Wang_Miaohan_HW6

March 8, 2020

[1]:
```
import pandas as pd
import numpy as np
from sklearn.preprocessing import scale
from sklearn import svm
from sklearn.datasets import make_moons, make_blobs
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score, train_test_split,␣
↪GridSearchCV
```

[2]:
```
seed = 1234
alt_seed = 4321
```

Conceptual Exercise

Non-linear separation

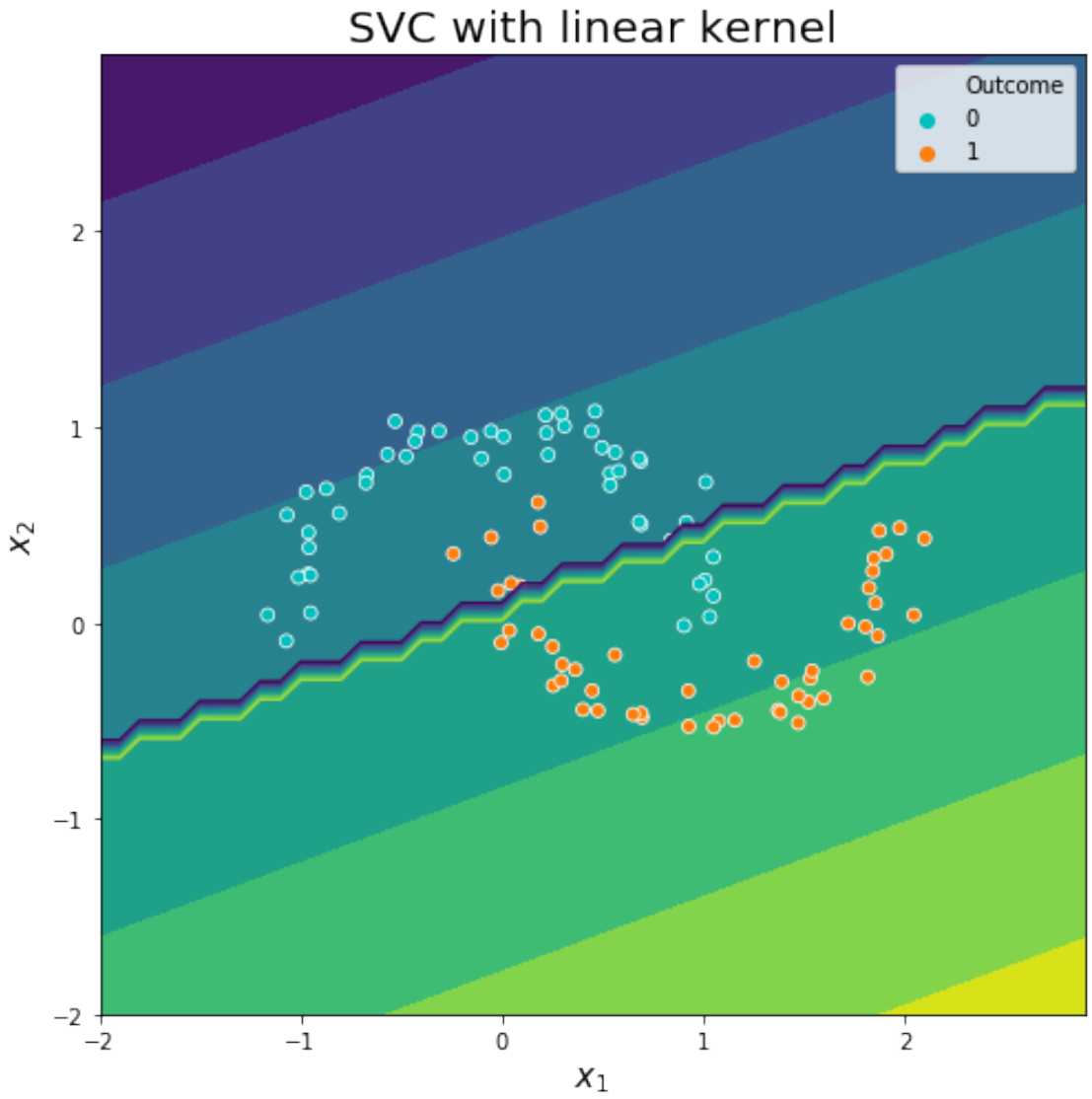
[3]:
```
# Initializing dataset
X_train, y_train = make_moons(n_samples=100, noise=0.1, random_state=seed)
X_test, y_test = make_moons(n_samples=100, noise=0.1, random_state=alt_seed)
```

[4]:
```
def plot_svm(mod, X, y): # function customized for plotting the dataset and the␣
↪decision boundary

    x1 = np.arange(-2, 3, 0.1)
    x2 = np.arange(-2, 3, 0.1)
    xx1, xx2 = np.meshgrid(x1, x2, sparse=False)
    Xgrid = np.stack((xx1.flatten(), xx2.flatten())).T

    # Get decision boundary values for plot grid
    decision_boundary = mod.predict(Xgrid)
    decision_boundary_grid = decision_boundary.reshape(len(x2), len(x1))
```

```
    # Get decision function values for plot grid
    decision_function = mod.decision_function(Xgrid)
    decision_function_grid = decision_function.reshape(len(x2), len(x1))

    fig = plt.figure(figsize=(8, 8))
    plt.contourf(x1, x2, decision_function_grid);
    plt.contour(x1, x2, decision_boundary_grid);
    plt.title(f"SVC with {mod.get_params()['kernel']} kernel", fontsize=20)

    df = pd.concat([pd.DataFrame(data=X, columns=['x_1', 'x_2']), pd.Series(y,␣
↪name='Outcome')], axis=1)
    sns.scatterplot(x='x_1', y='x_2', hue='Outcome', data=df, palette=['c',␣
↪'tab:orange'])
    plt.xlabel('$x_1$', fontsize=15)
    plt.ylabel('$x_2$', fontsize=15)

    plt.show();
```

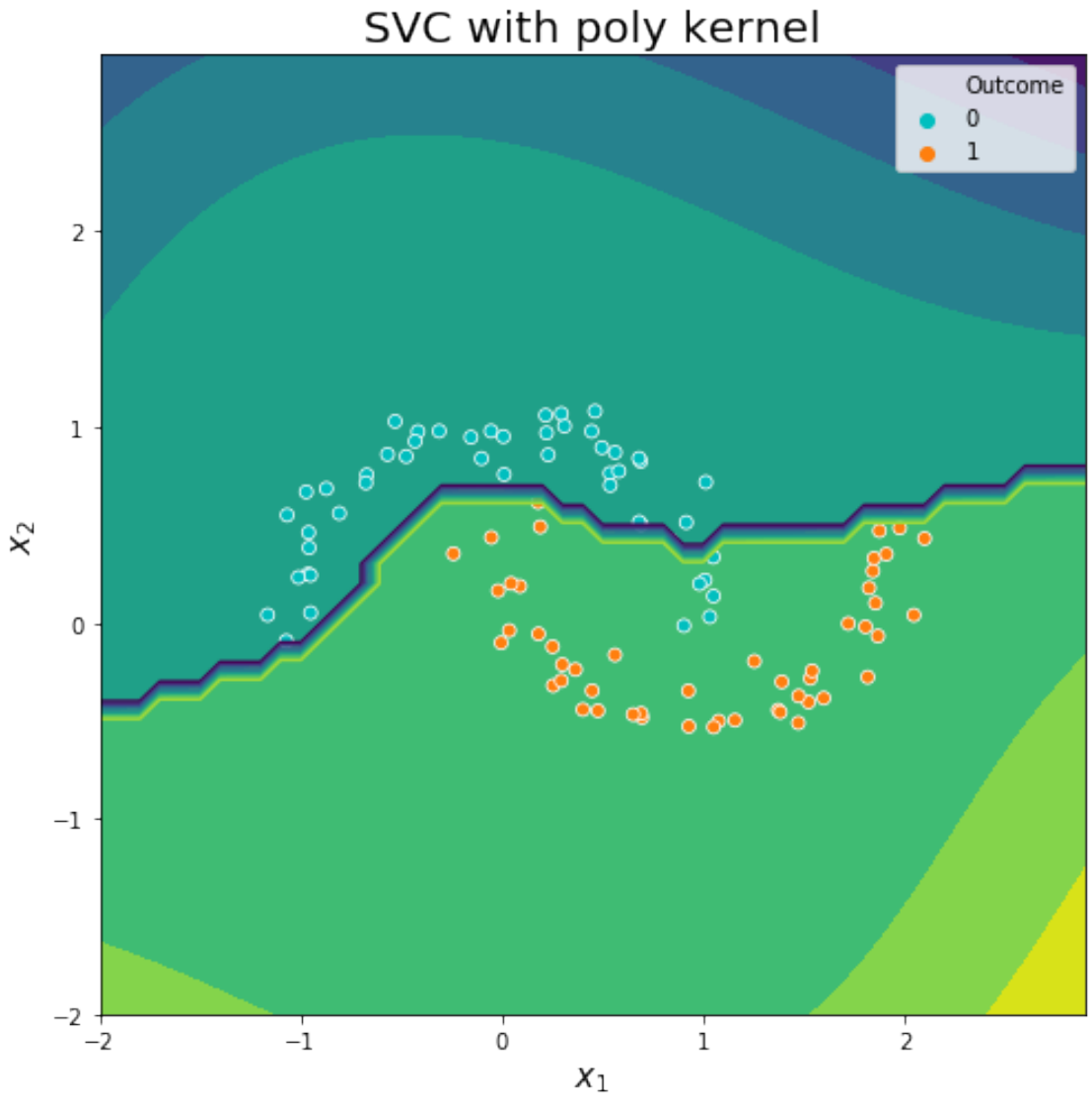
[5]:
```
mod_linear = SVC(kernel='linear', random_state=seed).fit(X_train, y_train)

print('With a linear kernel,')
print(f'the training error rate is {round(1 - mod_linear.score(X_train,␣
↪y_train), 5)}.')
print(f'the testing error rate is {round(1 - mod_linear.score(X_test, y_test),␣
↪5)}.')

plot_svm(mod_linear, X_train, y_train)
```

```
With a linear kernel,
the training error rate is 0.13.
the testing error rate is 0.16.
```



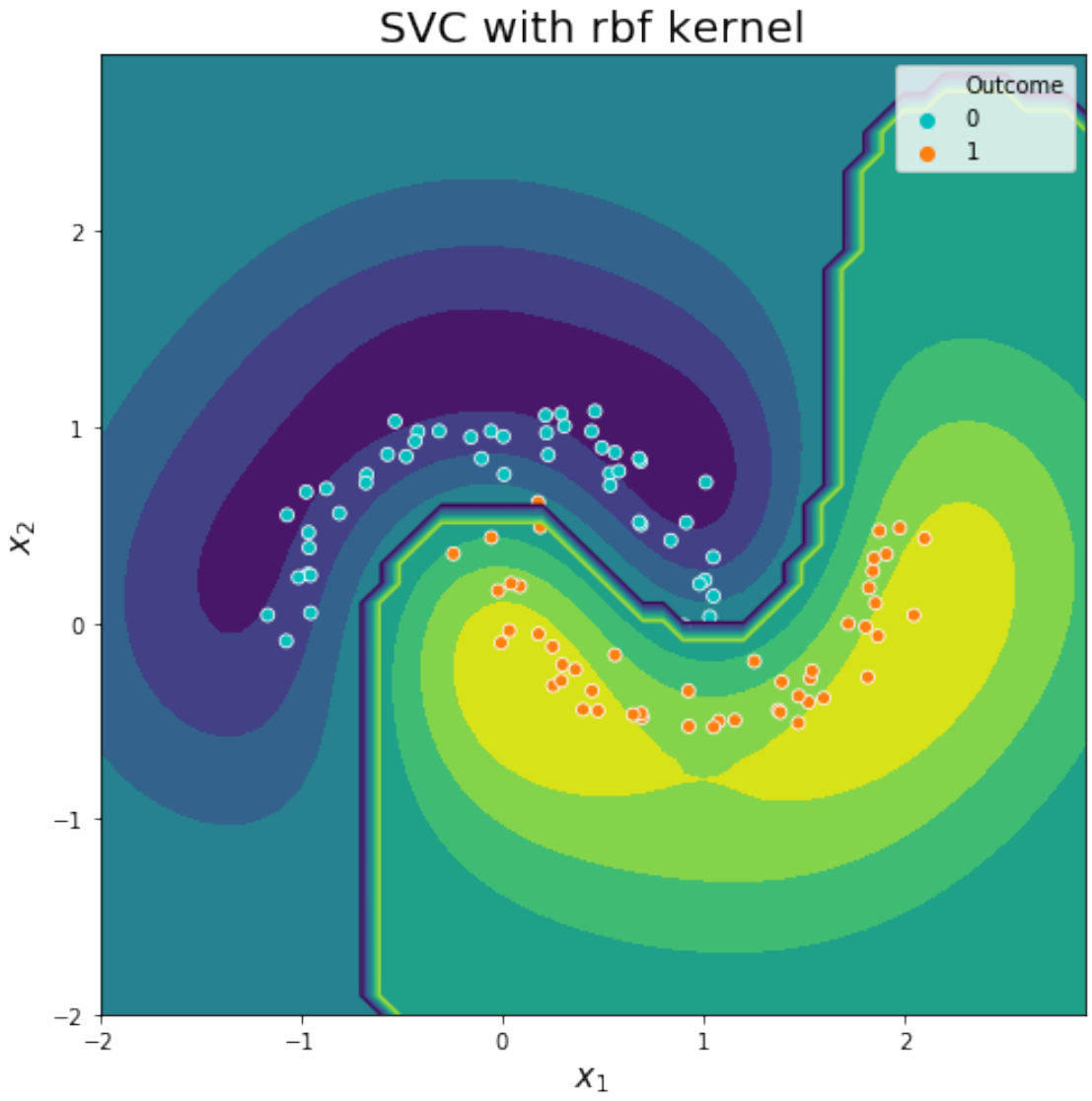
[6]:
```
mod_poly = SVC(kernel='poly', degree=3, random_state=seed).fit(X_train, y_train)

print('With a polynomial kernel (degree=3),')
print(f'the training error rate is {round(1 - mod_poly.score(X_train, y_train),␣
↪5)}.')
print(f'the testing error rate is {round(1 - mod_poly.score(X_test, y_test),␣
↪5)}.')

plot_svm(mod_poly, X_train, y_train)
```

```
With a polynomial kernel (degree=3),
the training error rate is 0.06.
the testing error rate is 0.07.
```



[7]:
```
mod_radial = SVC(kernel='rbf', random_state=seed).fit(X_train, y_train)

print('With a radial kernel,')
print(f'the training error rate is {round(1 - mod_radial.score(X_train,␣
 ↪y_train), 5)}.')
print(f'the testing error rate is {round(1 - mod_radial.score(X_test, y_test),␣
 ↪5)}.')

plot_svm(mod_radial, X_train, y_train)
```

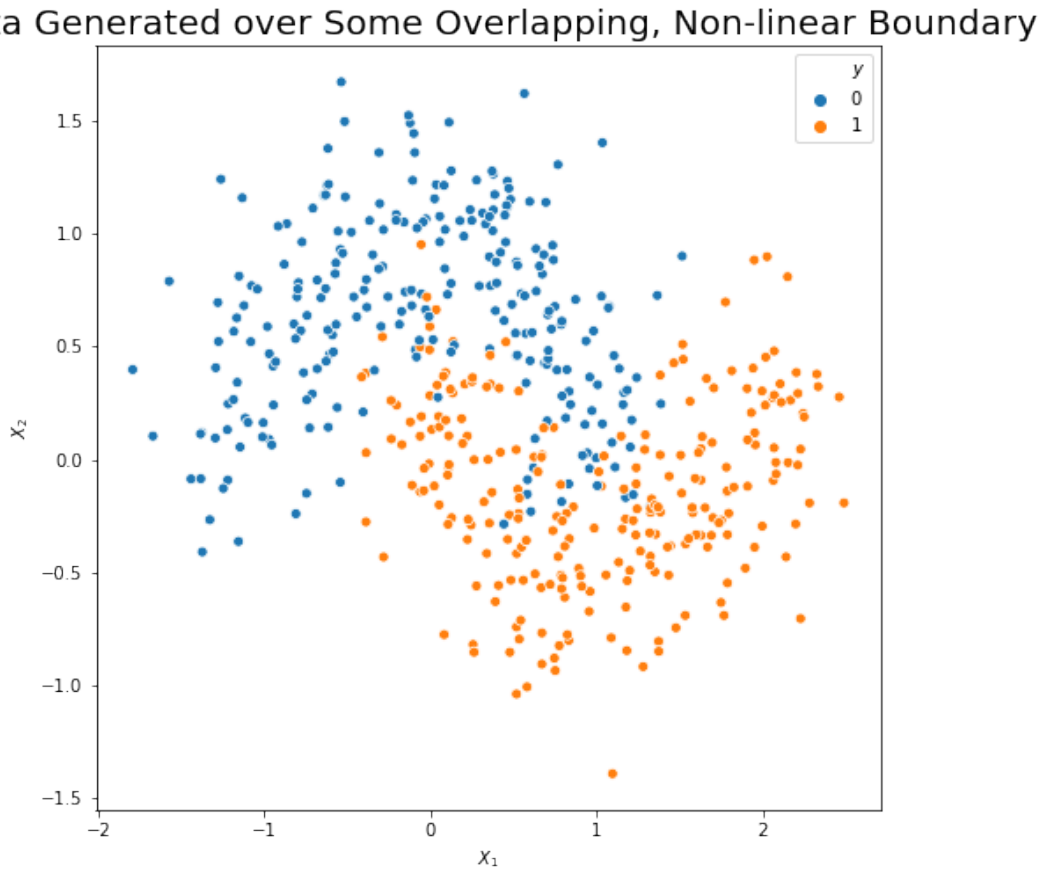
```
With a radial kernel,
the training error rate is 0.01.
the testing error rate is 0.02.
```



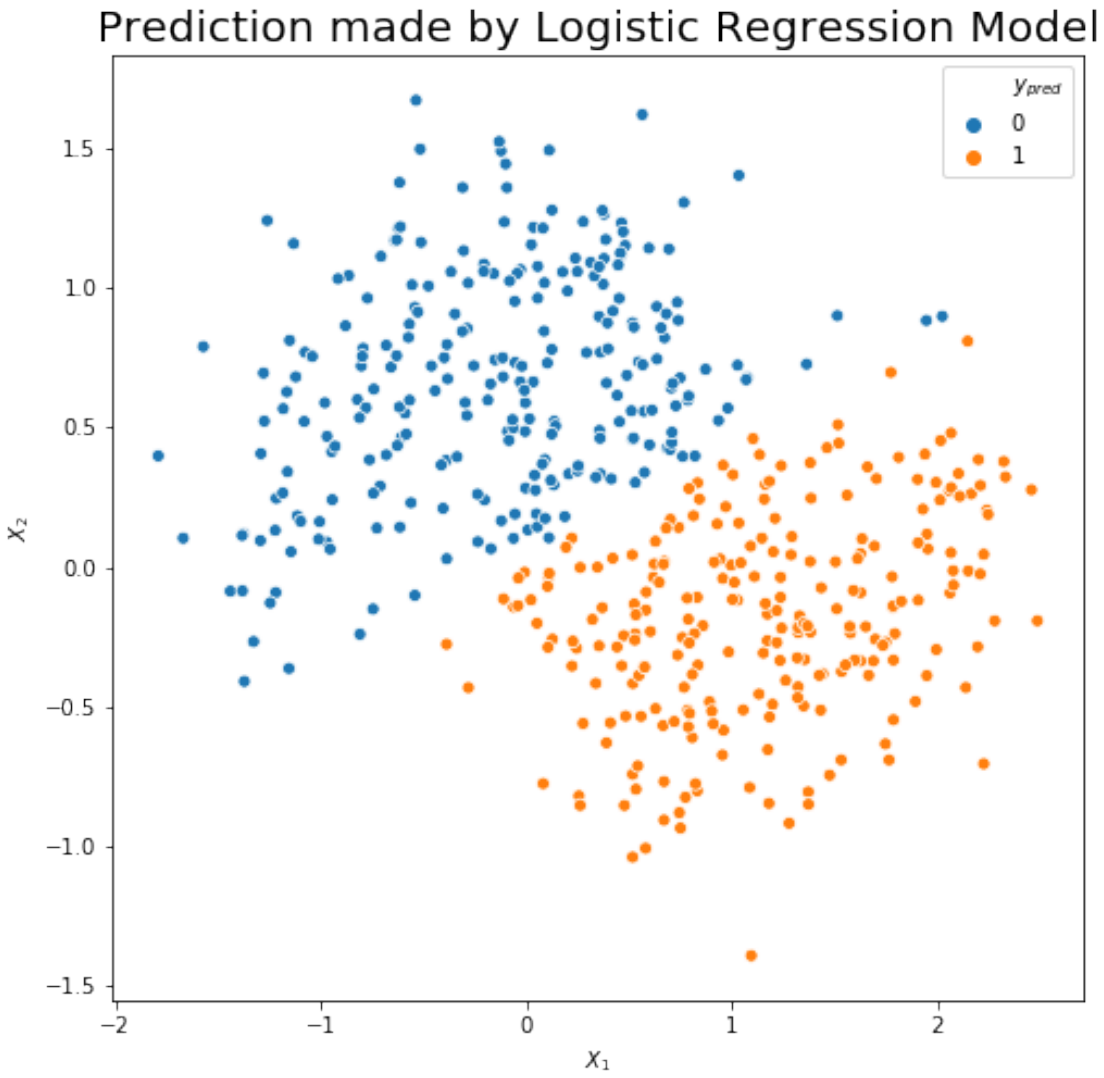
The best perfoming model is the SVM with the radial kernel, it has the lowest test error as well as train error rates. (Performance: radial > polynomial > linear) According to the graph, the SVM with the radial kernel recovers the original distribition of data the best, which is two interleaving half circles. Such performance results from the fact that radial kernel draw decision boundary by Euclidean distance from test data to nearby train data. Hence, it does not assume that the data can be linearly (or polynomially) separated, so the radial kernel SVM can fit non-linear (or further, non-polynomial) relationships much better than linear and polynomial kernels.

SVM vs. logistic regression

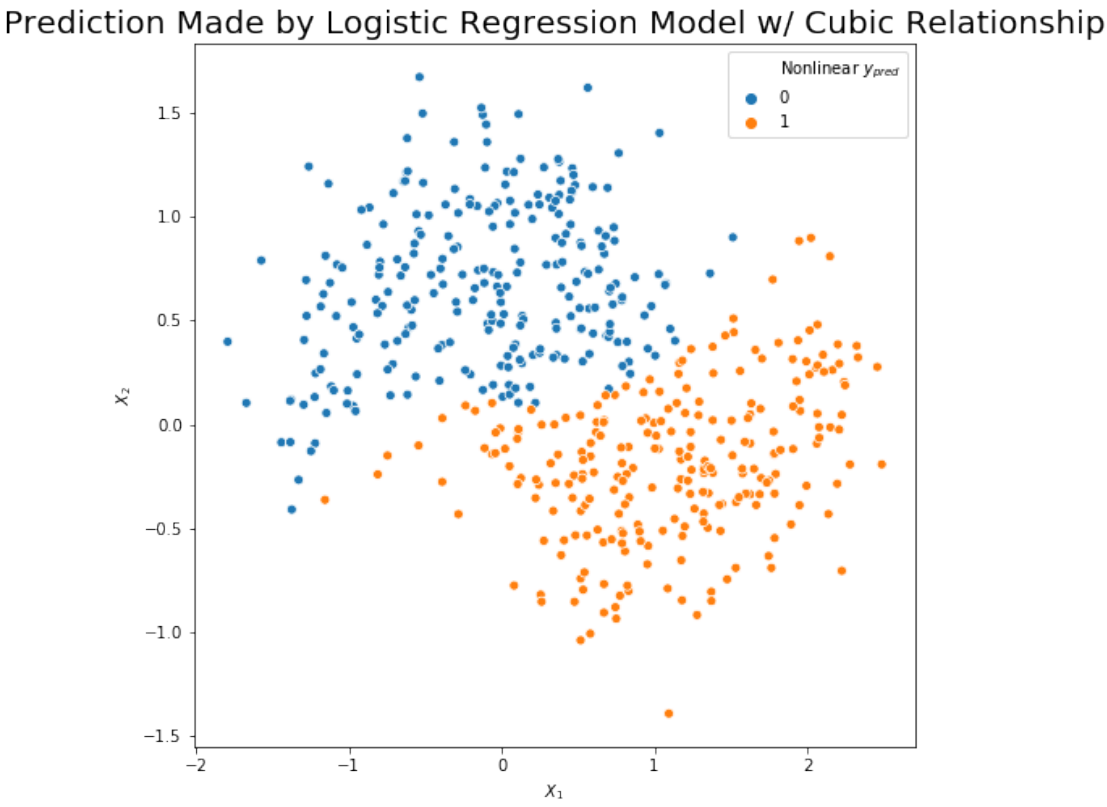
[8]:
```
X, y = make_moons(n_samples=500, noise=0.3, random_state=seed)
df = pd.concat([pd.DataFrame(data=X, columns=['$X_1$', '$X_2$']), pd.Series(y,␣
↪name='$y$')], axis=1)
plt.figure(figsize=(8,8))
sns.scatterplot(x='$X_1$', y='$X_2$', hue='$y$', data=df)
plt.title('Data Generated over Some Overlapping, Non-linear Boundary',␣
↪fontsize=20);
```



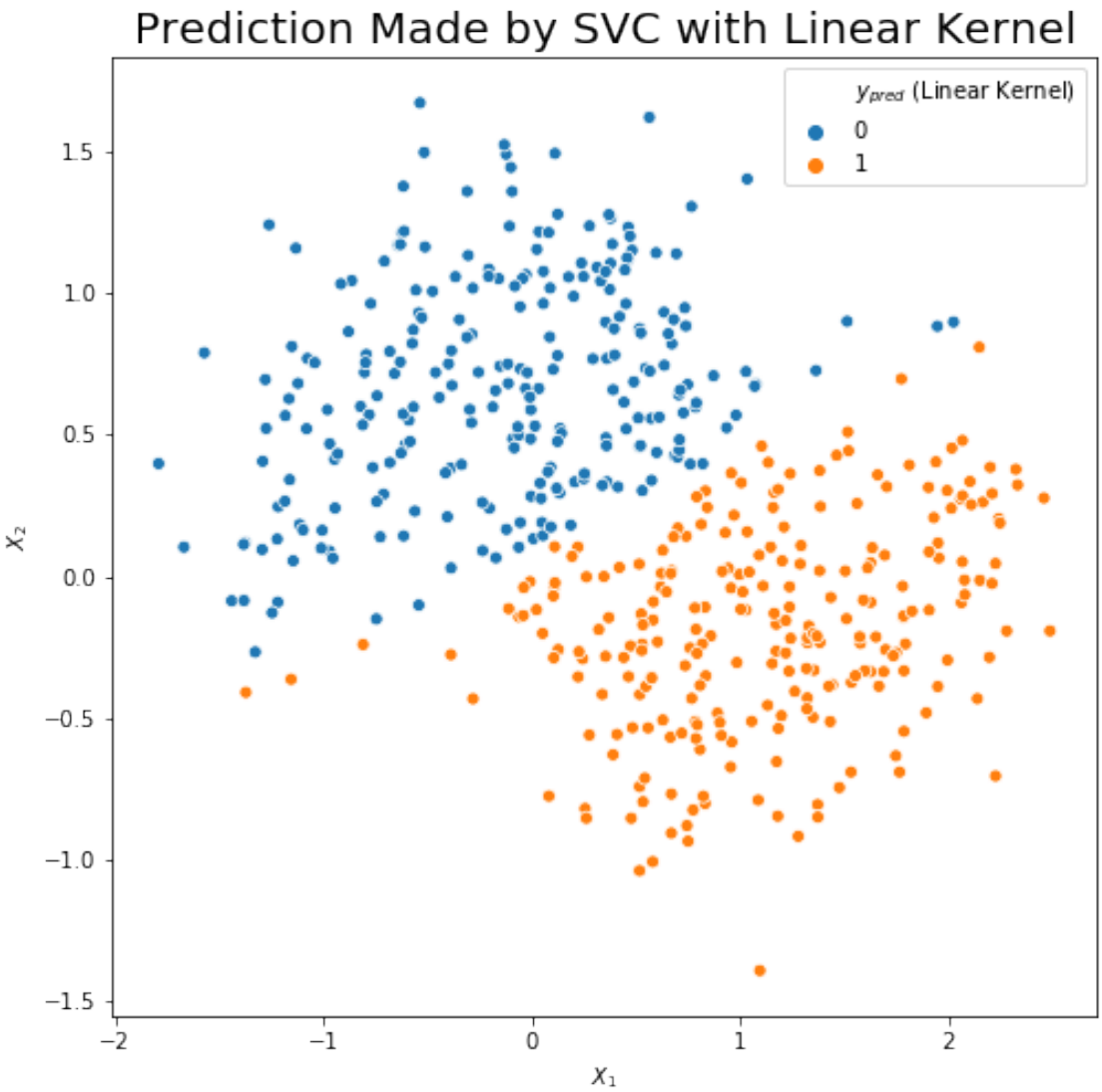
[9]:
```
log_mod = LogisticRegression(random_state=seed).fit(X, y)
df['$y_{pred}$'] = log_mod.predict(X)
plt.figure(figsize=(8,8))
sns.scatterplot(x='$X_1$', y='$X_2$', hue='$y_{pred}$', data=df)
plt.title('Prediction made by Logistic Regression Model', fontsize=20);
```



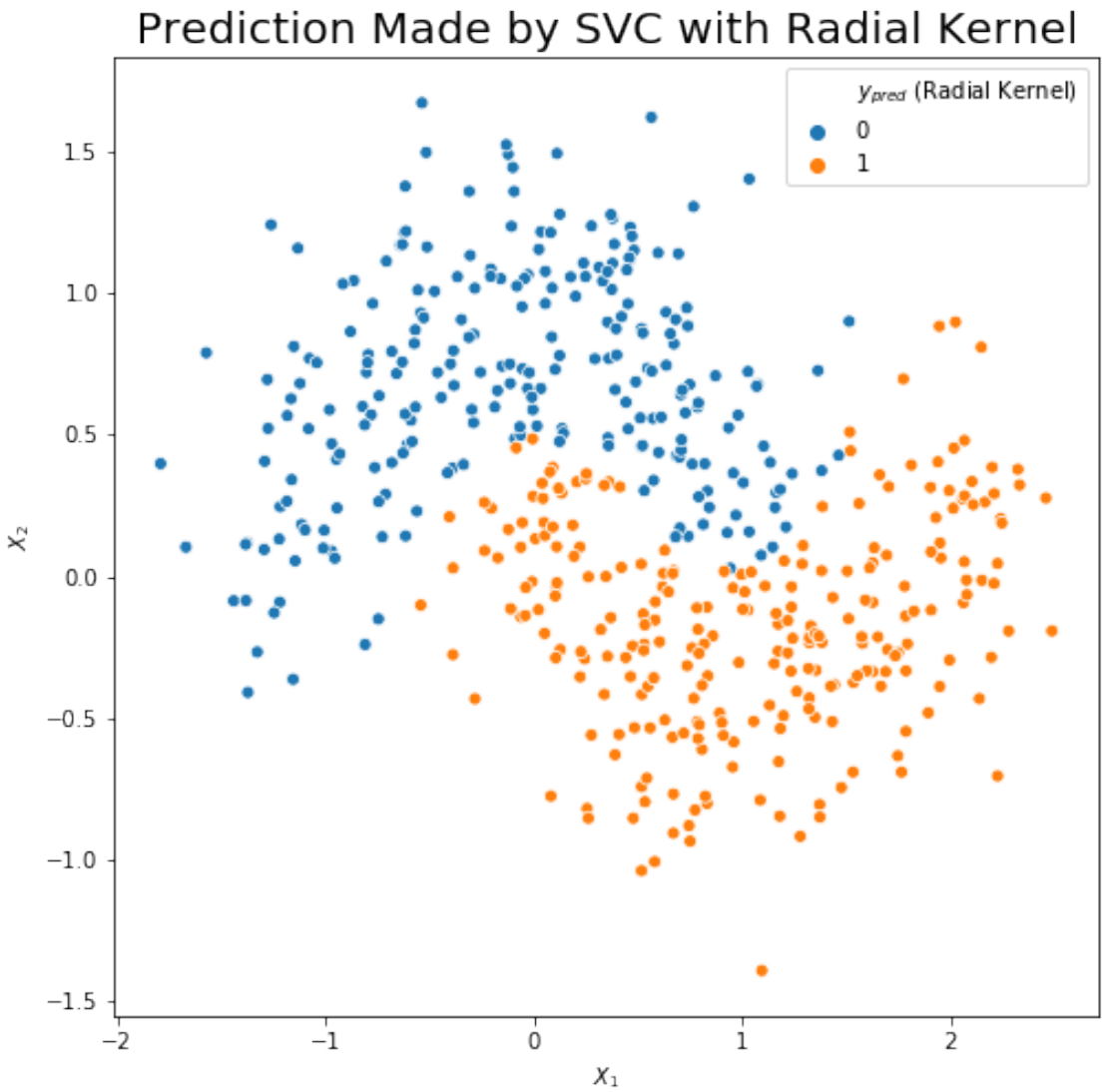
[10]:
```
df['$X_1^3$'] = df['$X_1$'] ** 3
df['$X_2^3$'] = df['$X_2$'] ** 3

X_new = df[['$X_1$', '$X_2$', '$X_1^3$', '$X_2^3$']]
log_mod_nl = LogisticRegression(random_state=seed).fit(X_new, y)
df['Nonlinear $y_{pred}$'] = log_mod_nl.predict(X_new)

plt.figure(figsize=(8,8))
sns.scatterplot(x='$X_1$', y='$X_2$', hue='Nonlinear $y_{pred}$', data=df)
plt.title('Prediction Made by Logistic Regression Model w/ Cubic Relationship',␣
↪fontsize=20);
```



[11]:
```
svc_linear = SVC(kernel='linear', random_state=seed).fit(X, y)
svc_df = pd.concat([pd.DataFrame(data=X, columns=['$X_1$', '$X_2$']), pd.
 ↪Series(y, name='$y$')], axis=1)
svc_df['$y_{pred}$ (Linear Kernel)'] = svc_linear.predict(X)

plt.figure(figsize=(8,8))
plt.title('Prediction Made by SVC with Linear Kernel', fontsize=20)
sns.scatterplot(x='$X_1$', y='$X_2$', hue='$y_{pred}$ (Linear Kernel)',␣
 ↪data=svc_df);
```



[12]:
```
svc_radial = SVC(kernel='rbf', random_state=seed).fit(X, y)
svc_df['$y_{pred}$ (Radial Kernel)'] = svc_radial.predict(X)

plt.figure(figsize=(8,8))
plt.title('Prediction Made by SVC with Radial Kernel', fontsize=20)
sns.scatterplot(x='$X_1$', y='$X_2$', hue='$y_{pred}$ (Radial Kernel)',␣
↪data=svc_df);
```



[13]:
```
pd.DataFrame.from_dict({'Logistic Linear': 1 - log_mod.score(X, y),
                        'Logistic Non-linear': 1 - log_mod_nl.score(X_new, y),
                        'SVC Linear': 1 - svc_linear.score(X, y),
                        'SVC Radial': 1 - svc_radial.score(X, y)},␣
↪orient='index', columns=['Error Rate'])
```

[13]:

	Error Rate
Logistic Linear	0.156
Logistic Non-linear	0.138
SVC Linear	0.160
SVC Radial	0.078

We observe that when making prediction about data with non-linear relationship, linear logistic regression outperforms linear kernel SVC, showing that linear logistic regression performs fit a

better regression line than linear kernel SVC when meeting non-linear data. But ultimately, the radial kernel SVC gives the best error rate, because the data we generated is from two interleaving half circle, it cannot perfectly separated by a polynomial boundary. Therefore the non-linear logistic regression, limited by clear functional form, doesn't have as great a performance as radial kernel SVC does.

When we don't know the underlying relationships of a dataset, if we aim for predictive accuracy, we should choose radial kernel since it can process very complex non-linear relationships. But if we aim for an explanatory model, non-linear logistic regression will be more easily interpretable.

Tuning cost

[14]:
```
Xc, _ = make_blobs(n_samples=100, n_features=2, centers=1, random_state=seed)
yc = np.zeros(100)
yc[np.sum(Xc, axis=1) > -3.6] = 1

X_train, X_test, y_train, y_test = train_test_split(Xc, yc, test_size=0.3,␣
↪random_state=seed)

X_train = scale(X_train)
X_test = scale(X_test)
```

[15]:
```
# customized function for SVC with linear kernel, different costs
def svc_linear(cost, X_train, y_train, X_test, y_test):
    mod = SVC(kernel='linear', C=cost, random_state=seed)
    cv_train_err = 1 - cross_val_score(mod, X_train, y_train,␣
↪scoring='accuracy', cv=10).mean()
    mod.fit(X_train, y_train)
    train_err = 1 - mod.score(X_train, y_train)
    test_err = 1 - mod.score(X_test, y_test)

    return [cost, train_err, cv_train_err, test_err]
```

[16]:
```
# drawing a table of errors with different costs
cost_df = []

for cost in [1e-05, 1e-04, 1e-03, 1e-01, 1, 1e01, 1e02, 1e03, 1e04, 1e05]:
    cost_df.append(svc_linear(cost, X_train, y_train, X_test, y_test))

cost_df = pd.DataFrame(cost_df, columns=['Cost', 'Train Error', '10-Fold CV␣
↪Train Error', 'Test Error'])
cost_df
```

[16]:

	Cost	Train Error	10-Fold CV Train Error	Test Error
0	0.00001	0.428571	0.428571	0.466667
1	0.00010	0.428571	0.428571	0.466667

```
2       0.00100     0.428571                0.428571    0.466667
3       0.10000     0.014286                0.028571    0.100000
4       1.00000     0.000000                0.014286    0.100000
5      10.00000     0.000000                0.014286    0.066667
6     100.00000     0.000000                0.000000    0.066667
7    1000.00000     0.000000                0.000000    0.066667
8   10000.00000     0.000000                0.000000    0.066667
9  100000.00000     0.000000                0.000000    0.066667
```

The training error reaches 0 at $C = 1$, but the cross-validated training error only reaches 0 at $C = 100$. On the side of test error, we see its gradual reduction as cost value rises, ultimately stablized after $C = 10$. This shows that when tuning for cost value in a SVC model, we don't need a extremely large cost value to yield the optimal model. Higher cost would not lower the test error, it will only cause overfitting and the model will not be improved. In this case, $C = 100$ is perfoming as good as any SVC model with a larger cost value. Hence, we don't need to endlessly enlarge cost value for model accuracy, which also saves us computation time.

Application Exercise

[17]:
```
train_df = pd.read_csv('data/gss_train.csv')
test_df = pd.read_csv('data/gss_test.csv')

y_train = train_df.colrac
y_test = test_df.colrac
X_train = train_df.drop('colrac', axis=1)
X_test = test_df.drop('colrac', axis=1)
```

[18]:
```
# Customized function for fitting SVC model
def svc_mod(X_train, y_train, kernel='linear', C=1, degree=3, gamma='scale'):
    mod = SVC(C=C, kernel=kernel, degree=degree, gamma=gamma, random_state=seed)
    cv_err = 1 - cross_val_score(mod, X_train, y_train, scoring='accuracy',␣
 ↪cv=10).mean()
    result = []
    if kernel == 'linear':
        result = [C, cv_err]
    elif kernel == 'poly':
        result = [C, degree, gamma, cv_err]
    elif kernel == 'rbf':
        result = [C, gamma, cv_err]
    return result
```

Linear Kernel

[19]:
```
costs = [1e-05, 1e-04, 1e-03, 1e-02, 1e-01, 1]
linear_results = [svc_mod(X_train, y_train, kernel='linear', C=c) for c in␣
↪costs]

pd.DataFrame(linear_results, columns=['Cost', '10-fold cv train error'])
```

[19]:

	Cost	10-fold cv train error
0	0.00001	0.474678
1	0.00010	0.303138
2	0.00100	0.239702
3	0.01000	0.205265
4	0.10000	0.203238
5	1.00000	0.205931

With a linear kernel SVC, the best 10-fold cv training error is achieved at $C = 0.1$. Hence, we see that with a linear kernel SVC, the cost should not be infinitely high, otherwise it would cause overfitting.

Polynomial Kernel

[38]:
```
poly_params = {'C': [0.001, 0.01, 0.1, 1],
               'degree': [1, 2, 3, 4, 5, 6],
               'gamma': [1e-02, 1e-01, 1]}

poly_search = GridSearchCV(SVC(kernel='poly', random_state=seed), poly_params,␣
↪scoring='accuracy', cv=10)
poly_search.fit(X_train, y_train)

poly_search.best_estimator_
```

[38]:
```
SVC(C=0.001, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=2, gamma=0.1, kernel='poly',
    max_iter=-1, probability=False, random_state=1234, shrinking=True,
    tol=0.001, verbose=False)
```

[39]:
```
poly_result = pd.DataFrame.from_dict(poly_search.cv_results_)[['param_C',␣
↪'param_degree', 'param_gamma', 'mean_test_score', 'rank_test_score']]

poly_result.sort_values(by='rank_test_score').head()
```

[39]:

	param_C	param_degree	param_gamma	mean_test_score	rank_test_score
4	0.001	2	0.1	0.798104	1
39	0.1	2	0.01	0.798104	1
55	1	1	0.1	0.796762	3

38	0.1	1	1	0.796762	3
37	0.1	1	0.1	0.795411	5

With a polynomial kernel, the best model is found at $C = 0.001$, $d = 2$ and $\gamma = 0.1$ or $C = 0.1$, $d = 2$ and $\gamma = 0.01$. The polynomial kernel SVC again shows that cost does not have to be extremely high to guarantee an accurate model, a cost too high only causes overfitting instead of bringing a more predictive model.

Radial Kernel

[34]:
```
radial_params = {'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000],
                 'gamma': [1e-05, 1e-04, 1e-03, 1e-02, 1e-01, 1, 10]}

radial_search = GridSearchCV(SVC(kernel='rbf', random_state=seed),␣
 ↪radial_params, scoring='accuracy', cv=10)
radial_search.fit(X_train, y_train)

radial_search.best_estimator_
```

[34]:
```
SVC(C=1000, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma=1e-05, kernel='rbf',
    max_iter=-1, probability=False, random_state=1234, shrinking=True,
    tol=0.001, verbose=False)
```

[40]:
```
radial_result = pd.DataFrame.from_dict(radial_search.cv_results_)[['param_C',␣
 ↪'param_gamma', 'mean_test_score', 'rank_test_score']]

radial_result.sort_values(by='rank_test_score').head()
```

[40]:

	param_C	param_gamma	mean_test_score	rank_test_score
42	1000	1e-05	0.794735	1
43	1000	0.0001	0.794069	2
36	100	0.0001	0.790681	3
30	10	0.001	0.785271	4
35	100	1e-05	0.774469	5

The radial kernel SVC behaves differently when compared to linear or polynomial kernel SVC. The cross-validated error rate decreases infinitely as cost increases and gamma decreases. This is because radial kernel SVC does not assume a functional form for the decision boundary. It uses Euclidean distance between a test data point and train data point to compute predicted outcome. Hence, the radial kernel can easily overfit the training data if the cost is too high, and this might not be detectable with 10-fold CV training error rate.

[43]:
```
# Introducing the test set for final validation
mod1 = SVC(C=0.1, kernel='linear', random_state=seed)
mod2 = SVC(C=0.1, kernel='poly', degree=2, gamma=0.01, random_state=seed)
mod3 = SVC(C=1000, kernel='rbf', gamma=1e-05, random_state=seed)
```

```
mod1.fit(X_train, y_train)
mod2.fit(X_train, y_train)
mod3.fit(X_train, y_train)
comp_dict = {'Linear Kernel': 1-mod1.score(X_test, y_test),
             'Polynomial Kernel': 1-mod2.score(X_test, y_test),
             'Radial Kernel': 1-mod3.score(X_test, y_test)}
pd.DataFrame(comp_dict, index=["Best Model's Test Error Rate"]).T
```

[43]:

	Best Model's Test Error Rate
Linear Kernel	0.219067
Polynomial Kernel	0.196755
Radial Kernel	0.200811

When brought to the test data, we found that the polynomial kernel SVC yielded the lowest test error rate. Therefore, we can safely assume that decision boundary separating 'colrac' outcome in the GSS data very possibly has a polynomial function form of degree 2.

[ ]: