SVM- Diagnostic

Siyuan Zhang

1 Resources and Citation

Student: Siyuan Zhang
Language: Python
Code: https://github.com/Siyuan-gwu/Machine-Learning-SVM-Diagnostic
Instruction: The instruction to run the code is in github (readme)
Resource:

1. Breast Cancer Wisconsin (Diagnostic) Data Set
2. https://towardsdatascience.com/support-vector-machine-python-example-d67d9b63f1c8

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import svm
import time
```

2 Dataset details

The dataset includes data of 30 features except 'id', but only 10 real valued features, including:

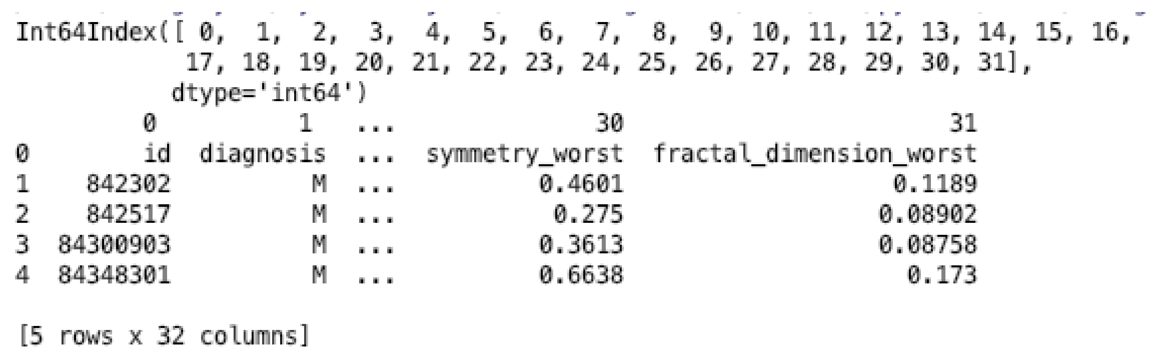
1) ID number
2) Diagnosis (M = malignant, B = benign)
3-32)

Ten real-valued features are computed for each cell nucleus:

a) radius (mean of distances from center to points on the perimeter)
b) texture (standard deviation of gray-scale values)
c) perimeter
d) area
e) smoothness (local variation in radius lengths)
f) compactness (perimeter^2 / area - 1.0)
g) concavity (severity of concave portions of the contour)
h) concave points (number of concave portions of the contour)
i) symmetry
j) fractal dimension ("coastline approximation" - 1)

Inspect the Dataset

```
#read the dataset
data = pd.read_csv('wdbc.data', header=None)
print (data.columns)
print (data.head(5))
```

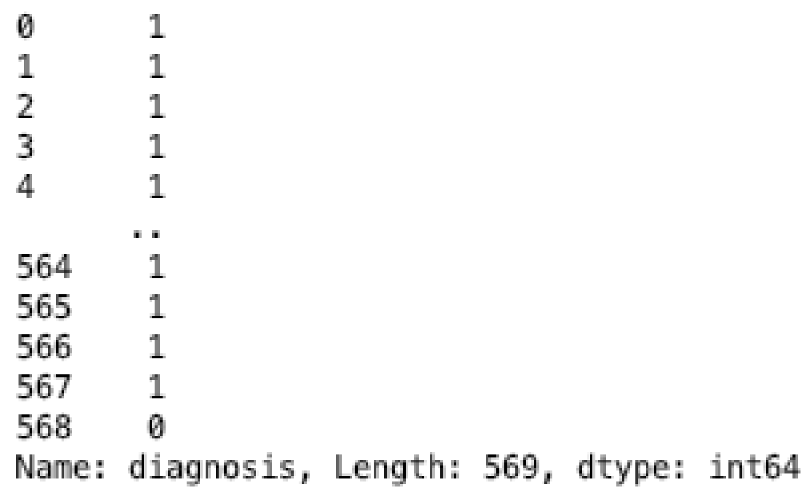


```
Int64Index([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
            17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31],
           dtype='int64')
          0          1  ...              30                       31
0        id  diagnosis  ...  symmetry_worst  fractal_dimension_worst
1    842302          M  ...          0.4601                   0.1189
2    842517          M  ...           0.275                  0.08902
3  84300903          M  ...          0.3613                  0.08758
4  84348301          M  ...          0.6638                    0.173

[5 rows x 32 columns]
```

Data Pre-processing

During the inspect, I noticed that the Diagnosis (M = malignant, B = benign).
So, here I replace the M to 1 and B to 0, which makes the training and testing easier.

```
# replace 'M' and 'B' with 1 and 0
data['diagnosis'] = data['diagnosis'].map({'M':1,'B':0})
print (data['diagnosis'])
```



```
0      1
1      1
2      1
3      1
4      1
      ..
564    1
565    1
566    1
567    1
568    0
Name: diagnosis, Length: 569, dtype: int64
```

Then, the column 'id' is useless, I deleted the 'id' column to make the data more concise.

```
# drop the column 0, which contains 'id' (useless)
data.drop('id', axis=1, inplace=True)
print (data.head(5))
```

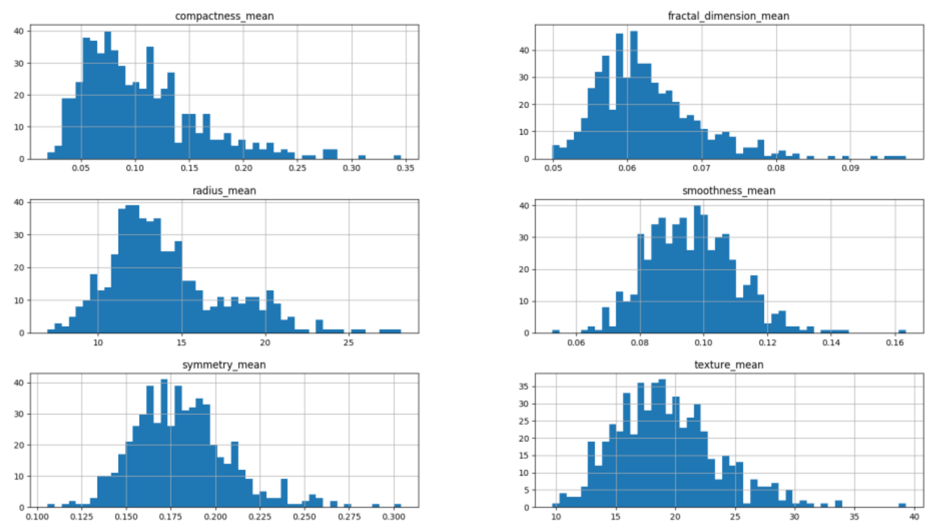
```
   diagnosis  radius_mean  ...  symmetry_worst  fractal_dimension_worst
0          1        17.99  ...          0.4601                  0.11890
1          1        20.57  ...          0.2750                  0.08902
2          1        19.69  ...          0.3613                  0.08758
3          1        11.42  ...          0.6638                  0.17300
4          1        20.29  ...          0.2364                  0.07678

[5 rows x 31 columns]
```

Dataset Visualization

We can plot some main features of dataset to visualize the data as histogram and chart

eg.
'radius_mean','texture_mean','smoothness_mean','compactness_mean','symmetry_mean', 'fractal_dimension_mean'

```
# visualization
data[feature].hist(bins=50, figsize=(20, 15))
plt.show()
```



Data Splitting

I split the dataset into training data (70%) and testing data (30%), used the training data to get the model and predict the result with testing data.

```
#split dataset
from sklearn.model_selection import train_test_split
train, test = train_test_split(data,test_size=0.3,train_size=0.7)
```

Then, I selected 6 features to train the model. With a small number of features, the model can represent the characteristic of the data, which can also enhance the generalization ability of the classifier as well as avoid data over-fitting.

```
feature = ['radius_mean','texture_mean', 'smoothness_mean','compactness_mean','symmetry_mean',
'fractal_dimension_mean']
# 2, 3, 6, 7, 10, 11
print (train.shape)
train_X = train[feature]
train_y = train['diagnosis']
test_X = test[feature]
test_y = test['diagnosis']
print (train_X.head(5))
```

```
     radius_mean  texture_mean  ...  symmetry_mean  fractal_dimension_mean
82         25.22         24.91  ...         0.1829                 0.06782
547        10.26         16.58  ...         0.1669                 0.06714
240        13.64         15.60  ...         0.1717                 0.05660
321        20.16         19.66  ...         0.1928                 0.05096
436        12.87         19.54  ...         0.1861                 0.06347

[5 rows x 6 columns]
```

Above is the training data.

Data Normalization

Before training, I need to normalize the data so that the data is on the same level, avoiding data errors caused by dimensional problems.

```
# min-max normalization
def MaxMinNormalization(x):
    """[0,1] normaliaztion"""
    x = (x - np.min(x)) / (np.max(x) - np.min(x))
    return x
train_X = MaxMinNormalization(train_X)
test_X = MaxMinNormalization(test_X)
print (train_X)
```

```
     radius_mean  texture_mean  ...  symmetry_mean  fractal_dimension_mean
10      0.406800      0.457558  ...       0.236364                0.147641
369     0.700702      0.412242  ...       0.385859                0.240944
543     0.268927      0.620561  ...       0.286869                0.165333
63      0.070850      0.140345  ...       0.646970                0.414280
460     0.458810      0.589787  ...       0.370202                0.270640
..           ...           ...  ...            ...                     ...
552     0.247338      0.666892  ...       0.241919                0.135004
299     0.136451      0.452486  ...       0.320707                0.328559
392     0.380796      0.346973  ...       0.438889                0.368155
377     0.281193      0.625634  ...       0.182323                0.161542
27      0.533880      0.356442  ...       0.321717                0.148062

[398 rows x 6 columns]
```

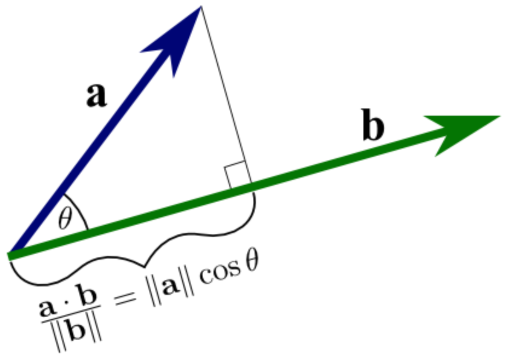
Here I use the min-max normalization.
The min-max normalization method is a linear transformation of the original data. Let minA and maxA be the minimum and maximum values of the attribute A, respectively, and normalize an original value x of A by min-max to the value x' in the interval [0, 1].

Equation: $$x' = \frac{x - x\_min}{x\_\max - x\_min}$$

3 Algorithm Description

The LSVM algorithm will select a line that not only separates the two classes but stays as far away from the closest samples as possible. In fact, the “support vector” in “support vector machine” refers to two position vectors drawn from the origin to the points which dictate the decision boundary.

Suppose, we had a vector w which is always normal to the hyperplane (perpendicular to the line in 2 dimensions). We can determine how far away a sample is from our decision boundary by projecting the position vector of the sample on to the vector w. As a quick refresher, the dot product of two vectors is proportional to the projection of the first vector on to the second.


a
b
θ
$\frac{\mathbf{a}\cdot\mathbf{b}}{\|\mathbf{b}\|} = \|\mathbf{a}\|\cos\theta$

If it is a positive sample, then return a value greater than or equal to 1, else if is is a negative sample, return a value smaller than or equal to -1.

$$\vec{w}\cdot\vec{x_+} + b \geq 1 \quad \vec{w}\cdot\vec{x_-} + b \leq -1$$

We need to maximize the margin, we subtract the first support vector from the one below it and the multiply the result by the unit vector of w which is always perpendicular to the decision boundary.

$$width = (\vec{x_+} - \vec{x_-})\cdot\frac{\vec{w}}{\|w\|} \quad width = \frac{2}{\|w\|}$$

Therefore, in order to select the optimal decision boundary, we must maximize the equation above (width). Maximize the width equal to minimize the ||w||.
In SVM, we use Lagrangian to minimize the function. The Lagrange tells us to subtract the cost function by the summation over all the constraints where each of those constraints will be multiplied by some constant alpha, etc.

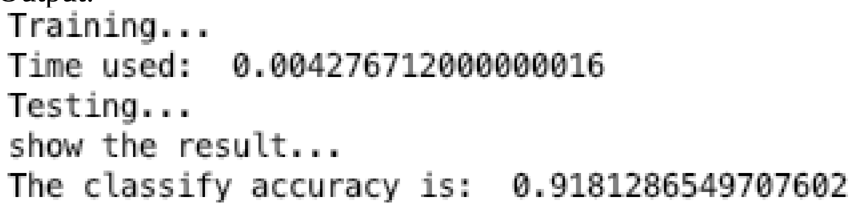
4 Algorithm Results

Algorithm accuracy

Here, I used the sklearn.svm to train the model and predict the test accuracy.

```
# Training...
# ------------------------
print ("Training...")
model = svm.SVC()
start = time.thread_time()
model.fit(train_X, train_y)
## step 3: testing
print ("Testing...")
prediction = model.predict(test_X)
end = time.thread_time()
print ('Time used: ', (end - start))
## step 4: show the result
print ("show the result...")
errorCount = 0
for y_res, y_predict in zip(test_y, prediction):
    if y_res != y_predict:
        errorCount += 1
print ('The classify accuracy is: ', (len(test_y) - errorCount) / len(test_y))
```

Output:



```
Training...
Time used:  0.004276712000000016
Testing...
show the result...
The classify accuracy is:  0.9181286549707602
```

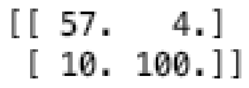
The total accuracy is 91.81%.

Confusion Matrix

Here, I define a function to print a 2x2 confusion matrix, which contains (True Positive), (True Negative), (False Positive), (False Negative).

```
def confusion_matrix(y_true, y_pred):
    matrix = np.zeros([2, 2])
    for y_true, y_pred in zip(y_true,y_pred):
        if y_true == 1 and y_pred == 1:
           matrix[0][0] += 1
        if y_true == 0 and y_pred == 1:
            matrix[0][1] += 1
        if y_true == 0 and y_pred == 0:
            matrix[1][1] += 1
        if y_true == 1 and y_pred == 0:
            matrix[1][0] += 1

    return matrix

c_matrix = confusion_matrix(test_y, prediction)
print (c_matrix)
```

Then the output is:



```
[[ 57.    4.]
 [ 10. 100.]]
```

The table below shows the true positive, true negative, false positive and false negative. In total 171 test samples, there are 57 true positives and 100 false negatives. So, the accuracy is (57 + 100) / 171 = 91.81%

	TRUE	**FALSE**
Positive	**57**	**4**
Negative	**10**	**100**

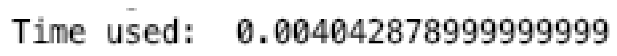
5 Runtime

It is very hard to measure the time complexity for SVM. For linear SVMs, at training time I need to estimate the vector w and bias b by solving a quadratic problem, and at test time prediction is linear in the number of features and constant in the size of the training data. For kernel SVMs, at training time I need to select the support vectors and at test time the complexity is linear on the number of the support vectors (which can be lower bounded by training set size * training set error rate) and linear on the number of features. We say n is the number of points and d is the number of dimensions.
The time complexity is $O(\max(n, d) * \min\,(n, d)\text{^}2)$.

Citation: *Chapelle, Olivier. "Training a support vector machine in the primal." Neural Computation 19.5 (2007): 1155-1178.*

Program runtime:

```
# Training...
# ------------------------
print ("Training...")
model = svm.SVC()
start = time.thread_time()
model.fit(train_X, train_y)
## step 3: testing
print ("Testing...")
prediction = model.predict(test_X)
end = time.thread_time()
print ('Time used: ', (end - start))
```



```
Time used:  0.004042878999999999
```

The program takes 0.0040429seconds.