lab1_iris_partial

April 21, 2021

1 Lab: Iris Flower Classification using Decision Theory

In this exercise, we will use decision theory on a simple multi-variable classification problem. In doing the exercise, you will learn to:

- Load a pre-installed dataset in the `sklearn` package.
- Estimate parameters of a multi-variable Gaussian from data
- Make multi-class predictions using linear and quadratic discriminants derived from the Gaussian parameters
- Evaluate the accuracy of the predictions on test data

For submission: * Complete all sections labeled `#TODO` * Run the notebook and print to PDF. * Submit the PDF in CCLE. No other formats accepted. Do not submit the jupyter notebook.

We load the following packages.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
matplotlib.rcParams.update({'font.size': 16})
```

1.1 Loading and Visualizing the Data

In this lab, we will use the classic dataset Iris flower dataset. The problem is to estimate the type of iris (‘setosa’ ‘versicolor’ ‘virginica’) from four features of the Iris flower. Since the data is widely-used in machine learning classes, it is included in the `sklearn` package in python. You can download the data with following command:

[2]:
```
from sklearn.datasets import load_iris
data = load_iris()
```

Set`X=data.data` and `y=data.target`. The matrix `X[i,j]` will then be the value of feature `j` in sample `i` and `y[i]` will be the index of the class for sample `i`. Also, print: * Number of samples, * Number of features per sample * Number of classes

[3]:
```
X = data.data
y = data.target

print('Number of samples: ' + str(X.shape[0]))
print('Number of features per sample: ' + str(X.shape[1]))
```

```python
print('Number of class: ' + str(np.unique(y)))
```

```
Number of samples: 150
Number of features per sample: 4
Number of class: [0 1 2]
```

Print the feature and target names in `data.feature_names` and `data.target_names`

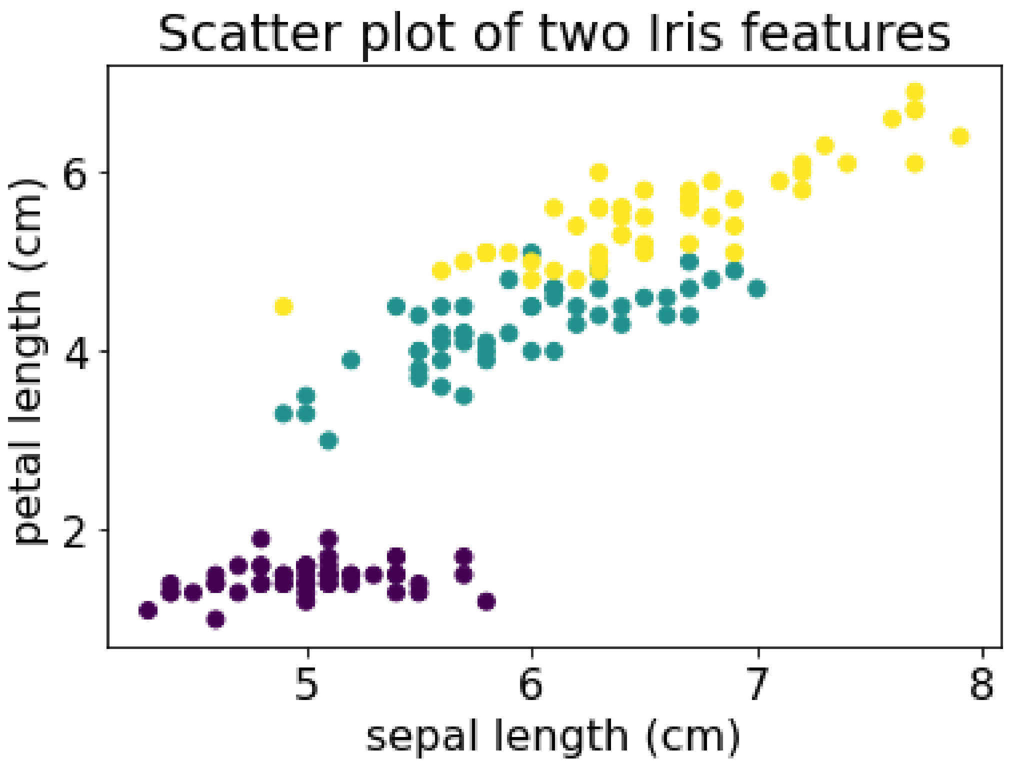
[4]:
```python
print('Feature names: ' + str(data.feature_names))
print('Target names: ' + str(data.target_names))
```

```
Feature names: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
'petal width (cm)']
Target names: ['setosa' 'versicolor' 'virginica']
```

To visualize the data, create a *scatter* plot of two of the four features: `(sepal length, petal length)`. In a scatter plot, each point is plotted as some marker, say a small circle. Different colors are used for different class labels. You can create a scatter plot using the `plt.scatter()` command. Make sure you label your axes using the `plt.xlabel()` and `plt.ylabel()` axes.

If you did the plot correctly, you should see that you can separate the points well even just using two features.

[5]:
```python
plt.scatter(X[:, 0], X[:, 2], c = y)
plt.xlabel('sepal length (cm)')
plt.ylabel('petal length (cm)')
plt.title('Scatter plot of two Iris features')
plt.show()
```


Scatter plot of two Iris features
petal length (cm)
sepal length (cm)
2
4
6
5
6
7
8

1.2 Classifier Based on a Linear Discriminator

Before trying any classification methods, it is necessary to split the data into two components: * Training samples: Used for fitting the classifier models * Test samples: Used for testing the classifier

We will discuss splitting the training and test data in detail later in the class. But, the reason for the splitting is that we want to test the classifier on samples not used in training. This ensures we see how well it works on *new* samples that have not been seen.

For this purpose, divide the data into 75 samples `(Xtr,ytr)` for training and 75 samples `(Xts,yts)` for test. You must randomly shuffle the samples before splitting – do not pick the first 75 for training. You can do the splitting manually or use the `sklearn.model_selection.train_test_split` function. If you use `train_test_split()`, set `shuffle=True`.

[6]:
```
from sklearn.model_selection import train_test_split
np.random.seed(123)

# split data into training and test set
Xtr, Xts, ytr, yts = train_test_split(X, y, train_size = 75, test_size = 75,␣
↪shuffle = True)
```

We will first try using a linear discriminator. For linear discriminator we first need to estimate the sample mean of each class. Complete the following function which returns:

```
mu[:,j]  = sample mean of X[i,:]  for samples with y[i]=j
```

[7]:
```
def fit_lin(X, y):
    mu = np.zeros([4, 3])
    for i in range(4):
        for j in range(3):
            mu[i, j] = np.mean(X[y == j, i])
    return mu
```

Run the function on the training data to get the sample means.

[8]:
```
mu = fit_lin(Xtr, ytr)
print(mu)
```

```
[[5.05217391 5.99333333 6.54545455]
 [3.47391304 2.84       2.94545455]
 [1.45217391 4.33333333 5.54090909]
 [0.24782609 1.37       2.02727273]]
```

Assuming a Gaussian model with equal and i.i.d. covariance matrices, the optimal estimator is given by the linear discrimantor: Given a test sample `x`, we compute:

```
 g[j]  = mu[:,j].dot(x) - 0.5*sum(mu[:,j]**2)
```

for each class `j`. Then, we select `yhat = argmax_j g[j]`. Complete the following code which takes a matrix of data samples `X` to compute a vector of class predictions `yhat`.

[9]:
```
def predict_lin(X, mu):
    g = []
    for j in range(3):
        g.append(mu[:, j].dot(X.T) - 0.5 * sum(mu[:, j] ** 2))
    yhat = np.argmax(g, axis = 0)
    return yhat
```

Test the linear classifier on the test data. Specifically, estimate `yhat` and measure the accuracy, which is the number of samples on which the classifier was correct. If you did everything correctly, you should get an accuracy of around 90%. But, it may be a little higher or lower depending on the random training / test split.

[10]:
```
yhat = predict_lin(Xts, mu)
acc = np.mean(yhat == yts)
print(acc)
```

```
0.9466666666666667
```

1.3 Quadratic Discriminator

Now, we will try a more sophisticated classifier. In this case, we will estimate both the sample mean and covariance matrix. Complete the following code that computes:

- `mu`: Array where `mu[:,j]` is the sample mean for the samples with `y[i]==j`. This is identical to the code you have above.
- `S`: Array of covariance matrices where `S[:,:,j]` is the sample covariance matrix for the samples with `y[i]==j`. You can use the `np.cov()` method.

[11]:
```python
def fit_quad(X, y):
    mu = np.zeros([4, 3])
    S = np.zeros([3, 4, 4])
    for j in range(3):
        S[j, :, :] = np.cov(X[y == j, :].T)
        for i in range(4):
            mu[i, j] = np.mean(X[y == j, :][:, i])
    return mu, S
```

Fit the quadratic model on the training data.

[12]:
```python
mu, S = fit_quad(Xtr, ytr)
print(mu)
print(S)
```

```
[[5.05217391 5.99333333 6.54545455]
 [3.47391304 2.84       2.94545455]
 [1.45217391 4.33333333 5.54090909]
 [0.24782609 1.37       2.02727273]]
[[[ 0.1226087   0.09369565  0.0126087   0.00466403]
  [ 0.09369565  0.12837945 -0.00448617  0.00675889]
  [ 0.0126087  -0.00448617  0.0226087   0.00420949]
  [ 0.00466403  0.00675889  0.00420949  0.01079051]]

 [[ 0.26685057  0.06372414  0.18229885  0.04944828]
  [ 0.06372414  0.0742069   0.06931034  0.03468966]
  [ 0.18229885  0.06931034  0.23264368  0.07206897]
  [ 0.04944828  0.03468966  0.07206897  0.04010345]]

 [[ 0.34640693  0.09878788  0.25281385  0.05536797]
  [ 0.09878788  0.10354978  0.08614719  0.07012987]
  [ 0.25281385  0.08614719  0.26348485  0.0521645 ]
  [ 0.05536797  0.07012987  0.0521645   0.08683983]]]
```

Given Gaussian models in each class the optimal decision rule is the following *quadratic* decision rule: Given a sample `x`, we compute the discrimant,

```
g[j] = 0.5*(x-mu[:,j).T.dot(Sinv[:,:,j]).dot(x-mu[:,j]) + 0.5*log(det(S[:,:,j]))
```

where `Sinv[:,:,j]` is the matrix inverse of `S[:,:,j]`. Then, we take `yhat = argmin_j g[j]`. Complete the following code to compute the predictions based on the quadratic discriminats.

```
[13]: def predict_quad(X, mu, S):
          """
          Quadratic discriminator using discriminants:

              g = -ln p(x|mu,X) = 0.5*(x-mu)*Sinv*(x-mu) + 0.5*log det(S)
          """

          sinv = np.zeros([3, 4, 4])
          logdet = np.array([])
          for ic in range(3):
              sinv[ic] = np.linalg.inv(S)[ic]
              logdet = np.append(logdet, np.log(np.linalg.det(S[ic])))

          g = np.zeros([3, 75])
          for j in range(3):
              for i in range(75):
                  g[j, i] = 0.5 * (X[i, :] - mu[:, j]).T.dot(sinv[j, :, :]).dot(X[i, :
       ↪] - mu[:, j]) + 0.5 * logdet[j]
          yhat = np.argmin(g, axis = 0)
          return yhat
```

Test the quadratic discriminator on the test data. You should get around 98% accuracy depending on the train / test split.

```
[14]: yhat = predict_quad(Xts, mu, S)
      acc = np.mean(yhat == yts)
      print(acc)
```

```
0.96
```