Problem Set 6

Chia-Yun Chang

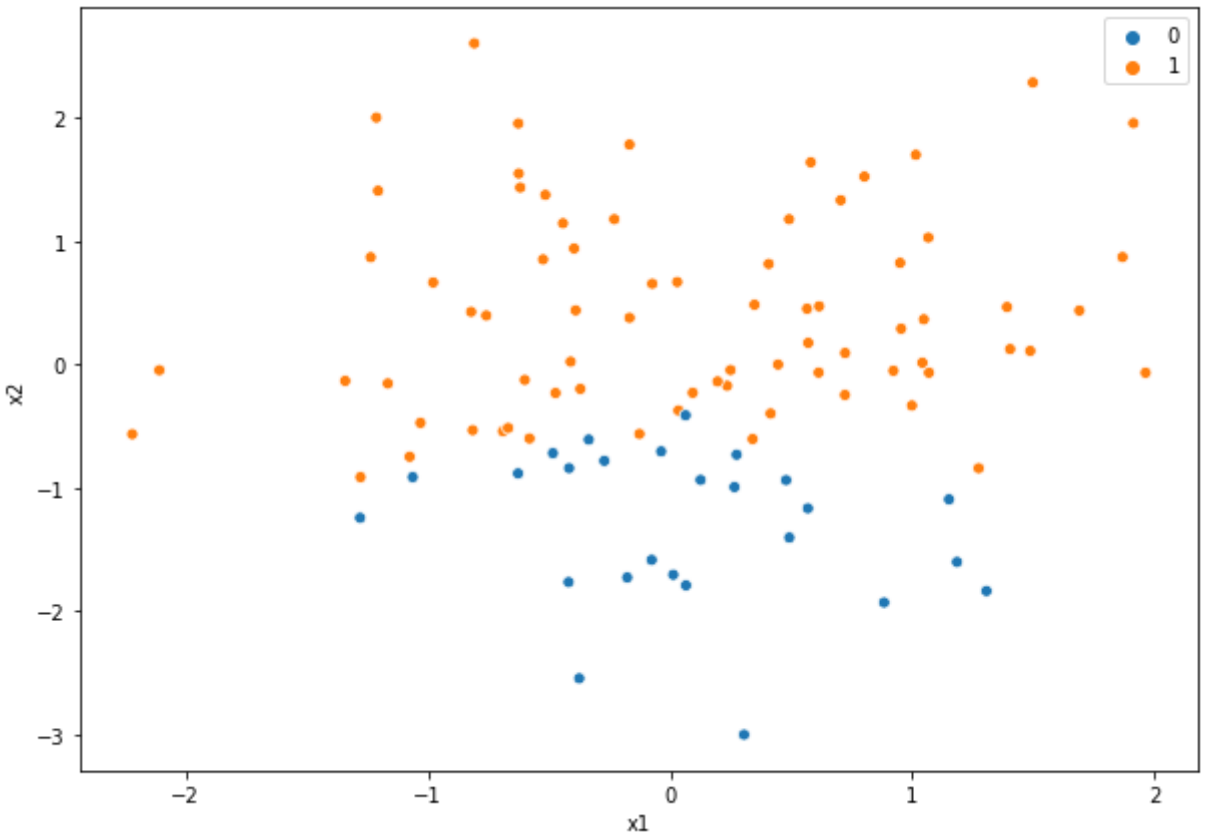
Non-linear seperation

In [264]:
```
import random
import numpy as np
import pandas as pd
import sklearn.model_selection
from sklearn.model_selection import train_test_split
from tabulate import tabulate
import math
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.svm import SVC, SVR
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import cross_val_score
from sklearn.metrics import accuracy_score
```

In [174]:
```
#Generate data
random.seed(5566)
x1 = np.random.randn(100)
x2 = np.random.randn(100)
y = pd.DataFrame(x1**2 + x2**3 + x2 + np.random.uniform(0,1,100)>0)
y = y.apply(lambda x: np.int64(x)[0], axis = 1)
x = pd.DataFrame({'x1':x1, 'x2':x2})
```

In [175]:
```
# visualization of simulated data
plt.figure(figsize=(10,7))
sns.scatterplot(x = 'x1', y = 'x2', data = x, hue = y)
```

Out[175]: <matplotlib.axes._subplots.AxesSubplot at 0x123d79e90>



In [176]:
```
x_tr, x_te, y_tr, y_te = train_test_split(x, y, train_size=0.8)
```

In [177]:
```
svc_rad = SVC(kernel='rbf')
svc_rad.fit(x_tr, y_tr)
svc_lin = SVC(kernel='linear')
svc_lin.fit(x_tr, y_tr)
```

Out[177]:
```
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='linea
r',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

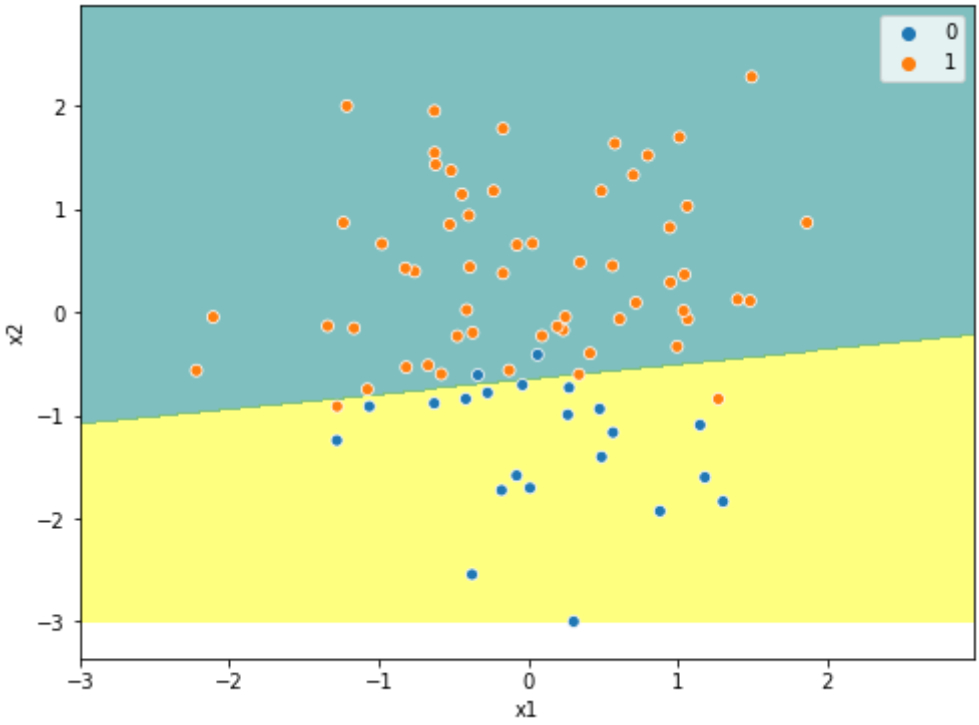
In [178]:
```
# Check test and train error
print('Radial Kernel train accuracy:', svc_rad.score(x_tr, y_tr))
print('Radial Kernel test accuracy: ', svc_rad.score(x_te, y_te))
print('Linear Kernel train accuracy:',sc_lin.score(x_tr, y_tr))
print('linear Kernel test accuracy: ',sc_lin.score(x_te, y_te))
```

```
Radial Kernel train accuracy: 0.9625
Radial Kernel test accuracy:  1.0
Linear Kernel train accuracy: 0.925
linear Kernel test accuracy:  0.95
```

In [179]:
```
ax1, ax2 = np.meshgrid(np.arange(-3, 3, 0.01), np.arange(-3, 3, 0.01))
lin = svc_lin.predict(np.c_[ax1.ravel(), ax2.ravel()]).reshape(ax1.shape)
rad = svc_rad.predict(np.c_[ax1.ravel(), ax2.ravel()]).reshape(ax1.shape)
```

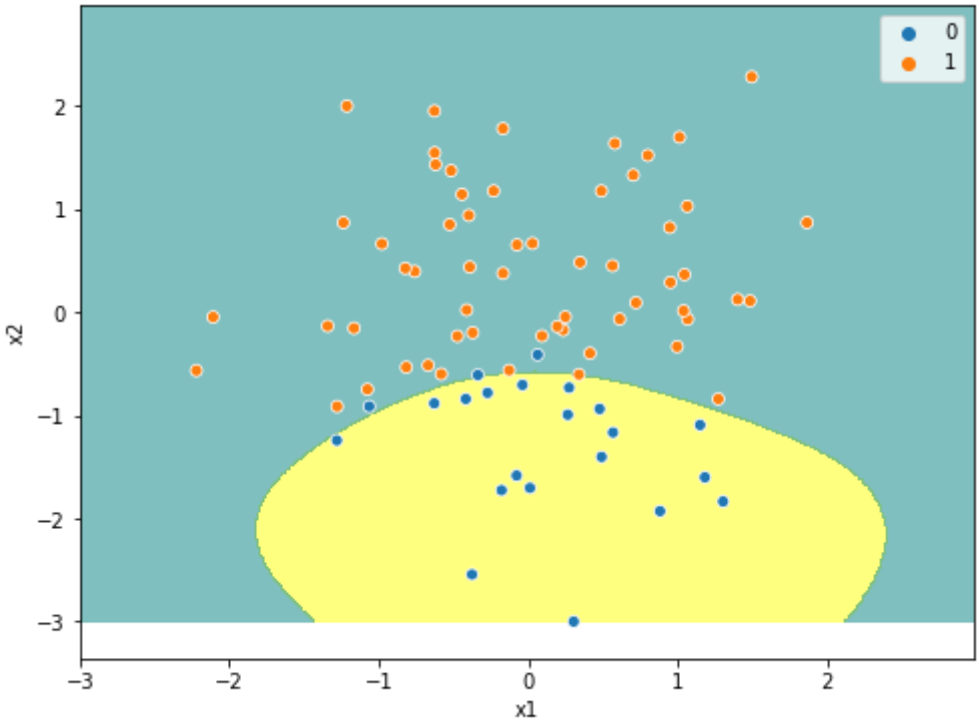
In [180]:
```
plt.figure(figsize=(8,6))
plt.contourf(ax1, ax2, lin, colors=['yellow','teal' ], levels=1, alpha=.5)
sns.scatterplot(x='x1', y='x2', data=x_tr, hue=y_tr)
# SVC with a linear kernel
```

Out[180]: <matplotlib.axes._subplots.AxesSubplot at 0x11f5c4d50>



In [181]:
```
plt.figure(figsize=(8,6))
plt.contourf(ax1, ax2, rad, colors=['yellow', 'teal'], levels=1, alpha=.5)
sns.scatterplot('x1', 'x2', data=x_tr, hue=y_tr)
# SVC with a radial kernel
```

Out[181]: <matplotlib.axes._subplots.AxesSubplot at 0x123455b50>



This simulation confirms that SVC with a radial kernel outperforms SVC with a linear kernel. The radial kernel test accuracy is 1.0, while the linear Kernel test accuracy is 0.95. The true relation between y and x1, x2 is y = x1^2 + x2^3 + x2, with a simulated error term to provide a less than perfect split. This relation is clearly non-linear, which means that raising the dimension with a linear kernel will not give an ideal hyperplane that splits the two classes.

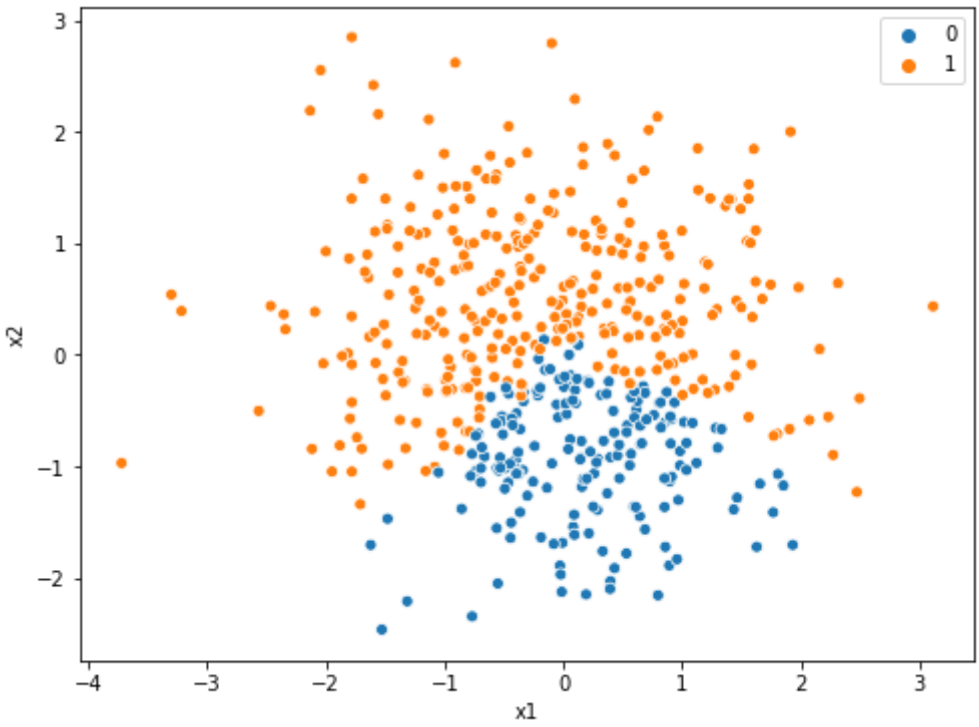
SVM and Logistic Regression

In [214]:
```
random.seed(5566)
x1 = np.random.randn(500)
x2 = np.random.randn(500)
y = pd.DataFrame(x1**2 + x2**3 + x2 + x1*x2 + np.random.uniform(-0.22,0.22,
y = y.apply(lambda x: np.int64(x)[0], axis = 1)
x = pd.DataFrame({'x1':x1, 'x2':x2})
```

Logistic Regression with and without polinomial term

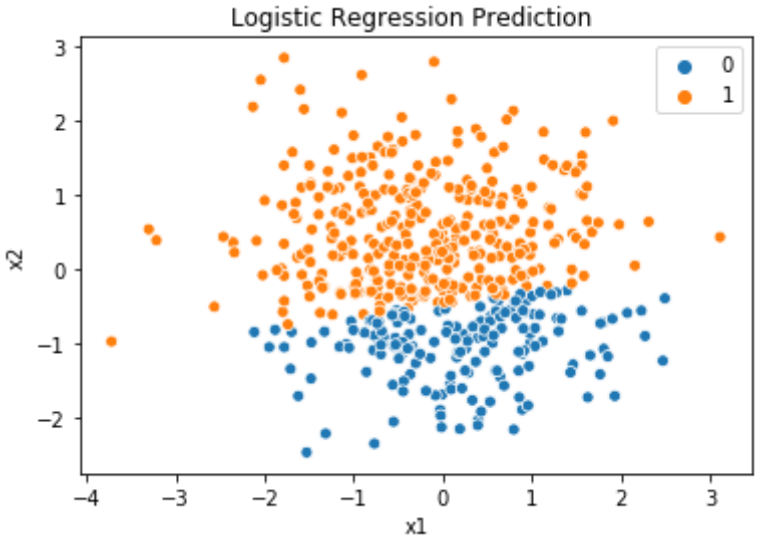
In [215]:
```
# Check visualization
plt.figure(figsize=(8,6))
sns.scatterplot( x='x1', y='x2', hue=y, data=x)
```

Out[215]: <matplotlib.axes._subplots.AxesSubplot at 0x131bc4bd0>



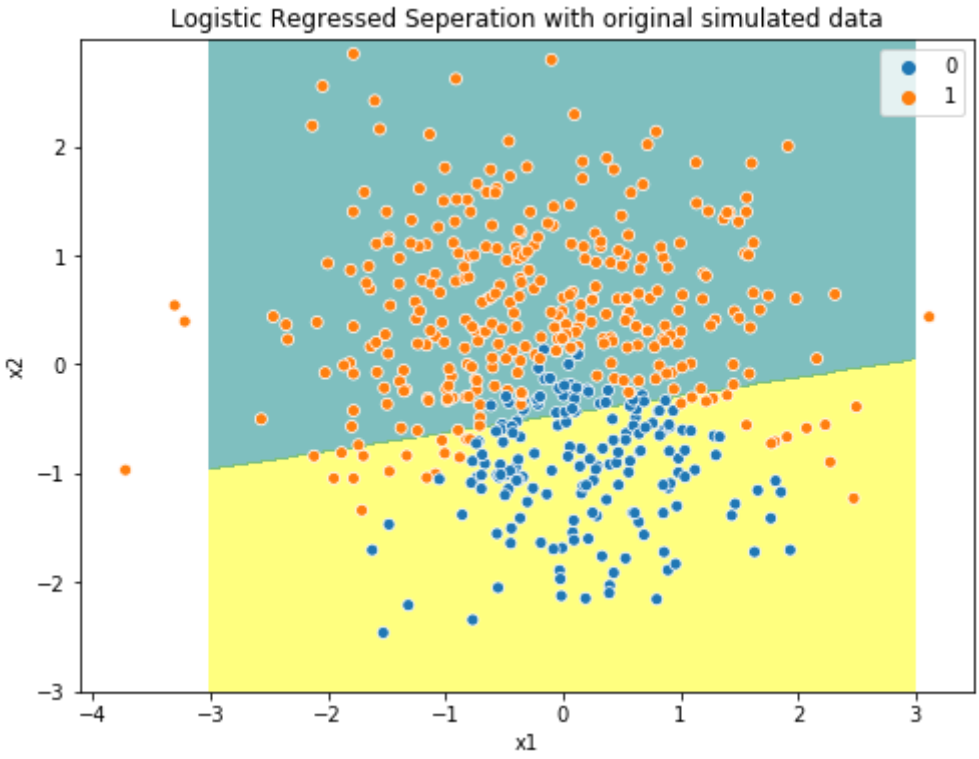
In [216]:
```
logr = LogisticRegression()
logr.fit(x, y)
sns.scatterplot(x='x1', y='x2', hue=logr.predict(x), data=x)
plt.title('Logistic Regression Prediction ')
```

Out[216]: Text(0.5, 1.0, 'Logistic Regression Prediction ')



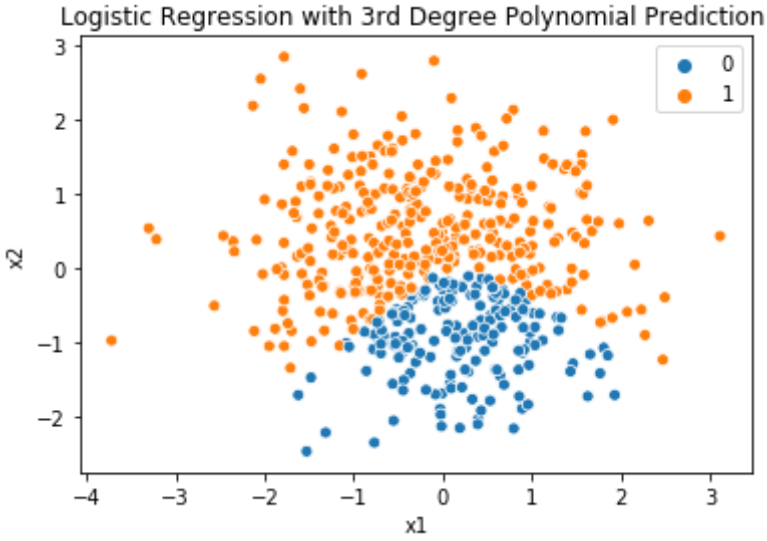
In [217]:
```
# visualize the hyperplane and original data
ax1, ax2 = np.meshgrid(np.arange(-3, 3, 0.01), np.arange(-3, 3, 0.01))
loglin = logr.predict(np.c_[ax1.ravel(), ax2.ravel()]).reshape(ax1.shape)
plt.figure(figsize=(8,6))
plt.contourf(ax1, ax2, loglin, colors=['yellow','teal' ], levels=1, alpha=.
sns.scatterplot(x='x1', y='x2', data=x, hue=y)
plt.title('Logistic Regressed Seperation with original simulated data')
```

Out[217]:
```
Text(0.5, 1.0, 'Logistic Regressed Seperation with original simulated dat
a')
```



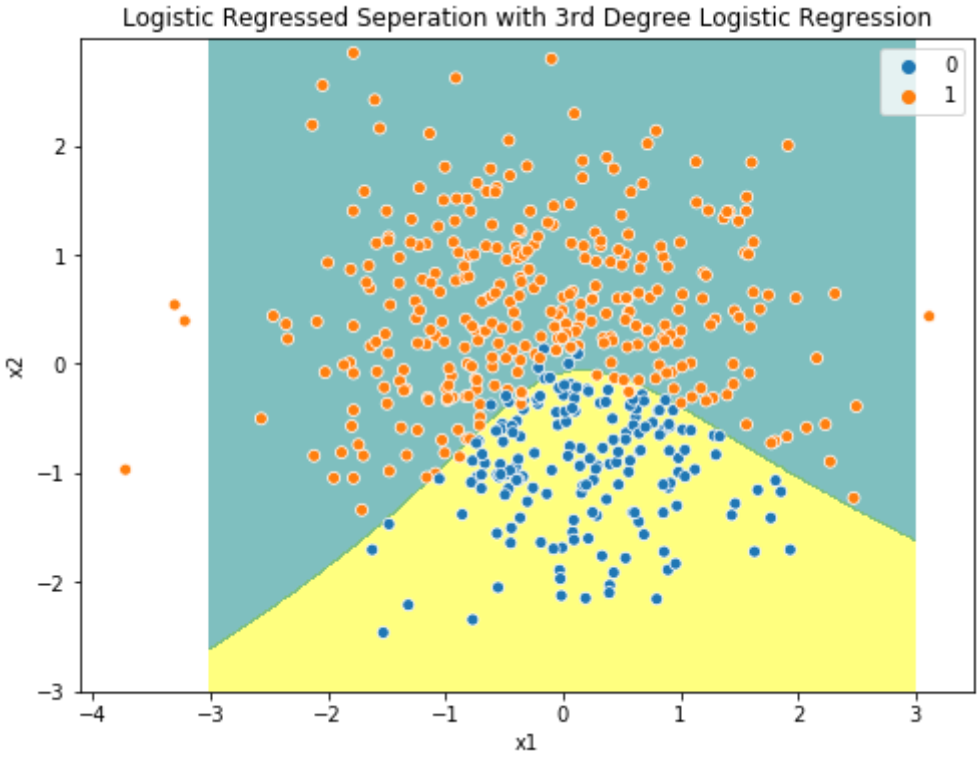
In [218]:
```
logr_poly = Pipeline([('poly', PolynomialFeatures(degree=3)),('logit', Logi
logr_poly.fit(x, y)
sns.scatterplot(x='x1', y='x2', hue=logr_poly.predict(x), data=x)
plt.title('Logistic Regression with 3rd Degree Polynomial Prediction ')
```

Out[218]:
```
Text(0.5, 1.0, 'Logistic Regression with 3rd Degree Polynomial Prediction
')
```



In [219]:
```
ax1, ax2 = np.meshgrid(np.arange(-3, 3, 0.01), np.arange(-3, 3, 0.01))
loglin = logr_poly.predict(np.c_[ax1.ravel(), ax2.ravel()]).reshape(ax1.sha
plt.figure(figsize=(8,6))
plt.contourf(ax1, ax2, loglin, colors=['yellow','teal' ], levels=1, alpha=.
sns.scatterplot(x='x1', y='x2', data=x, hue=y)
plt.title('Logistic Regressed Seperation with 3rd Degree Logistic Regressio
```

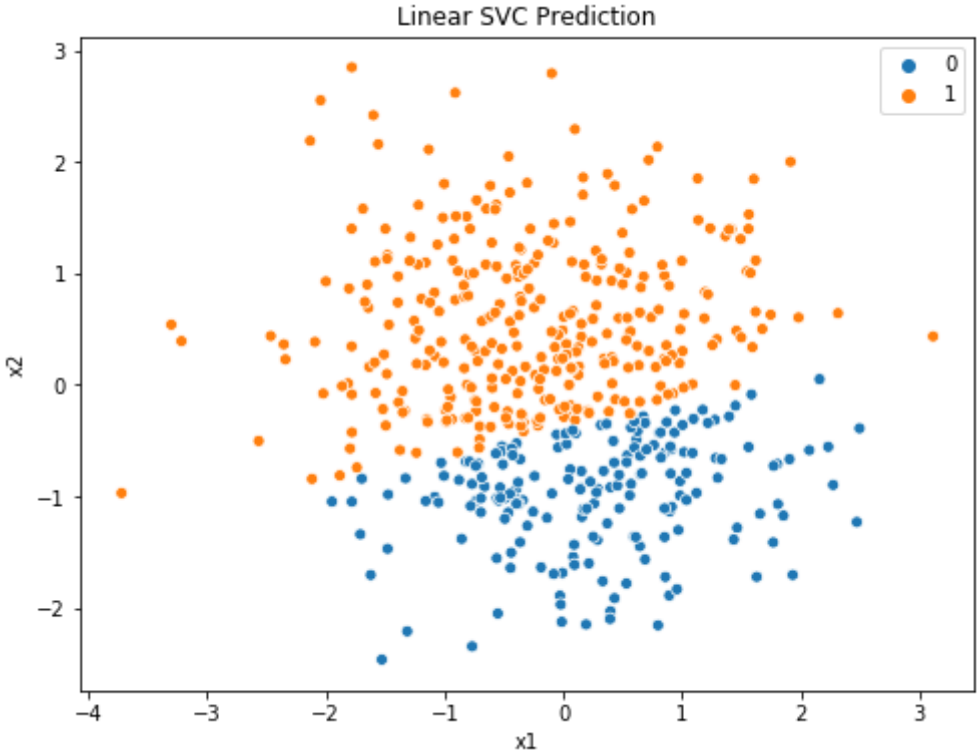
Out[219]:
```
Text(0.5, 1.0, 'Logistic Regressed Seperation with 3rd Degree Logistic Re
gression')
```



SVCs

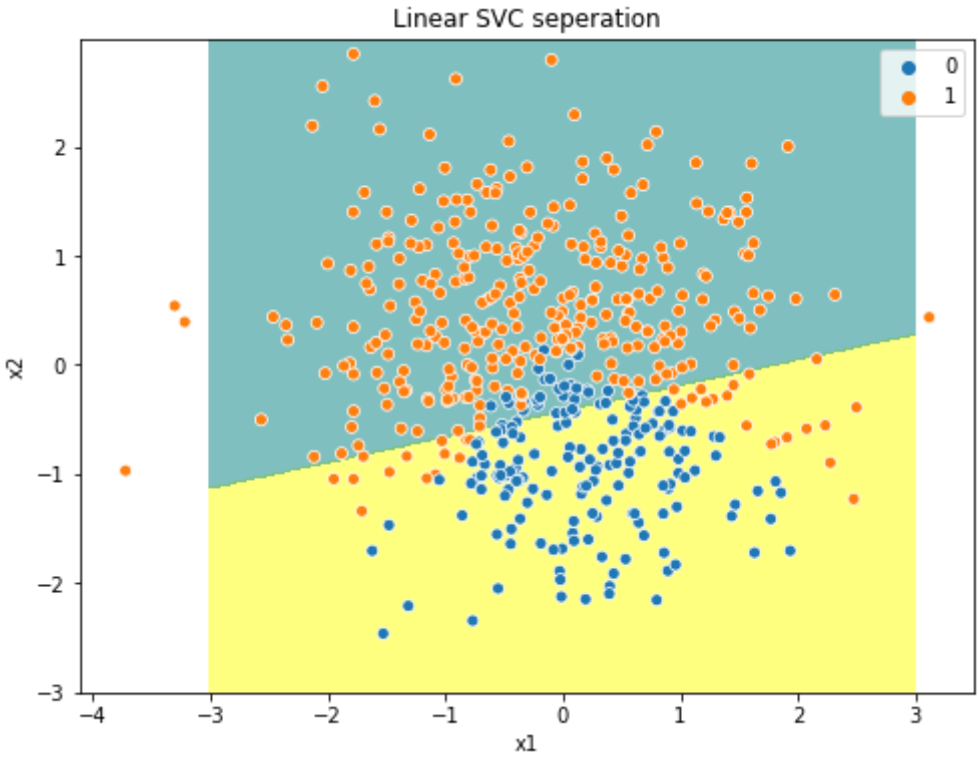
In [220]:
```
svc_lin2 = SVC(kernel='linear')
svc_lin2.fit(x, y)
plt.figure(figsize=(8,6))
sns.scatterplot(x='x1', y='x2', hue = svc_lin2.predict(x), data=x)
plt.title('Linear SVC Prediction')
```

Out[220]: Text(0.5, 1.0, 'Linear SVC Prediction')



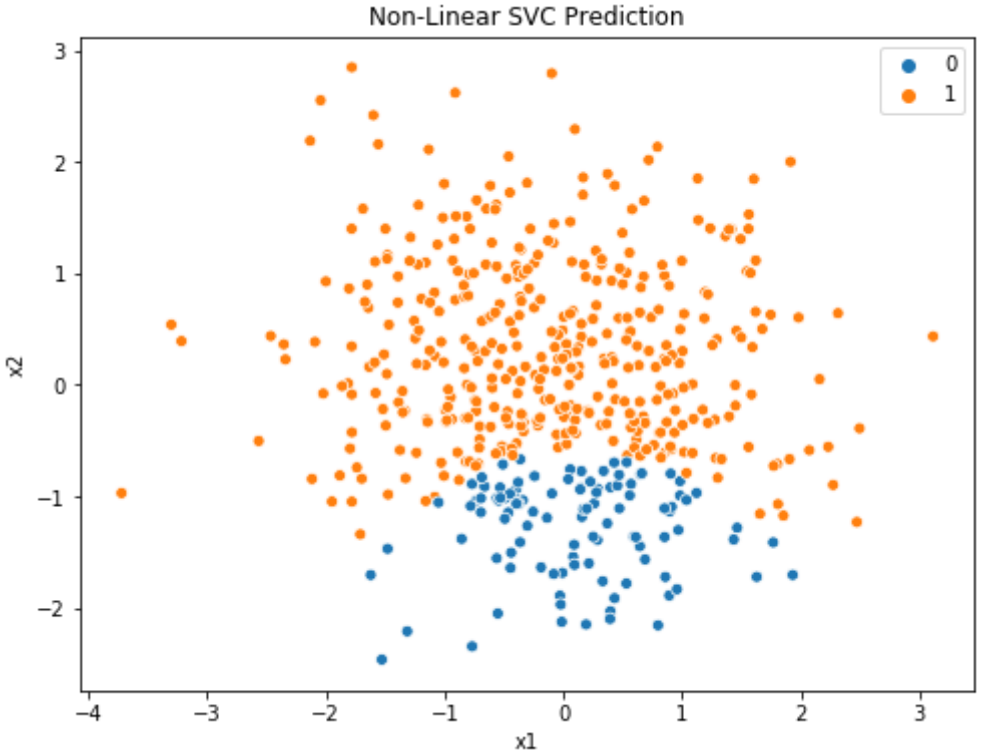
In [221]:
```
ax1, ax2 = np.meshgrid(np.arange(-3, 3, 0.01), np.arange(-3, 3, 0.01))
lin = svc_lin2.predict(np.c_[ax1.ravel(), ax2.ravel()]).reshape(ax1.shape)
plt.figure(figsize=(8,6))
plt.contourf(ax1, ax2, lin, colors=['yellow','teal' ], levels=1, alpha=.5)
sns.scatterplot(x='x1', y='x2', data=x, hue=y)
plt.title('Linear SVC seperation')
```

Out[221]:
```
Text(0.5, 1.0, 'Linear SVC seperation')
```



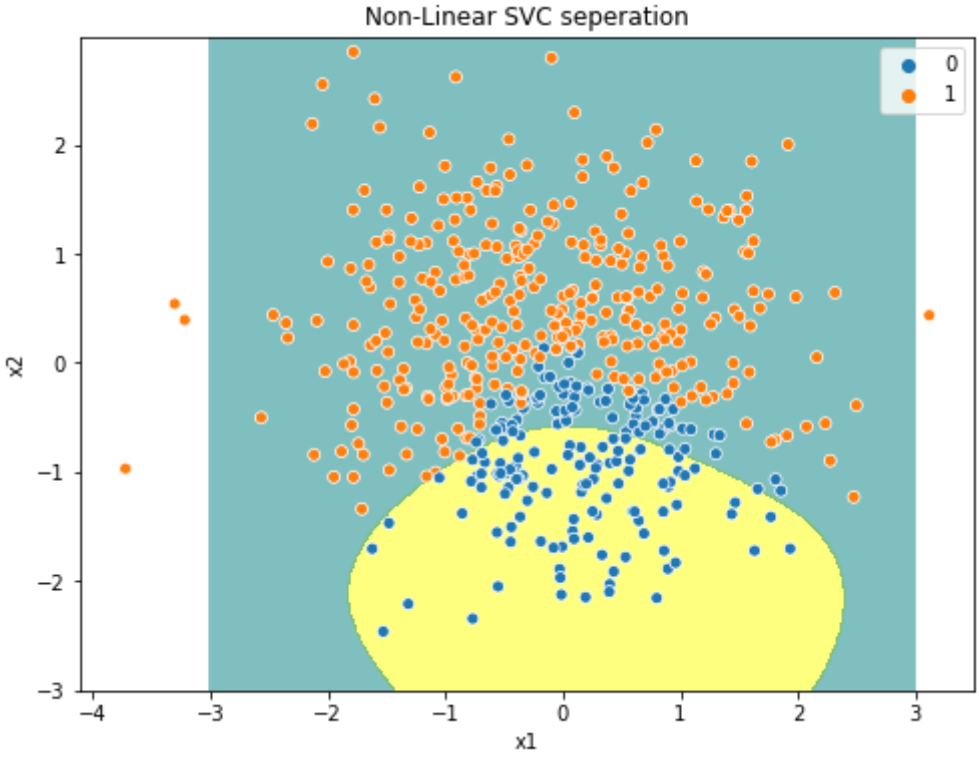
In [229]:
```
svc_rad2 = SVC(kernel='rbf')
svc_rad2.fit(x_tr, y_tr)
plt.figure(figsize=(8,6))
sns.scatterplot(x='x1', y='x2', hue = svc_rad2.predict(x), data=x)
plt.title('Non-Linear SVC Prediction')
```

Out[229]:
```
Text(0.5, 1.0, 'Non-Linear SVC Prediction')
```



In [230]:
```
ax1, ax2 = np.meshgrid(np.arange(-3, 3, 0.01), np.arange(-3, 3, 0.01))
rad2 = svc_rad2.predict(np.c_[ax1.ravel(), ax2.ravel()]).reshape(ax1.shape)
plt.figure(figsize=(8,6))
plt.contourf(ax1, ax2, rad2, colors=['yellow','teal' ], levels=1, alpha=.5)
sns.scatterplot(x='x1', y='x2', data=x, hue=y)
plt.title('Non-Linear SVC seperation')
```

Out[230]:
```
Text(0.5, 1.0, 'Non-Linear SVC seperation')
```



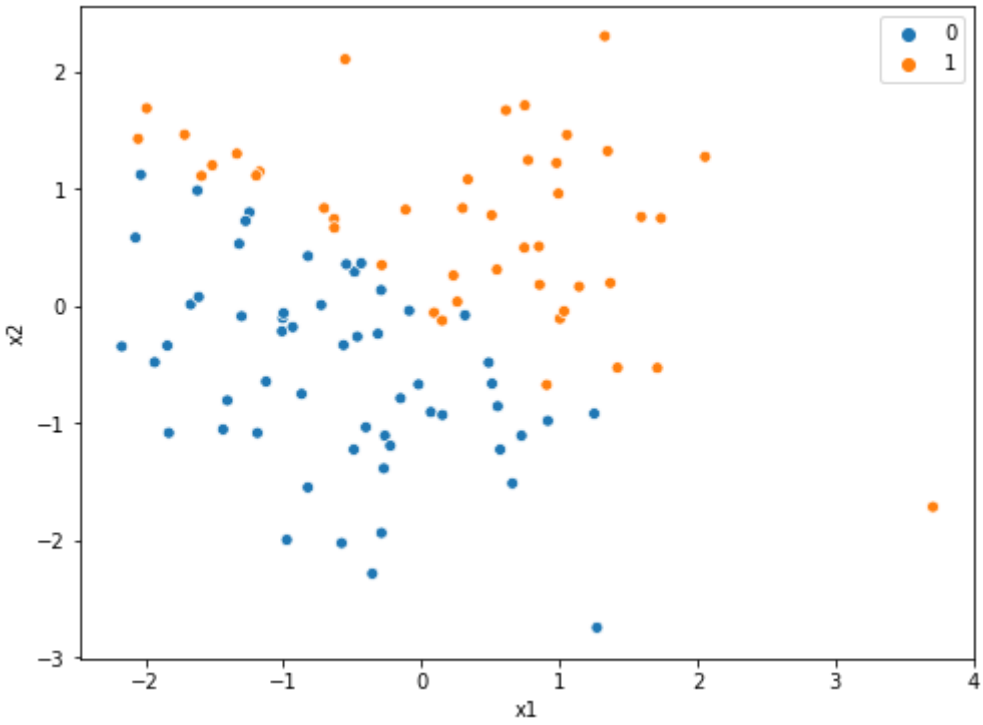
For this simulation data, my 3rd degree polynominal logistic regression outperforms the SVC with radial kernel. We can see from the boundaries created by the 3rd degree polynominal logistic regression and the Radial SVC that the latter provided a worse fit by allowing too much misclassification, too many data points to be outliers. This could be due to the fact that the original boundary function was better captured by the 3rd degree polynominal. The original function was x1^2 + x2^3 + x2 + x1*x2, clearly having a 3rd degree polynomial, may be hard for the radial kernel to capture.

Tuning cost

In [241]:
```
#Generate data

random.seed(5566)
x1 = np.random.randn(100)
x2 = np.random.randn(100)
y = pd.DataFrame(x1*2 + x2*3 + 0.001*x1**3 + np.random.uniform(-0.7,0.7,100
# use a really small x1**3 term and an error term to fake subtle non-linear
y = y.apply(lambda x: np.int64(x)[0], axis = 1)
x = pd.DataFrame({'x1':x1, 'x2':x2})
plt.figure(figsize=(8,6))
sns.scatterplot( x='x1', y='x2', hue=y, data=x)
```

Out[241]: <matplotlib.axes._subplots.AxesSubplot at 0x13961c810>



In [245]:
```
x_tr, x_te, y_tr, y_te = train_test_split(x, y, train_size = 0.8)
```

In [266]:
```
cost = [0.0001,0.1, 1 , 2, 5, 50, 1000 ]
svcs = []
for c in cost:
    svc_lin = SVC(C=c, kernel = 'linear')
    acc = cross_val_score(svc_lin, x_tr, y_tr, scoring = 'accuracy', cv = 1
    print('Cost = {}'.format(c), 'CV mean accuracy:', sum(acc)/10)
    svc_lin.fit(x_tr,y_tr)
    svcs.append((c, svc_lin))
```

```
Cost = 0.0001 CV mean accuracy: 0.55
Cost = 0.1 CV mean accuracy: 0.9
Cost = 1 CV mean accuracy: 0.925
Cost = 2 CV mean accuracy: 0.925
Cost = 5 CV mean accuracy: 0.9625
Cost = 50 CV mean accuracy: 0.9375
Cost = 1000 CV mean accuracy: 0.925
```

In [268]:
```
for m in svcs:
    y = m[1].predict(x_te)
    acc = sklearn.metrics.accuracy_score(y_te, y)
    print('Cost = {}'.format(m[0]), 'test accuracy', acc)
```

```
Cost = 0.0001 test accuracy 0.6
Cost = 0.1 test accuracy 0.95
Cost = 1 test accuracy 0.95
Cost = 2 test accuracy 0.95
Cost = 5 test accuracy 0.95
Cost = 50 test accuracy 0.95
Cost = 1000 test accuracy 0.95
```

As shown above, as the cost increases, the accucary becomes better. With the cost at as low as 0.0001, the accuracy is pretty poor, at 0.55, just slightly better than making random guess. We can also see that at huge cost as 50 and 1000, the accuracy decreased since they are both too strict of a classifier that results in overfit. The perfect relaxation in the arbitrary cost set, cost = 5 performs the best.

In the testing stage, we see that cost = 0.0001 relaxes too much, and misclasifies alot, rendering the lowest accuracy rate. However, for costs = 0.1, 1 , 2, 5, 50, 1000, all of them perform well with 0.95 accuracy rate, although some of them performed poorly in the cross-validation. This could be due to that fact that the original generating function is highly linear, just very slightly non-linear, so there wouldn't be much chance to mis-classify even if we assign very strict margin hyperplanes. In other words, because the true function is so linear, it will not perform badly even its over-fitting. The discrepancies between the testing and training CV accuracies may be due to that CV accuracy are averaging less samples than the testing dataset.

Application

In [273]:
```
tr = pd.read_csv('gss_train.csv')
te = pd.read_csv('gss_test.csv')
x_tr = tr.drop('colrac', axis=1)
y_tr = tr.colrac
x_te = te.drop('colrac', axis=1)
y_te = te.colrac
```

In [289]:
```
cost = np.linspace(0.0001,1,10)
svcs = []
for c in cost:
    svc_lin = SVC(C=c, kernel = 'linear')
    acc = cross_val_score(svc_lin, x_tr, y_tr, scoring = 'accuracy', cv = 1
    print('Cost = {}'.format(c), 'CV mean accuracy:', sum(acc)/10)
    svc_lin.fit(x_tr,y_tr)
    svcs.append((c, svc_lin))
```

```
Cost = 0.0001 CV mean accuracy: 0.6968619626337746
Cost = 0.11120000000000001 CV mean accuracy: 0.7954108470886995
Cost = 0.2223 CV mean accuracy: 0.7960910575004534
Cost = 0.33340000000000003 CV mean accuracy: 0.794739706149102
Cost = 0.4445 CV mean accuracy: 0.7940640304734264
Cost = 0.5556 CV mean accuracy: 0.7947442408851805
Cost = 0.6667000000000001 CV mean accuracy: 0.7940685652095049
Cost = 0.7778 CV mean accuracy: 0.7933928895338291
Cost = 0.8889 CV mean accuracy: 0.7933928895338291
Cost = 1.0 CV mean accuracy: 0.7940685652095048
```

In [304]:
```
cost = [1,5,10]
for c in cost:
    svc_lin = SVC(C=c, kernel = 'linear')
    acc = cross_val_score(svc_lin, x_tr, y_tr, scoring = 'accuracy', cv = 1
    print('Cost = {}'.format(c), 'CV mean accuracy:', sum(acc)/10)
```

```
Cost = 1 CV mean accuracy: 0.7940685652095048
Cost = 5 CV mean accuracy: 0.7913658625068021
Cost = 10 CV mean accuracy: 0.7906901868311265
```

We get peak accuracy at cost = 1. Costs smaller are too relaxed, while costs larger than 1 are too strict.

In [307]:
```
for k in ['rbf','poly']:
    for c in [0.5, 1, 1.5]:
        for g in ['scale', 'auto']:
            if k == 'rbf':
                svc = SVC(kernel=k, C=c, gamma = g)
                acc = cross_val_score(svc, x_tr, y_tr, scoring = 'accuracy'
                print(k,c,g, 'CV mean accuracy:', sum(acc)/10)
            if k == 'poly':
                for d in [2,3,4]:
                    svc = SVC(kernel=k, C=c, gamma = g)
                    acc = cross_val_score(svc, x_tr, y_tr, scoring = 'accur
                    print(k,c,g,d, 'CV mean accuracy:', sum(acc)/10)
```

```
rbf 0.5 scale CV mean accuracy: 0.7386994376927264
rbf 0.5 auto CV mean accuracy: 0.7157536731362235
rbf 1 scale CV mean accuracy: 0.7494830400870669
rbf 1 auto CV mean accuracy: 0.7278976963540722
rbf 1.5 scale CV mean accuracy: 0.7569018683112643
rbf 1.5 auto CV mean accuracy: 0.7359831307817886
poly 0.5 scale 2 CV mean accuracy: 0.7380192272809724
poly 0.5 scale 3 CV mean accuracy: 0.7380192272809724
poly 0.5 scale 4 CV mean accuracy: 0.7380192272809724
poly 0.5 auto 2 CV mean accuracy: 0.7386676945401778
poly 0.5 auto 3 CV mean accuracy: 0.7386676945401778
poly 0.5 auto 4 CV mean accuracy: 0.7386676945401778
poly 1 scale 2 CV mean accuracy: 0.746122800653002
poly 1 scale 3 CV mean accuracy: 0.746122800653002
poly 1 scale 4 CV mean accuracy: 0.746122800653002
poly 1 auto 2 CV mean accuracy: 0.7386676945401778
poly 1 auto 3 CV mean accuracy: 0.7386676945401778
poly 1 auto 4 CV mean accuracy: 0.7386676945401778
poly 1.5 scale 2 CV mean accuracy: 0.7562443315799021
poly 1.5 scale 3 CV mean accuracy: 0.7562443315799021
poly 1.5 scale 4 CV mean accuracy: 0.7562443315799021
poly 1.5 auto 2 CV mean accuracy: 0.7386676945401778
poly 1.5 auto 3 CV mean accuracy: 0.7386676945401778
poly 1.5 auto 4 CV mean accuracy: 0.7386676945401778
```

For the gamma parameter, with higher gamma, the data points near the hyperplane carry more weight, dragging the plane more, more prone to overfit. The auto mode is 1 / n_features, while scale uses 1 / (n_features * X.var()). In my search for the hyper parameters, scale always render higher accuracy than auto with radial kernel, and most of the time with polynomial kernel. For cost =1, kernel = polynomial, auto performs better. For cost, cost = 1.5 outperforms cost = 1, I think we need a finer gridsearch to actually find the most ideal cost. For polynomial SVCs, the degree does not have observable impact on how the model performs. They remains the same within different hyper parameters. Based solely on the above finding, the most ideal SVC would be radial at cost=1.5 with gamma = scale. Overall the setting of kernel and cost has the most impact.