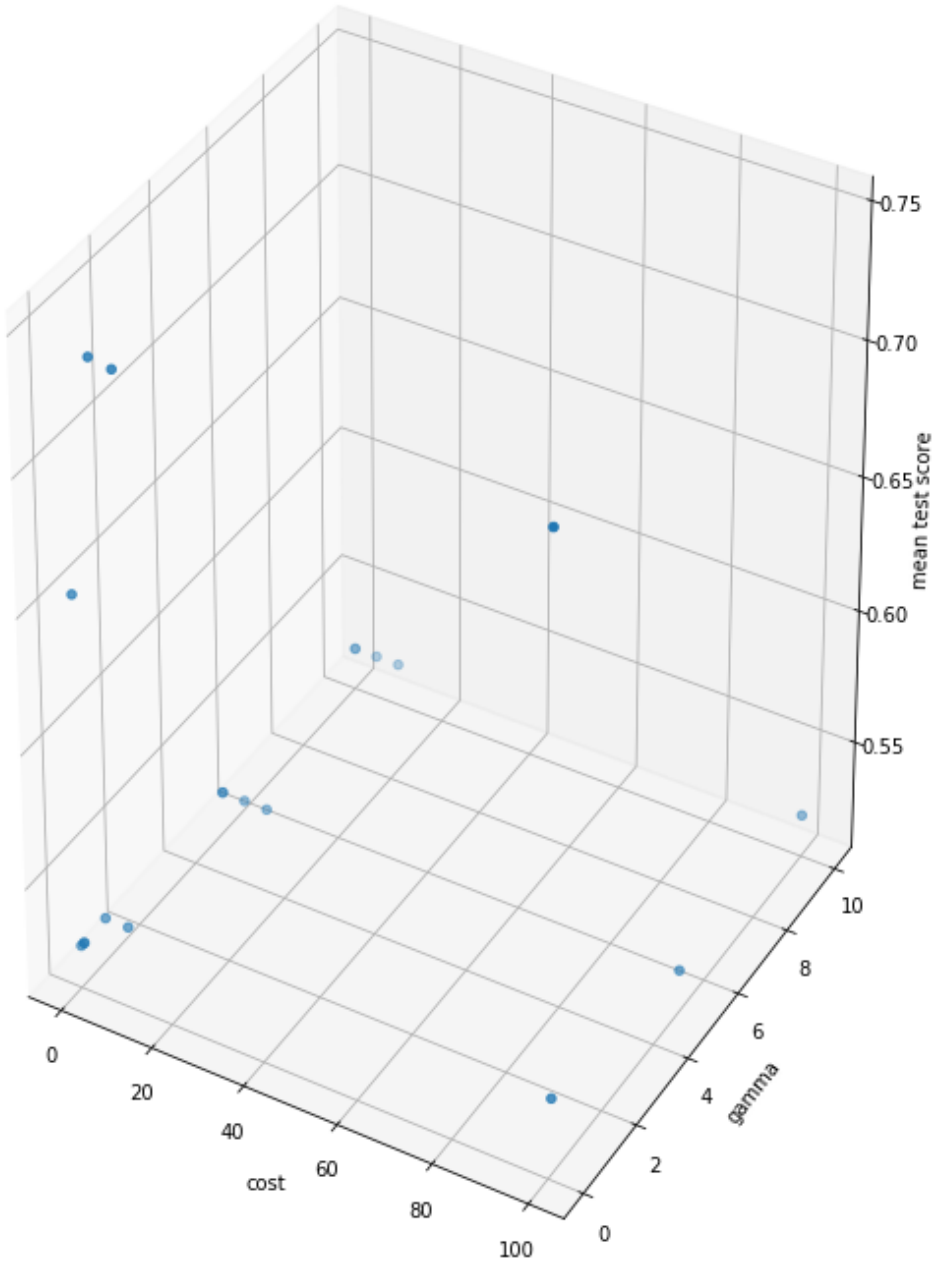
[29]:
```
radial_costs_errors.sort_values(by = "rank_test_score" , ascending =␣
↪True)[['params', 'rank_test_score','mean_test_score','mean_cv_errors']]
```

[29]:

	params	rank_test_score	mean_test_score	\
8	{'C': 5, 'gamma': 0.01}	1	0.742731	
16	{'C': 100, 'gamma': 0.01}	2	0.741379	
12	{'C': 10, 'gamma': 0.01}	2	0.741379	
4	{'C': 0.1, 'gamma': 0.01}	4	0.655664	
9	{'C': 5, 'gamma': 0.1}	5	0.538146	
17	{'C': 100, 'gamma': 0.1}	5	0.538146	
13	{'C': 10, 'gamma': 0.1}	5	0.538146	
15	{'C': 10, 'gamma': 10}	8	0.525322	
14	{'C': 10, 'gamma': 5}	8	0.525322	
11	{'C': 5, 'gamma': 10}	8	0.525322	
0	{'C': 0.01, 'gamma': 0.01}	8	0.525322	
18	{'C': 100, 'gamma': 5}	8	0.525322	
7	{'C': 0.1, 'gamma': 10}	8	0.525322	
6	{'C': 0.1, 'gamma': 5}	8	0.525322	
5	{'C': 0.1, 'gamma': 0.1}	8	0.525322	
3	{'C': 0.01, 'gamma': 10}	8	0.525322	
2	{'C': 0.01, 'gamma': 5}	8	0.525322	

```
1    {'C': 0.01, 'gamma': 0.1}                8         0.525322
10        {'C': 5, 'gamma': 5}                8         0.525322
19     {'C': 100, 'gamma': 10}                8         0.525322

    mean_cv_errors
8         0.257269
16        0.258621
12        0.258621
4         0.344336
9         0.461854
17        0.461854
13        0.461854
15        0.474678
14        0.474678
11        0.474678
0         0.474678
18        0.474678
7         0.474678
6         0.474678
5         0.474678
3         0.474678
2         0.474678
1         0.474678
10        0.474678
19        0.474678
```

[30]:
```
d3_radial = plt.figure().gca(projection = '3d')
d3_radial.scatter(radial_costs_errors['param_C'],␣
 ↪radial_costs_errors['param_gamma'], radial_costs_errors['mean_test_score'])
d3_radial.set_xlabel('cost')
d3_radial.set_ylabel('gamma')
d3_radial.set_zlabel('mean test score')
plt.show()
```



[31]:
```
parameters = {'C':[0.01, 0.1, 5, 10, 100], 'degree':[2,3,4,5]}
poly_grid = GridSearchCV(svm.SVC(kernel = 'poly'), parameters, cv = 10, scoring␣
 ↪= 'accuracy')
poly_grid.fit(x_train,y_train)
```

```
linear_costs_errors.sort_values(by = "rank_test_score" , ascending =␣
↪True)[['param_C', 'rank_test_score','mean_test_score','mean_cv_errors']]
```

[31]:

	param_C	rank_test_score	mean_test_score	mean_cv_errors
1	0.1	1	0.796762	0.203238
0	0.01	2	0.794735	0.205265
4	100	3	0.792717	0.207283
2	5	4	0.791366	0.208634
3	10	5	0.790690	0.209310

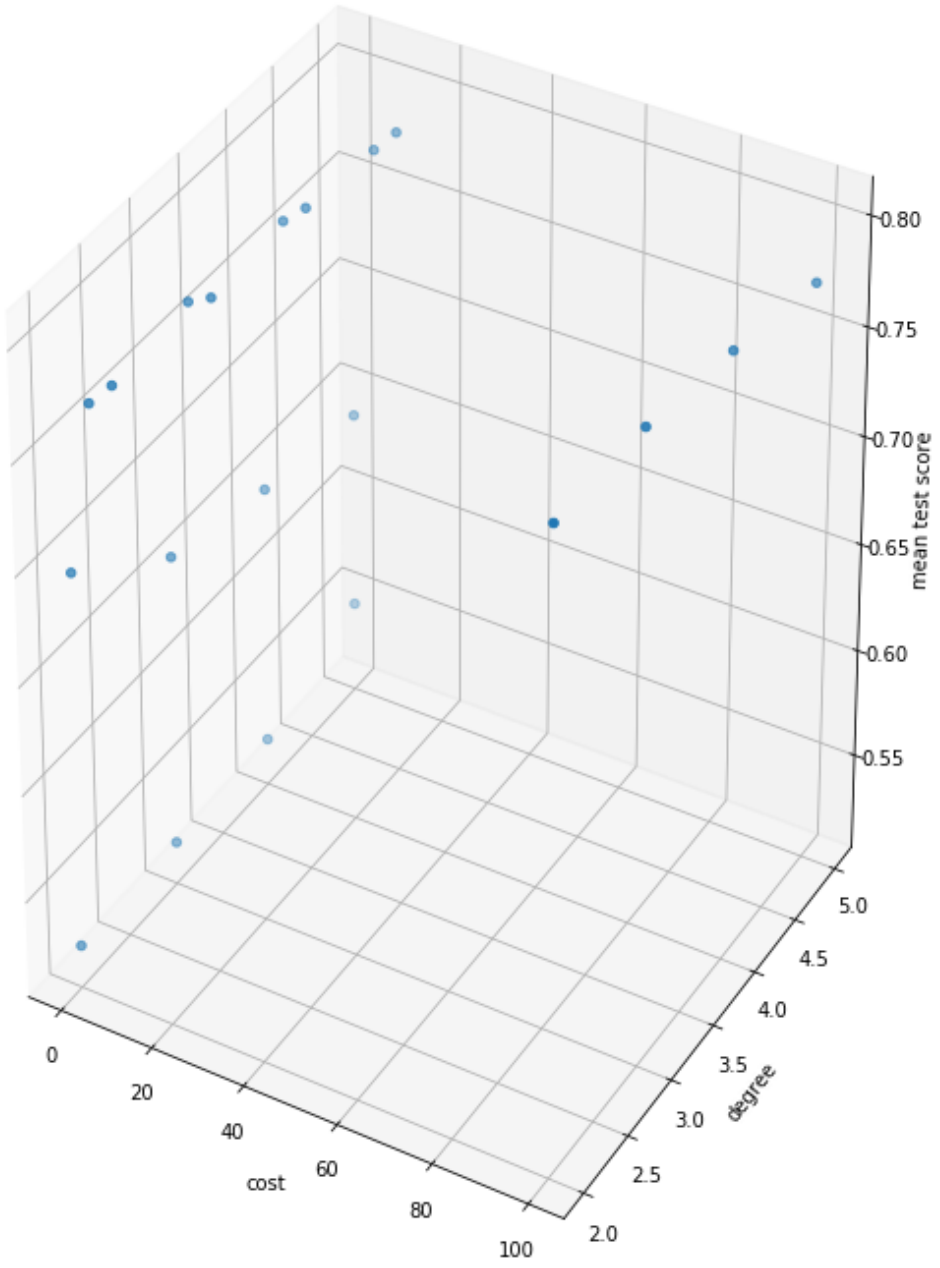
[32]:
```
poly_costs_errors = pd.DataFrame(poly_grid.cv_results_)
poly_costs_errors['mean_cv_errors'] = 1 - poly_costs_errors['mean_test_score']
```

[33]:
```
poly_costs_errors.sort_values(by = "rank_test_score" , ascending =␣
↪True)[['params', 'rank_test_score','mean_test_score','mean_cv_errors']]
```

[33]:

	params	rank_test_score	mean_test_score	mean_cv_errors
16	{'C': 100, 'degree': 2}	1	0.797429	0.202571
17	{'C': 100, 'degree': 3}	2	0.794731	0.205269
12	{'C': 10, 'degree': 2}	3	0.788640	0.211360
13	{'C': 10, 'degree': 3}	4	0.785271	0.214729
18	{'C': 100, 'degree': 4}	5	0.785267	0.214733
14	{'C': 10, 'degree': 4}	6	0.784591	0.215409
9	{'C': 5, 'degree': 3}	7	0.779866	0.220134
15	{'C': 10, 'degree': 5}	8	0.779190	0.220810
8	{'C': 5, 'degree': 2}	9	0.777172	0.222828
10	{'C': 5, 'degree': 4}	10	0.775145	0.224855
19	{'C': 100, 'degree': 5}	11	0.773776	0.226224
11	{'C': 5, 'degree': 5}	12	0.767713	0.232287
4	{'C': 0.1, 'degree': 2}	13	0.698209	0.301791
5	{'C': 0.1, 'degree': 3}	14	0.660389	0.339611
6	{'C': 0.1, 'degree': 4}	15	0.647551	0.352449
7	{'C': 0.1, 'degree': 5}	16	0.639420	0.360580
3	{'C': 0.01, 'degree': 5}	17	0.547597	0.452403
2	{'C': 0.01, 'degree': 4}	18	0.527349	0.472651
1	{'C': 0.01, 'degree': 3}	19	0.525322	0.474678
0	{'C': 0.01, 'degree': 2}	19	0.525322	0.474678

[34]:
```
d3_poly = plt.figure().gca(projection = '3d')
d3_poly.scatter(poly_costs_errors['param_C'],␣
↪poly_costs_errors['param_degree'], poly_costs_errors['mean_test_score'])
d3_poly.set_xlabel('cost')
d3_poly.set_ylabel('degree')
d3_poly.set_zlabel('mean test score')
plt.show()
```



[35]:
```
name = ['linear SVC', 'radial SVM','poly SVM']
Cs = [svc_grid.best_params_['C'], radial_grid.best_params_['C'], poly_grid.
 ↪best_params_['C']]
scores = [svc_grid.best_score_, radial_grid.best_score_, poly_grid.best_score_]
data = {'kernels': name, 'cost_value': Cs, 'best_score': scores}
```

```
comparison = pd.DataFrame(data)
comparison.sort_values(by = "best_score" , ascending = False)
```

[35]:

	kernels	cost_value	best_score
2	poly SVM	100.0	0.797429
0	linear SVC	0.1	0.796762
1	radial SVM	5.0	0.742731

According to the above analysis, we can see that for this dataset, the best cost value for linear SVC is 0.1, the best cost value for radial SVM is 5 and the best cost value for poly SVM is 100 (in the ranges that we have tested for cost value). After comparing their accuracy scores after 10-fold cross validation, we prefer to say that poly SVM with cost value of 100 is the best model to predict colrac. In addition, we can see that the best scores whether of polynomial SVM, radial SVM or linear SVC are relatively high. This indicates that SVM or SVC can be a good model in this case to make the prediction.