MLHomework6

December 14, 2021

0.1 Part 1

(a) What is the fundamental idea behind Support Vector Machines?

The idea behind Support Vector Machines is to fit the widest possible street (represented by parallel dashed lines) in between the 2 classes. Thus, the algorithm for SVM creates either a line or a hyperplane that ends up separating data into distinct classes.

[34]:
```
import numpy as np
import pandas as pd
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC
from sklearn.model_selection import train_test_split

# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
```

[35]:
```
data = pd.read_csv("Auto.csv")
```

[36]:
```
print(data.shape)
data.head()
```

```
(397, 9)
```

[36]:
```
    mpg  cylinders  displacement horsepower  weight  acceleration  year  \
0  18.0          8         307.0        130    3504          12.0    70
1  15.0          8         350.0        165    3693          11.5    70
2  18.0          8         318.0        150    3436          11.0    70
3  16.0          8         304.0        150    3433          12.0    70
4  17.0          8         302.0        140    3449          10.5    70

   origin                       name
0       1  chevrolet chevelle malibu
1       1          buick skylark 320
2       1         plymouth satellite
3       1              amc rebel sst
```

```
4       1                 ford torino
```

[37]:
```
data.describe()
```

[37]:
```
              mpg   cylinders  displacement       weight  acceleration  \
count  397.000000  397.000000    397.000000   397.000000    397.000000
mean    23.515869    5.458438    193.532746  2970.261965     15.555668
std      7.825804    1.701577    104.379583   847.904119      2.749995
min      9.000000    3.000000     68.000000  1613.000000      8.000000
25%     17.500000    4.000000    104.000000  2223.000000     13.800000
50%     23.000000    4.000000    146.000000  2800.000000     15.500000
75%     29.000000    8.000000    262.000000  3609.000000     17.100000
max     46.600000    8.000000    455.000000  5140.000000     24.800000

             year      origin
count  397.000000  397.000000
mean    75.994962    1.574307
std      3.690005    0.802549
min     70.000000    1.000000
25%     73.000000    1.000000
50%     76.000000    1.000000
75%     79.000000    2.000000
max     82.000000    3.000000
```

[38]:
```
data.isnull().any()
```

[38]:
```
mpg             False
cylinders       False
displacement    False
horsepower      False
weight          False
acceleration    False
year            False
origin          False
name            False
dtype: bool
```

[39]:
```
data[data['horsepower'] == '?']
```

[39]:
```
      mpg  cylinders  displacement horsepower  weight  acceleration  year  \
32   25.0          4          98.0          ?    2046          19.0    71
126  21.0          6         200.0          ?    2875          17.0    74
330  40.9          4          85.0          ?    1835          17.3    80
336  23.6          4         140.0          ?    2905          14.3    80
354  34.5          4         100.0          ?    2320          15.8    81

     origin                  name
```

```
32          1             ford pinto
126         1          ford maverick
330         2   renault lecar deluxe
336         1     ford mustang cobra
354         2            renault 18i
```

[40]:
```
data = data[data['horsepower'] != '?']
data['horsepower'] = data['horsepower'].astype(int)
```

[41]:
```
print(data.shape)
data.head()
```

```
(392, 9)
```

[41]:
```
    mpg  cylinders  displacement  horsepower  weight  acceleration  year  \
0  18.0          8         307.0         130    3504          12.0    70
1  15.0          8         350.0         165    3693          11.5    70
2  18.0          8         318.0         150    3436          11.0    70
3  16.0          8         304.0         150    3433          12.0    70
4  17.0          8         302.0         140    3449          10.5    70

   origin                       name
0       1  chevrolet chevelle malibu
1       1          buick skylark 320
2       1         plymouth satellite
3       1              amc rebel sst
4       1                ford torino
```

(b) Create a binary variable that takes on a 1 for cars with gas mileage above the median, and a 0 for cars with gas mileage below the median.

[42]:
```
mpg_median = data["mpg"].median()
mpg_median
```

[42]:
```
22.75
```

[43]:
```
# y = (data["mpg"] > mpg_median).astype(np.float64)
data["mileage_rate"]= (data["mpg"] >= mpg_median).astype(np.intc)
data.tail()
# y
```

[43]:
```
      mpg  cylinders  displacement  horsepower  weight  acceleration  year  \
392  27.0          4         140.0          86    2790          15.6    82
393  44.0          4          97.0          52    2130          24.6    82
394  32.0          4         135.0          84    2295          11.6    82
395  28.0          4         120.0          79    2625          18.6    82
396  31.0          4         119.0          82    2720          19.4    82
```

```
     origin               name  mileage_rate
392        1  ford mustang gl             1
393        2        vw pickup             1
394        1    dodge rampage             1
395        1      ford ranger             1
396        1       chevy s-10             1
```

(c) Fit a linear support vector classifier to the data with various values of cost, in order to predict whether a car gets high or low gas mileage. Comment on your results.

[ ]:

```
[44]: X = data.iloc[:,:-2]
y = data["mileage_rate"]
X.head()
```

```
[44]:    mpg  cylinders  displacement  horsepower  weight  acceleration  year  \
0  18.0          8         307.0         130    3504          12.0    70
1  15.0          8         350.0         165    3693          11.5    70
2  18.0          8         318.0         150    3436          11.0    70
3  16.0          8         304.0         150    3433          12.0    70
4  17.0          8         302.0         140    3449          10.5    70

   origin
0       1
1       1
2       1
3       1
4       1
```

```
[45]: X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.8,␣
 ↪random_state = 0)
```

```
[46]: from sklearn import svm

cost = [0.0001,1000]
results={}
for c in cost:
    svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("linear_svc", LinearSVC(C=c, loss="hinge", random_state=42, max_iter =␣
 ↪5000)),
    ])
    svm_clf.fit(X_train, y_train)
    y_pred = svm_clf.predict(X_test)
    results[c] = y_pred
    print(y_pred)
    print("Accuracy:  ", accuracy_score(y_test, y_pred))
```

```
[1 1 0 1 1 0 1 1 0 1 1 0 1 0 0 1 1 0 1 1 0 0 1 1 1 0 1 1 0 0 0 1 1 1 0 1 1
 0 0 0 0 1 1 0 1 0 1 0 0 0 1 0 0 1 1 1 0 0 1 1 0 1 1 0 1 0 0 1 1 1 1 1 0 0
 0 0 1 0 1]
Accuracy:   0.8987341772151899
[1 0 0 1 1 0 1 1 0 0 1 0 1 0 0 1 0 0 1 1 0 0 0 1 1 0 1 1 0 0 0 1 0 1 0 1 1
 0 0 0 0 1 1 0 1 0 1 0 0 0 1 0 0 0 1 1 0 0 1 0 0 1 1 0 1 0 0 1 1 1 1 1 0 0
 1 0 1 0 1]
Accuracy:   1.0
```

```
C:\Users\shree\anaconda3\envs\cmps530\lib\site-
packages\sklearn\svm\_base.py:976: ConvergenceWarning: Liblinear failed to
converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
```

[47]:

```python
results = pd.DataFrame(results)
results
```

[47]:

	0.0001	1000.0000
0	1	1
1	1	0
2	0	0
3	1	1
4	1	1
..	…	…
74	0	1
75	0	0
76	1	1
77	0	0
78	1	1

[79 rows x 2 columns]

[48]:

```python
print("Low mileage predictions for C = 0.0001: ",len(results[results[0.
↪0001]==0]))
print("High mileage predictions for C = 0.0001: ",len(results[results[0.
↪0001]==1]))

print("Low mileage predictions for C = 1000.0: ",len(results[results[1000.
↪0]==0]))
print("High mileage predictions for C = 1000.0: ",len(results[results[1000.
↪0]==1]))
```

```
Low mileage predictions for C = 0.0001:  36
High mileage predictions for C = 0.0001:  43
Low mileage predictions for C = 1000.0:  42
High mileage predictions for C = 1000.0:  37
```

The C parameter tells the SVM optimization how much we want to avoid misclassifying each training example. The larger the value of C such as 1000, the smaller is the margin. C parameter adds a penalty for each misclassified data point. Penalty is not same for all misclassified examples. It is directly proportional to the distance to decision boundary.

For C = 0.0001, the penalty for misclassified points is low so a decision boundary with a large margin is chosen at the expense of a greater number of misclassifications.

However, for C = 1000, SVM tries to minimize the number of misclassified examples due to high penalty which results in a decision boundary with a smaller margin.

(d) Now repeat (c), this time using SVMs with radial and polynomial basis kernels, with different values of gamma and degree and cost. Comment on your results.

[49]:
```python
from sklearn.svm import SVC

gamma = [0.1, 5]
degree = [3, 10]
```

[50]:
```python
from sklearn.metrics import accuracy_score
#radial basis kernels
gamma1, gamma2 = 0.1, 5
C1, C2 = 0.001, 1000
hyperparams = (gamma1, C1), (gamma1, C2), (gamma2, C1), (gamma2, C2)

svm_clfs = []
for gamma, C in hyperparams:
    rbf_kernel_svm_clf = Pipeline([
            ("scaler", StandardScaler()),
            ("svm_clf", SVC(kernel="rbf", gamma=gamma, C=C))
        ])
    rbf_kernel_svm_clf.fit(X_train, y_train)
    y_pred = rbf_kernel_svm_clf.predict(X_test)
    print(f"Radial SVM for gamma = {gamma}, C = {C}", y_pred)
    print("Accuracy:  ", accuracy_score(y_test, y_pred))

#     svm_clfs.append(rbf_kernel_svm_clf)
```

```
Radial SVM for gamma = 0.1, C = 0.001 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1]
Accuracy:   0.46835443037974683
Radial SVM for gamma = 0.1, C = 1000 [1 1 0 1 1 0 1 1 0 0 1 0 1 0 0 1 0 0 1 1 0
0 0 1 1 0 1 1 0 0 0 1 0 1 0 1 1
 0 0 0 0 1 1 0 1 0 1 0 0 0 1 0 0 0 1 1 0 0 1 0 0 1 1 0 1 0 0 1 1 1 1 1 0 0
 0 0 1 0 1]
Accuracy:   0.9746835443037974
Radial SVM for gamma = 5, C = 0.001 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1]
```

```
Accuracy:   0.46835443037974683
Radial SVM for gamma = 5, C = 1000 [1 1 0 1 1 0 1 1 0 1 1 0 1 0 0 1 0 0 1 1 1 0
0 1 1 0 1 1 0 0 1 1 1 1 0 1 1
 0 0 0 0 1 1 0 1 0 1 0 0 0 1 0 0 0 1 1 0 0 1 0 0 1 1 0 1 0 0 1 1 1 1 1 0 0
 0 0 1 0 1]
Accuracy:   0.9240506329113924
```

[51]:
```
#polynomial basis kernels
for i in range(0,2):
    poly_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=degree[i], coef0=1, C=cost[i]))
    ])
    poly_kernel_svm_clf.fit(X_train, y_train.values.ravel())
    y_pred = poly_kernel_svm_clf.predict(X_test)
    print(f"Polynomial SVM for degrees = {degree[i]}, C = {cost[i]}", y_pred)
    print("Accuracy:  ", accuracy_score(y_test, y_pred))
```

```
Polynomial SVM for degrees = 3, C = 0.0001 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1]
Accuracy:   0.46835443037974683
Polynomial SVM for degrees = 10, C = 1000 [1 1 0 1 1 0 1 1 0 0 1 0 1 0 1 1 0 0 1
1 0 0 0 0 1 0 1 1 0 0 0 1 1 1 0 1 1
 0 0 0 0 1 1 0 1 0 1 0 0 0 1 0 0 0 1 1 0 0 1 0 0 1 1 0 1 0 0 1 1 1 1 1 0 0
 0 0 0 0 1]
Accuracy:   0.9240506329113924
```

0.2 Part 2

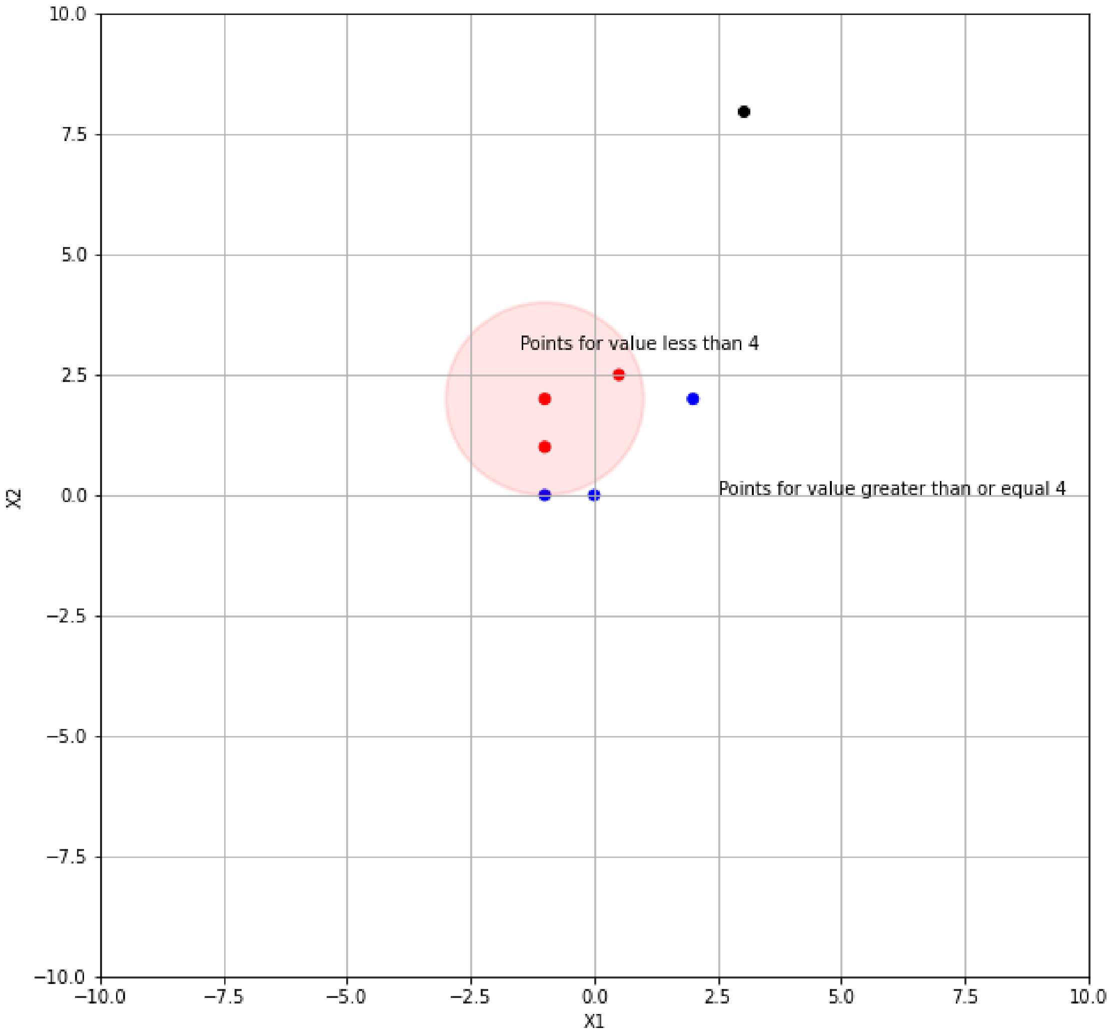
(a) Sketch the curve (1 + x1)2 + (2 −x2)2 = 4

[52]:
```
circle = plt.Circle((-1, 2), radius=2, facecolor='r', alpha=0.1, edgecolor='r',␣
 ↪linewidth=2.0)

fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(111)
ax.add_artist(circle)
plt.text(-1.5, 3, "Points for value less than 4", fontdict={'color':'black',␣
 ↪'size':10})
plt.text(2.5, 0, "Points for value greater than or equal 4", fontdict={'color':
 ↪'black', 'size':10})
plt.scatter([-1,-1,0.5, 0, 2, -1], [2, 1,2.5, 0, 2, 0], c=['r','r','r', 'b',␣
 ↪'b', 'b'])
plt.scatter([3], [8], c='black')
ax.set_xlim(-10, 10)
ax.set_ylim(-10, 10)
```

```python
ax.set_xlabel('X1')
ax.set_ylabel('X2')

plt.grid()
plt.show()
```



b.

- Points: (-1, 1), (2, 2), and (-1, 0) would be for value <= 4
- Point: (0,0), (-1, 1) and (0.5, 2.5) for value > 4

c. Since the three points (-1, 1), (2, 2), and (0, 0) were already already plotted before, testing for (3,8), it falls on blue class

- (-1,1): red class
- (2,2): red class

- (0,0): blue class
- (3,8): blue class

```
[ ]:
```