Xu_Weijie_HW6

March 7, 2020

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import svm
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
from warnings import simplefilter
```

[2]:
```
SEED = 970608
```

[3]:
```
# Mute FutureWarning
simplefilter(action='ignore', category=FutureWarning)
```

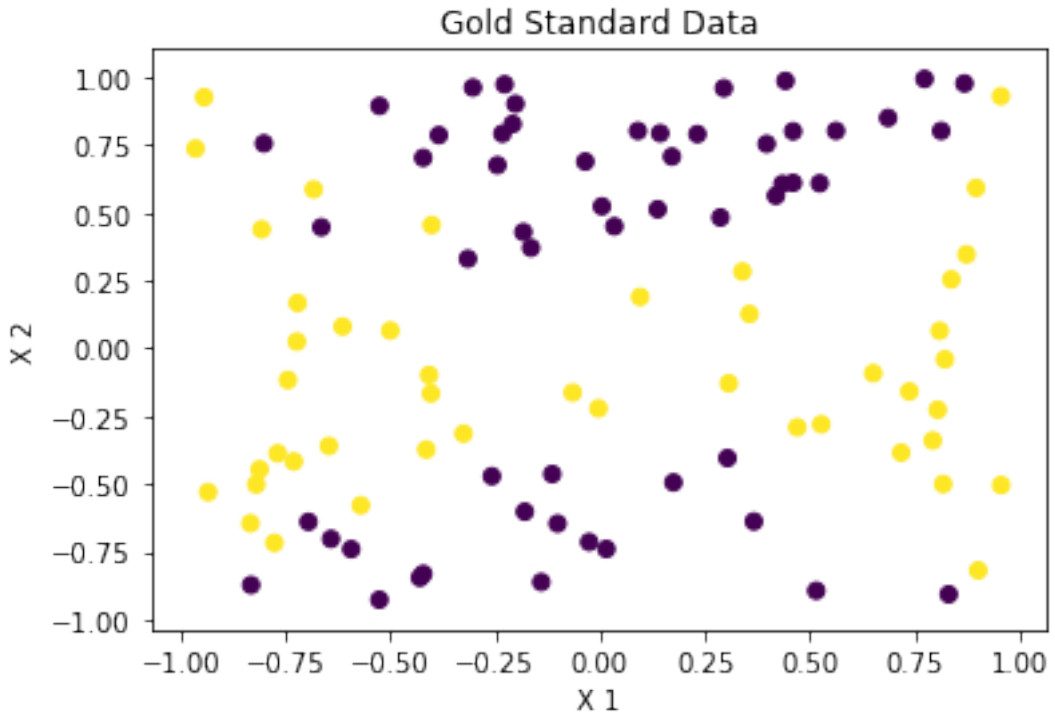
1 Non-linear separation

1.1 Task 1

The SVM with radial kernel achieved a better performance than SVC with linear kernel. Firstly, as we can see from the plots below, the plot with predicted label by SVM looks much more similar to the gold standard labeled data than the linear SVC. All the purple dots in the bottom of the plot of the gold-standard data have been incorrectly classified by SVC, but SVM could successfully pick them out. On the other hand, as for the accuracy rate calculated below, we can easily see that both the training accuracy and the test accuracy of SVM is significantly better than those of SVC. Therefore, SVM performs better than SVC.

[4]:
```
np.random.seed(SEED)
x1 = np.random.uniform(-1, 1, 100)
x2 = np.random.uniform(-1, 1, 100)
y = x1 ** 2 - x2 ** 2 > np.random.normal(0, 0.1, 100)

x1_train, x1_test = x1[:80], x1[80:]
x2_train, x2_test = x2[:80], x2[80:]

X_train, X_test = np.stack((x1_train, x2_train), axis=-1), np.stack((x1_test,␣
↪x2_test), axis=-1)
y_train, y_test = y[:80], y[80:]
```

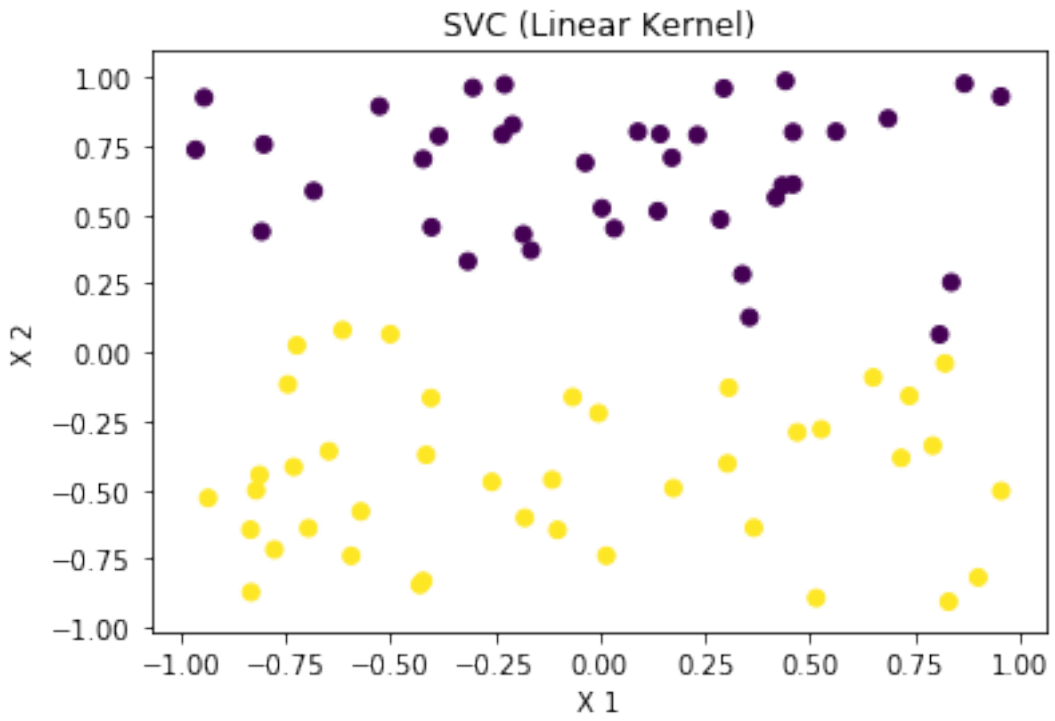
[5]:
```
plt.scatter(x1, x2, c=y)
plt.xlabel('X 1')
plt.ylabel('X 2')
plt.title('Gold Standard Data')
plt.show()
```



[6]:
```
clf_svc_linear = svm.SVC(kernel='linear', gamma=1, probability=True,␣
↪random_state=SEED)
clf_svc_linear.fit(X_train, y_train)
```

[6]:
```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma=1, kernel='linear',
    max_iter=-1, probability=True, random_state=970608, shrinking=True,
    tol=0.001, verbose=False)
```

[7]:
```
y_train_linear_pred = clf_svc_linear.predict(X_train)

plt.scatter(x1_train, x2_train, c=y_train_linear_pred)
plt.xlabel('X 1')
plt.ylabel('X 2')
plt.title('SVC (Linear Kernel)')
plt.show()
```



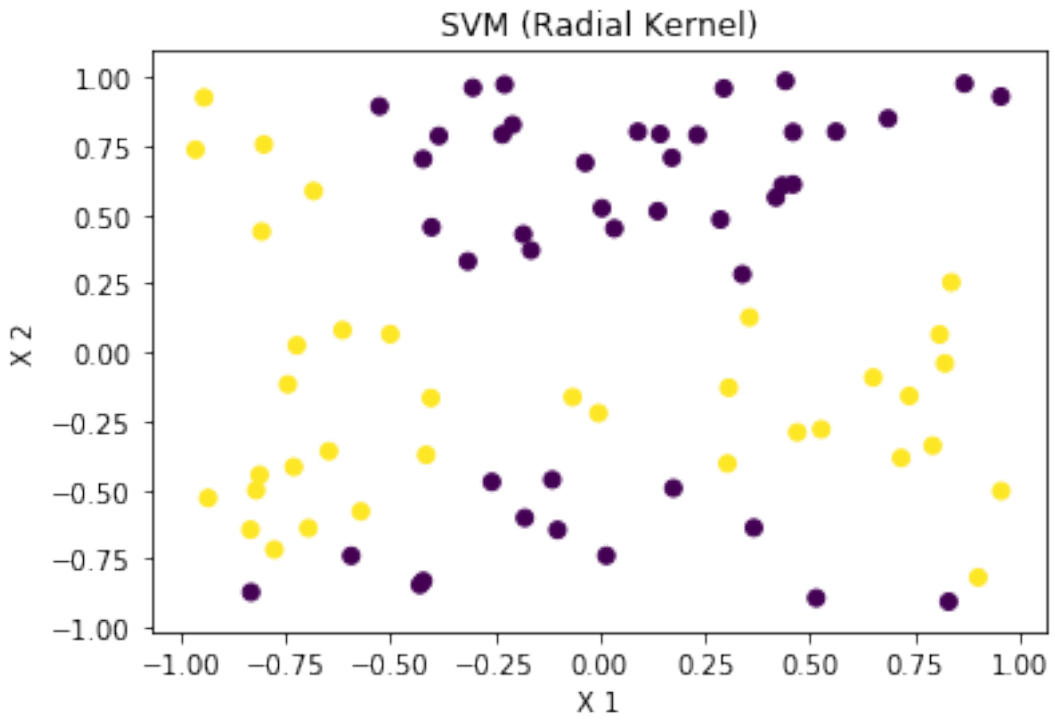
[8]:
```
print(f'Training accuracy of linear kernel: {clf_svc_linear.score(X_train,␣
↪y_train)}')
print(f'Testing accuracy of linear kernel: {clf_svc_linear.score(X_test,␣
↪y_test)}')
```

```
Training accuracy of linear kernel: 0.6875
Testing accuracy of linear kernel: 0.65
```

[9]:
```
clf_svm_radial = svm.SVC(gamma=1, probability=True, random_state=SEED)
clf_svm_radial.fit(X_train, y_train)
```

[9]:
```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma=1, kernel='rbf', max_iter=-1,
    probability=True, random_state=970608, shrinking=True, tol=0.001,
    verbose=False)
```

[10]:
```
y_train_radial_pred = clf_svm_radial.predict(X_train)

plt.scatter(x1_train, x2_train, c=y_train_radial_pred)
plt.xlabel('X 1')
plt.ylabel('X 2')
plt.title('SVM (Radial Kernel)')
plt.show()
```



[11]:
```
print(f'Training accuracy of radial kernel: {clf_svm_radial.score(X_train,␣
↪y_train)}')
print(f'Testing accuracy of radial kernel: {clf_svm_radial.score(X_test,␣
↪y_test)}')
```

```
Training accuracy of radial kernel: 0.925
Testing accuracy of radial kernel: 0.9
```

2 SVM vs. logistic regression

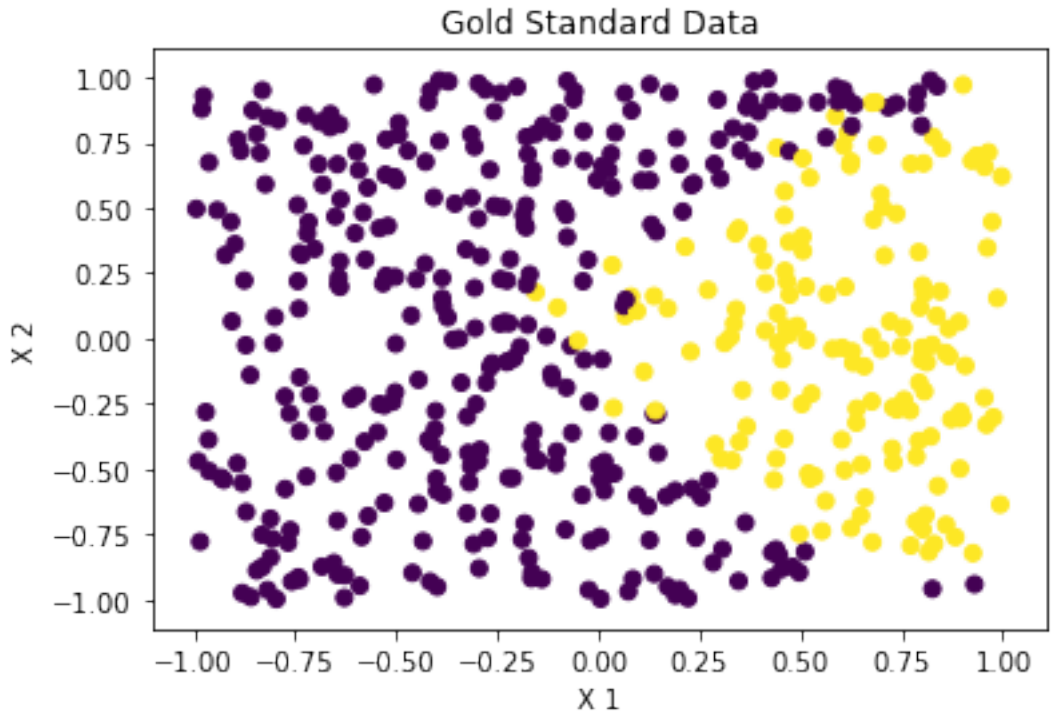
2.1 Task 2

[12]:
```
np.random.seed(SEED)
x1 = np.random.uniform(-1, 1, 500)
x2 = np.random.uniform(-1, 1, 500)
y = x1 - x2 ** 2 > np.random.normal(0, 0.1, 500)
```

2.2 Task 3

[13]:
```
plt.scatter(x1, x2, c=y)
plt.xlabel('X 1')
plt.ylabel('X 2')
plt.title('Gold Standard Data')
```

```
plt.show()
```



2.3 Task 4

[14]:
```
df = pd.DataFrame({'x1':x1, 'x2':x2, 'y':y})
X, y = df[['x1', 'x2']], df['y']
```

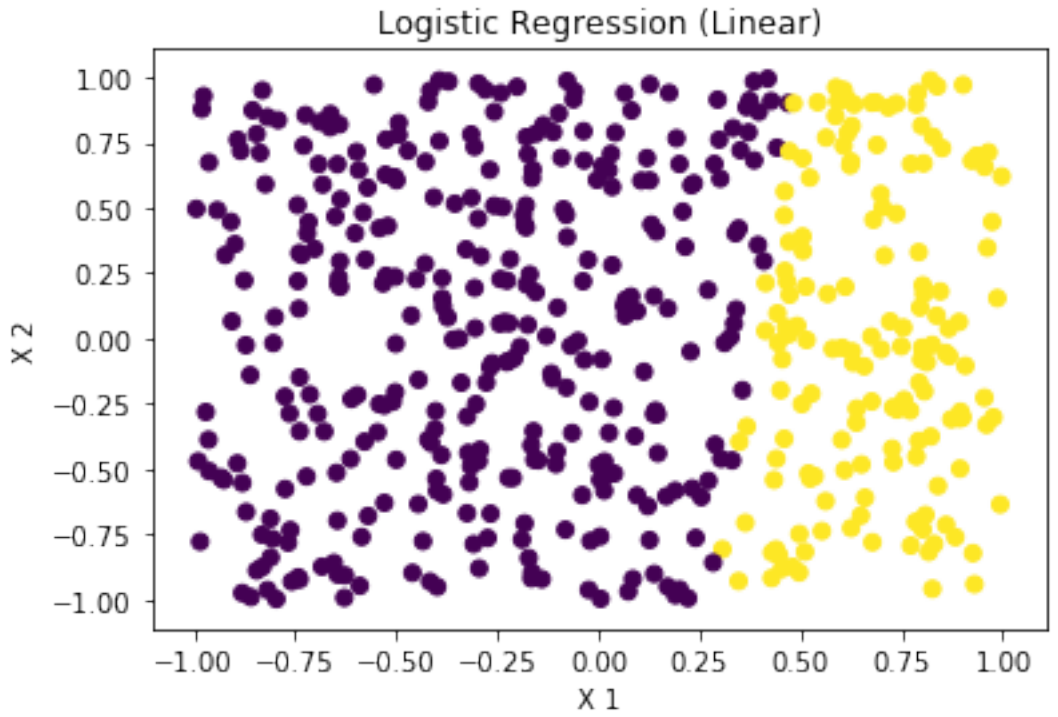
[15]:
```
clf_log1 = LogisticRegression(random_state=SEED)
clf_log1.fit(X, y)
```

[15]:
```
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=970608, solver='warn', tol=0.0001, verbose=0,
                   warm_start=False)
```

2.4 Task 5

[16]:
```
y_log1_pred = clf_log1.predict(X)

plt.scatter(x1, x2, c=y_log1_pred)
plt.xlabel('X 1')
plt.ylabel('X 2')
```

```
plt.title('Logistic Regression (Linear)')
plt.show()
```



2.5 Task 6

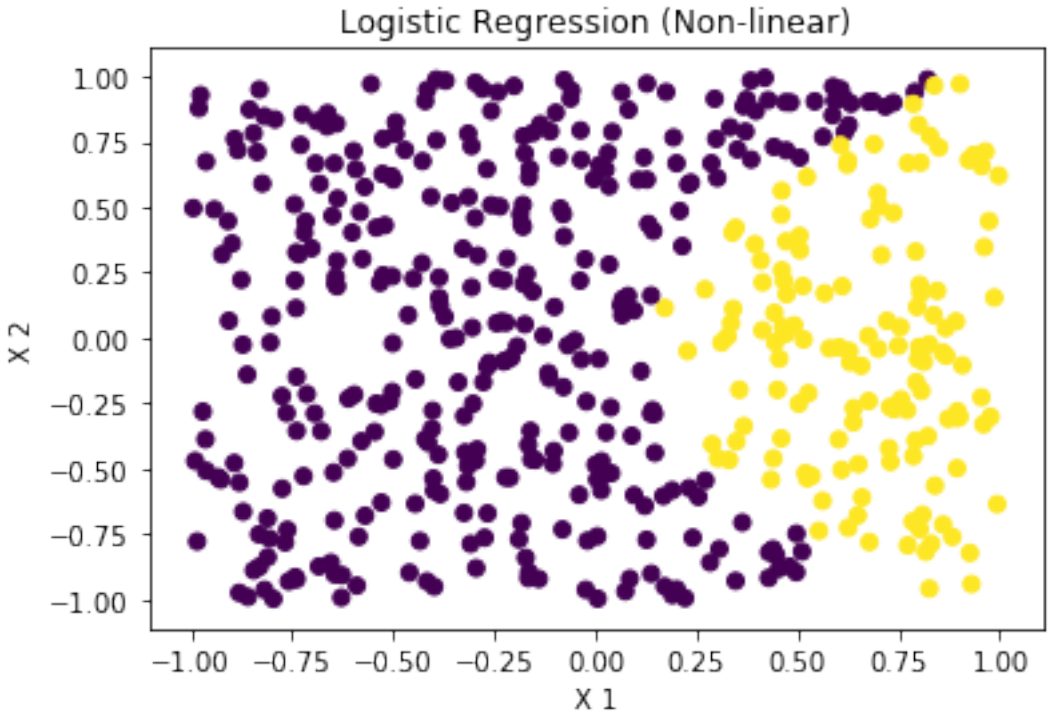
[17]:
```
x1_vander = np.vander(X['x1'], N=2 + 1)[:, :-1]
x2_vander = np.vander(X['x2'], N=2 + 1)
X_nl = np.hstack((x1_vander, x2_vander))
```

[18]:
```
clf_log2 = LogisticRegression(random_state=SEED)
clf_log2.fit(X_nl, y)
```

[18]:
```
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=970608, solver='warn', tol=0.0001, verbose=0,
                   warm_start=False)
```

2.6 Task 7

[19]:
```
y_log2_pred = clf_log2.predict(X_nl)

plt.scatter(x1, x2, c=y_log2_pred)
plt.xlabel('X 1')
plt.ylabel('X 2')
plt.title('Logistic Regression (Non-linear)')
plt.show()
```



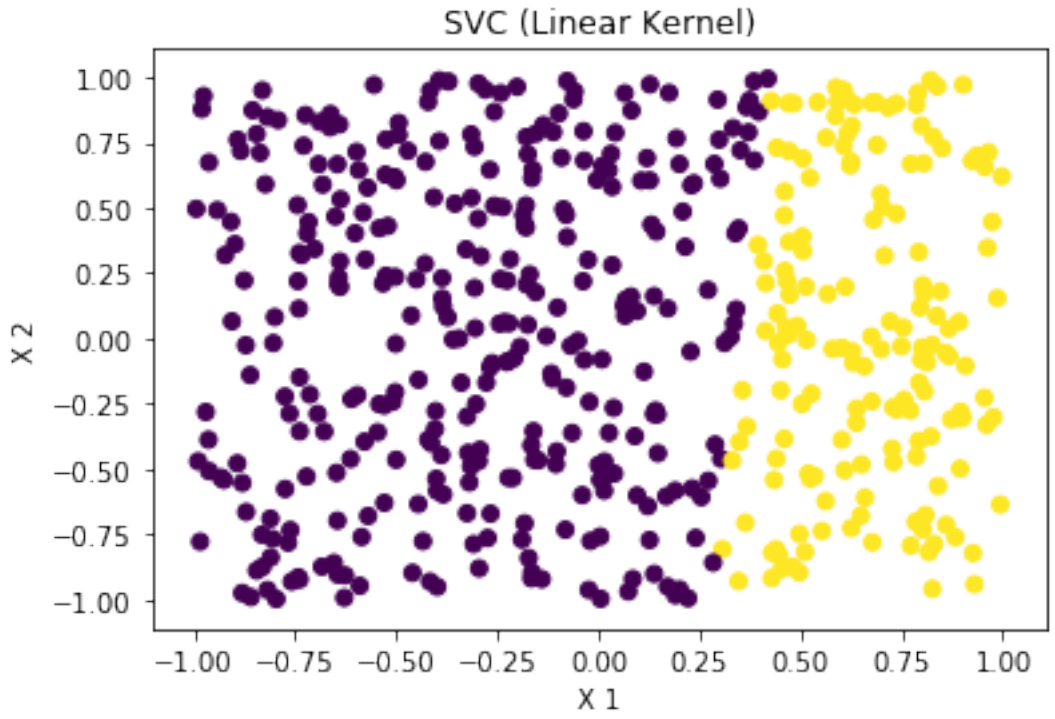
2.7 Task 8

[20]:
```
clf_svm1 = svm.SVC(kernel='linear', gamma=1, probability=True,␣
↪random_state=SEED)
clf_svm1.fit(X, y)
```

[20]:
```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma=1, kernel='linear',
    max_iter=-1, probability=True, random_state=970608, shrinking=True,
    tol=0.001, verbose=False)
```

[21]:
```
y_svm1_pred = clf_svm1.predict(X)

plt.scatter(x1, x2, c=y_svm1_pred)
```

```
plt.xlabel('X 1')
plt.ylabel('X 2')
plt.title('SVC (Linear Kernel)')
plt.show()
```

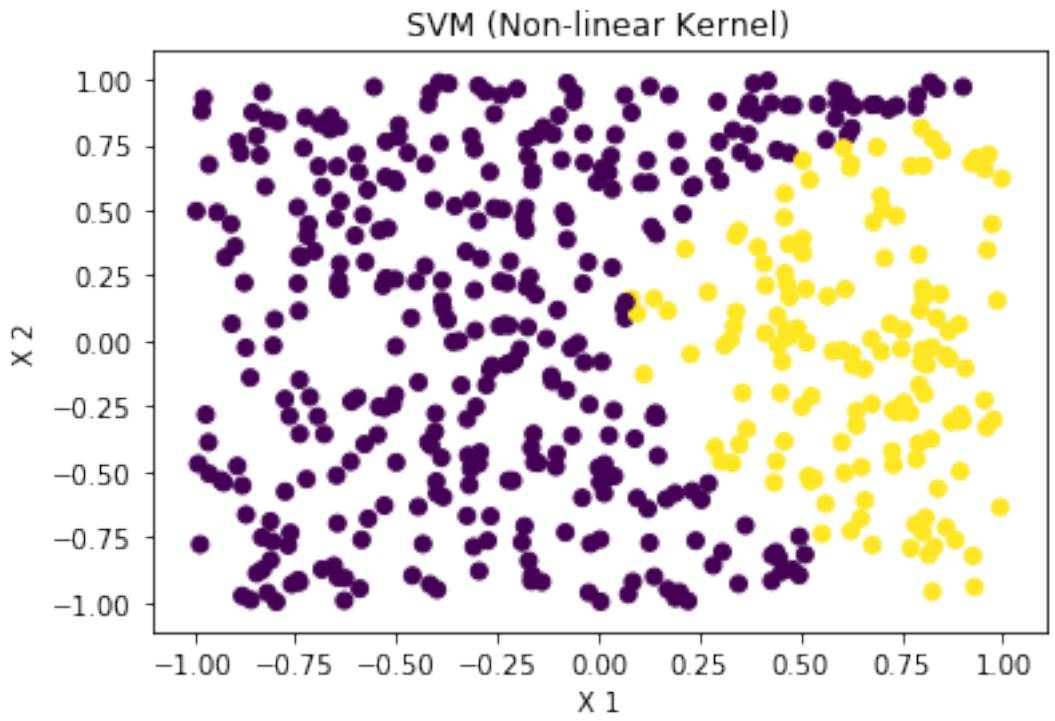


2.8 Task 9

[22]:
```
clf_svm2 = svm.SVC(gamma=1, probability=True, random_state=SEED)
clf_svm2.fit(X, y)
```

[22]:
```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma=1, kernel='rbf', max_iter=-1,
    probability=True, random_state=970608, shrinking=True, tol=0.001,
    verbose=False)
```

[23]:
```
y_svm2_pred = clf_svm2.predict(X)

plt.scatter(x1, x2, c=y_svm2_pred)
plt.xlabel('X 1')
plt.ylabel('X 2')
plt.title('SVM (Non-linear Kernel)')
plt.show()
```



2.9 Task 10

Generally speaking, for both logistic regression and SVC/SVM, non-linear models perform better than linear model because the underlying data itself is non-linear. As for the tradeoffs when estimating non-linear boundary, on the one hand, the benefit of using logistic regression consists in the computational cost, that is, logistic models are more computationally efficient than SVM. However, on the other hand, when implementing polynomial logistic models, we need to define a polynomial equation based on which to perform the regression. And how we define this polynomial equation may influence the performance of the logistic model. But this would not be a big issue for SVM. Therefore, if the error rates of non-linear logistic and non-linear SVM are not significantly different, using logistic regression would be more reasonable for the sake of computational efficiency; otherwise, SVM may be a better choice.

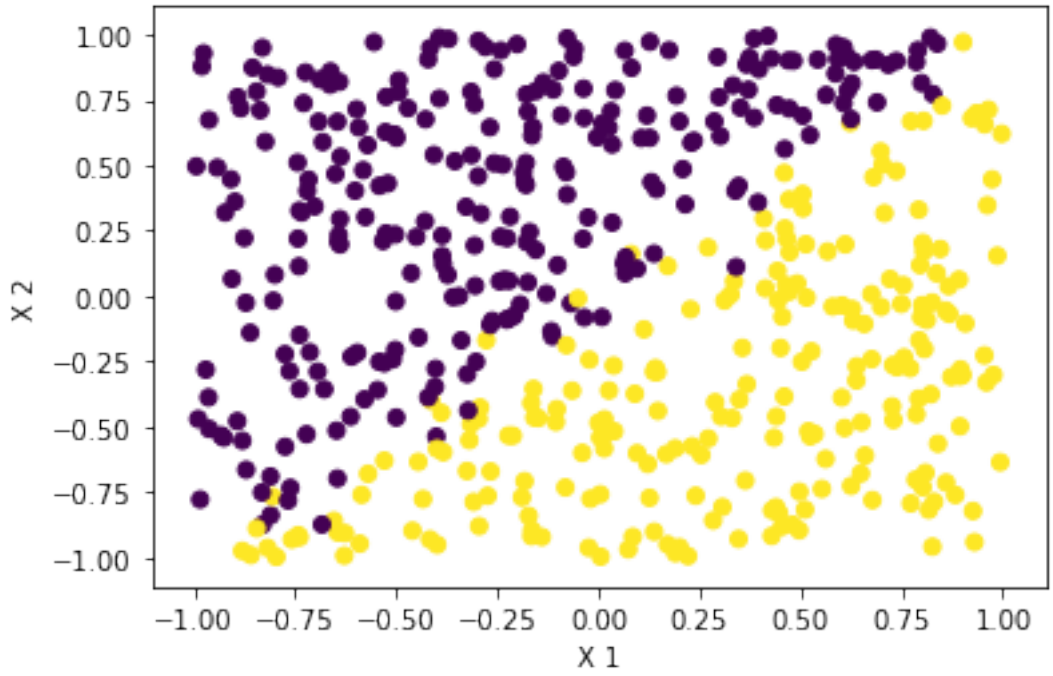
3 Tuning cost

3.1 Task 11

[24]:
```
np.random.seed(SEED)
x1 = np.random.uniform(-1, 1, 500)
x2 = np.random.uniform(-1, 1, 500)
y = x1 - x2 > np.random.normal(0, 0.1, 500)

x1_train, x1_test = x1[:400], x1[400:]
```

```
x2_train, x2_test = x2[:400], x2[400:]

X_train, X_test = np.stack((x1_train, x2_train), axis=-1), np.stack((x1_test,␣
 ↪x2_test), axis=-1)
y_train, y_test = y[:400], y[400:]
```

[25]:
```
plt.scatter(x1, x2, c=y)
plt.xlabel('X 1')
plt.ylabel('X 2')
plt.show()
```



3.2 Task 12, 13

Task 12:
As we can see from both the table and the plot below, the training error and cv error are highly correlated with each other. Both of them strikingly decrease when cost increases from 0.001 to 0.01. After then, both metrics only has limited change and lands on a quite stable stage when cost reaches 0.1. Furthermore, training error gets to its minimum (0.04) when cost equals to 0.5, while cv error gets to its minimum (0.037) when cost is 1. But the difference between their minimum value is very limited.

Task 13:
The test error gets to its minimum when cost is 1, which is the same as the cost which yields the minimal cv error is higher than the one yielding the minimal training error.

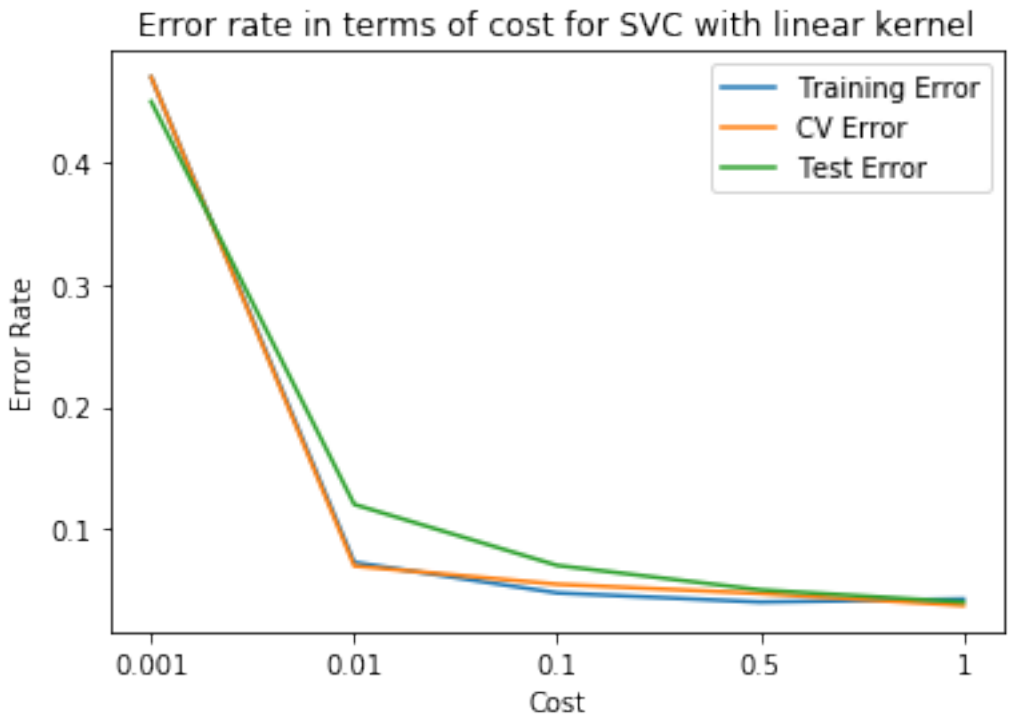
[26]:
```
costs = [0.001, 0.01, 0.1, 0.5, 1]
train_errors, cv_errors, test_errors = [], [], []
for c in costs:
    clf_svc = svm.SVC(kernel='linear', C=c, random_state=SEED)
    train_cv_err = 1 - cross_val_score(clf_svc, X_train, y_train, cv=10).mean()
    clf_svc.fit(X_train, y_train)
    train_err = 1 - clf_svc.score(X_train, y_train)
    test_err = 1 - clf_svc.score(X_test, y_test)
    train_errors.append(train_err)
    cv_errors.append(train_cv_err)
    test_errors.append(test_err)

pd.DataFrame({'cost': costs, 'training error': train_errors, 'cv error':␣
 ↪cv_errors, 'test error': test_errors})
```

[26]:

	cost	training error	cv error	test error
0	0.001	0.4700	0.469991	0.45
1	0.010	0.0725	0.069586	0.12
2	0.100	0.0475	0.054583	0.07
3	0.500	0.0400	0.047018	0.05
4	1.000	0.0425	0.037201	0.04

[27]:
```
plt.plot(train_errors)
plt.plot(cv_errors)
plt.plot(test_errors)
plt.xlabel('Cost')
plt.ylabel('Error Rate')
plt.xticks(list(range(5)), ('0.001', '0.01', '0.1', '0.5', '1'))
plt.legend(['Training Error', 'CV Error', 'Test Error'])
plt.title('Error rate in terms of cost for SVC with linear kernel')
```

[27]:
```
Text(0.5, 1.0, 'Error rate in terms of cost for SVC with linear kernel')
```



3.3 Task 14

Since the data set we generate is only barely linear, we cannot perfectly separate two classes using a linear hyperplane. As a result, when the cost is very low (which is 0.001 in our case), we would get a high error rate. However, this error rate could be reduced to a great extent if we introduce the tolerance of error for our linear SVC. This is also because our data set is barely linear, which means that many of the data points could still be correctly separated by a linear hyperplane. Furthermore, from the results above, we can see that the accuracy of our model, in terms of both training set and test set, could land on a relatively good level when we set the cost as 0.1. A cost higher than 0.1 would not bring significant improvement to the model.

4 Application: Predicting attitudes towards racist college professors

4.1 Task 15

As illustrated by the table below, when cost equals to 0.5, cv error lands on its minimal value (0.203); when cost is 0.01, test error gets to its minimum (0.197). But the difference between these two value is very limited.

[28]:
```
df_train = pd.read_csv('data/gss_train.csv')
df_test = pd.read_csv('data/gss_test.csv')
```

```
X_train, X_test = df_train.drop('colrac', axis=1), df_test.drop('colrac',␣
↪axis=1)
y_train, y_test = df_train['colrac'], df_test['colrac']
```

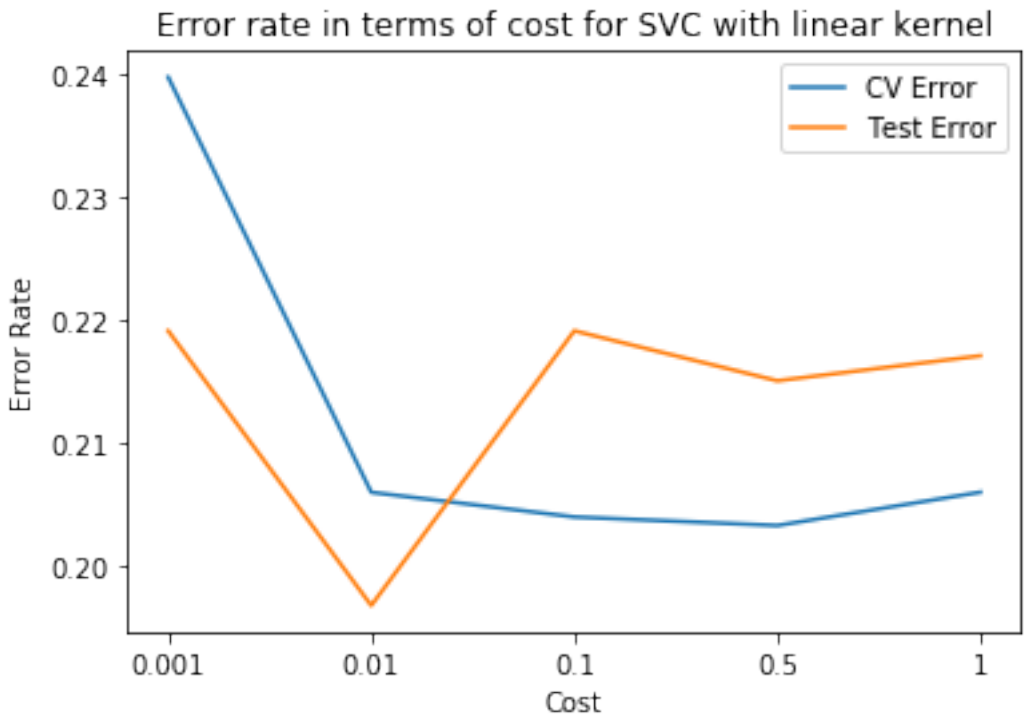
[29]:
```
# For SVC with linear kernel.
costs = [0.001, 0.01, 0.1, 0.5, 1]
cv_errors, test_errors = [], []
for c in costs:
    clf_svc = svm.SVC(kernel='linear', C=c, random_state=SEED)
    cv_err = 1 - cross_val_score(clf_svc, X_train, y_train, cv=10).mean()
    clf_svc.fit(X_train, y_train)
    test_err = 1 - clf_svc.score(X_test, y_test)
    cv_errors.append(cv_err)
    test_errors.append(test_err)
pd.DataFrame({'cost': costs, 'cv error': cv_errors, 'test error': test_errors})
```

[29]:

	cost	cv error	test error
0	0.001	0.239674	0.219067
1	0.010	0.205949	0.196755
2	0.100	0.203945	0.219067
3	0.500	0.203251	0.215010
4	1.000	0.205958	0.217039

[30]:
```
plt.plot(cv_errors)
plt.plot(test_errors)
plt.xlabel('Cost')
plt.ylabel('Error Rate')
plt.xticks(list(range(5)), ('0.001', '0.01', '0.1', '0.5', '1'))
plt.legend(['CV Error', 'Test Error'])
plt.title('Error rate in terms of cost for SVC with linear kernel')
```

[30]:
```
Text(0.5, 1.0, 'Error rate in terms of cost for SVC with linear kernel')
```



4.2 Task 16

According to the cv error and testing error of reported below, the optimal cost for both models is 1. Furthermore, we can easily see that, the SVM with polynomial kernel has a better performance since it has a lower error rate than the SVM with radial kernel.

However, if we compare the performance of the SVM with polynomial kernel with SVC, we can find that it does not achieve a significantly better performance than the linear SVC. Therefore, since the linear SVC is much more efficient in terms of computation cost, it would be more reasonable to use the linear SVC instead of the polynomial SVM for this data set.

[31]:
```
# For SVM with radial kernel.
grid_svm = {'C': [0.001, 0.01, 0.1, 0.5, 1],
            'degree': [1, 2, 3],
            'gamma': ['scale', 'auto']}
svm_radial = svm.SVC(kernel='rbf', random_state=SEED)
svm_radial_grid = GridSearchCV(estimator=svm_radial,
                               param_grid=grid_svm,
                               cv=10)
svm_radial_grid.fit(X_train, y_train)
svm_radial_grid.best_params_
```

[31]:
```
{'C': 1, 'degree': 1, 'gamma': 'scale'}
```

[32]:
```
svm_radial = svm.SVC(kernel='rbf', C=1, degree=1, gamma='scale',␣
↪random_state=SEED)
cv_err_radial = 1 - cross_val_score(svm_radial, X_train, y_train, cv=10).mean()
svm_radial.fit(X_train, y_train)
test_err_radial = 1 - svm_radial.score(X_test, y_test)
print(f'CV error of radial kernel: {cv_err_radial}')
print(f'Testing error of radial kernel: {test_err_radial}')
```

```
CV error of radial kernel: 0.24916570110721592
Testing error of radial kernel: 0.231237322515213
```

[33]:
```
# For SVM with poly kernel.
grid_svm = {'C': [0.001, 0.01, 0.1, 0.5, 1],
            'degree': [1, 2, 3],
            'gamma': ['scale', 'auto']}
svm_poly = svm.SVC(kernel='poly', random_state=SEED)
svm_poly_grid = GridSearchCV(estimator=svm_poly,
                             param_grid=grid_svm,
                             cv=10)
svm_poly_grid.fit(X_train, y_train)
svm_poly_grid.best_params_
```

[33]:
```
{'C': 1, 'degree': 1, 'gamma': 'auto'}
```

[34]:
```
svm_poly = svm.SVC(kernel='poly', C=1, degree=1, gamma='auto',␣
↪random_state=SEED)
cv_err_poly = 1 - cross_val_score(svm_poly, X_train, y_train, cv=10).mean()
svm_poly.fit(X_train, y_train)
test_err_poly = 1 - svm_poly.score(X_test, y_test)
print(f'CV error of poly kernel: {cv_err_poly}')
print(f'Testing error of poly kernel: {test_err_poly}')
```

```
CV error of poly kernel: 0.20189033712523652
Testing error of poly kernel: 0.2068965517241379
```