Logistic Regression & Clustering

Avijit Bose
Assistant professor
Department of Computer Science & Engineering, MCKVIE,Liluah,Howrah-711204

CASE STUDY

- In this we are taking a case study whether one is going to pass or fail an exam.
- The parameters are number of hours they study every day or the number of hours they have spent working with a tutor.
- We will import sklearn libraries and import a csv file which we have downloaded from github.

Sklearn

- import pandas as pd
  import numpy as np
  from sklearn import metrics
  from sklearn.linear_model import
  LogisticRegression
  from sklearn.model_selection import
  train_test_split

Loading the data

- df = pd.read_csv(‘Student-Pass-Fail-Data.csv’)
  df.head()

Feature Extraction

- Note that the loaded data has two features—namely, Self_Study_Daily and Tuition_Monthly. Self_Study_Daily indicates how many hours the student studies daily at home, and Tuition_Monthly indicates how many hours per month the student is taking private tutor classes.

Feature Extraction Continued

- Apart from these two features, we have one label in the dataset named Pass_or_Fail. This label has two values—either 1 or 0. A value of 1 indicates pass and a value of 0 indicates fail.
- Next we port the Pass_Or_Fail column to the y variable:
- x = df.drop(‘Pass_Or_Fail’,axis = 1)
  y = df.Pass_Or_Fail

Feature Extraction continued

- After the data is loaded, our next step is to separate the features and label them as x and y respectively. Note that we’re using a drop method to drop the Pass_Or_Fail label and assign the rest of the columns to the x variable.

Training & Testing Phase

- Once the features and labeled are separated, we’re ready to split the data into train and test sets. This will separate 25%(default value) of the data into a subset for testing part and the remaining 75% will be used for our training subset.

Training & Testing Phase

- x_train, x_test, y_train, y_test = train_test_split(x, y, random_state=4)
- At this stage, we're ready to create our logistic regression model. We'll do this using the LogisticRegression class we imported in the beginning.
- logistic_regression = LogisticRegression()

Training the Model

- Once the model is defined, we can work to fit our data. We’re going to use the fit method on the model to train the data. Note that the fit method takes two parameters here: variables x_train and y_train. We created these variables we created in some previous code using the train_test_split function.

Training The Model

- logistic_regression.fit(x_train,y_train)
- 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=None,solver='warn',tol=0.0001,verbose=0,warm_start=False)

Prediction

- Once model training is complete, its time to predict the data using the model. For this, we’re going to use the predict method on the model and pass the x_test values for predicting. We’re storing the predicted values in the y_pred variable.

Finding Accuracy

- We need to find the accuracy of our model in order to evaluate its performance. For this, we’ll use the accuracy_score method of the metrics class, as shown below:
- accuracy = metrics.accuracy_score(y_test, y_pred)
  accuracy_percentage = 100 * accuracy
  accuracy_percentage

K-Means Clustering

- Basic idea:-
- Choose the same number of random points on the 2D canvas as centroids. Calculate the distance of each data point from the centroids. Allocate the data point to a cluster where its distance from the centroid is minimum. Recalculate the new centroids.

Importing the dataset

- Import the toy dataset from scikit-learn
- Visualize the data points.

K- Means Clustering Algorithm

- Now that we have learned how the k-means algorithm works, let’s apply it to our sample dataset using the KMeans class from scikit-learn's cluster module:

K-Means Clustering Algorithm

```
from sklearn.cluster import KMeans
```

```
km = KMeans(
n_clusters=3, init='random',
n_init=10, max_iter=300,
tol=1e-04, random_state=0
)
y_km = km.fit_predict(X)
```

K-Means Clustering Algorithm

- Using the preceding code, we set the number of desired clusters to 3. We set n_init=10 to run the k-means clustering algorithms 10 times independently with different random centroids to choose the final model as the one with the lowest SSE. Via
  the max_iter parameter, we specify the maximum number of iterations for each single run (here, 300).

K-Means Clustering Algorithm

- Note that the k-means implementation in scikit-learn stops early if it converges before the maximum number of iterations is reached. However, it is possible that k-means does not reach convergence for a particular run, which can be problematic (computationally expensive) if we choose relatively large values for max_iter.

K-Means Clustering Algorithm

- One way to deal with convergence problems is to choose larger values for tol, which is a parameter that controls the tolerance with regard to the changes in the within-cluster sum-squared-error to declare convergence. In the preceding code, we chose a tolerance of 1e-04 (= 0.0001).

K-Means Clustering Algorithm

- A problem with k-means is that one or more clusters can be empty. However, this problem is accounted for in the current k-means implementation in scikit-learn. If a cluster is empty, the algorithm will search for the sample that is farthest away from the centroid of the empty cluster. Then it will reassign the centroid to be this farthest point.

K-Means Clustering Algorithm

- Now that we have predicted the cluster labels y_km, let’s visualize the clusters that k-means identified in the dataset together with the cluster centroids. These are stored under the cluster_centers_ attribute of the fitted KMeans object:

K-Means Clustering Algorithm

```
# plot the 3 clusters
plt.scatter(
X[y_km == 0, 0], X[y_km == 0, 1],
s=50, c='lightgreen',
marker='s', edgecolor='black',
label='cluster 1'
)

plt.scatter(
X[y_km == 1, 0], X[y_km == 1, 1],
s=50, c='orange',
marker='o', edgecolor='black',
label='cluster 2'
)
```

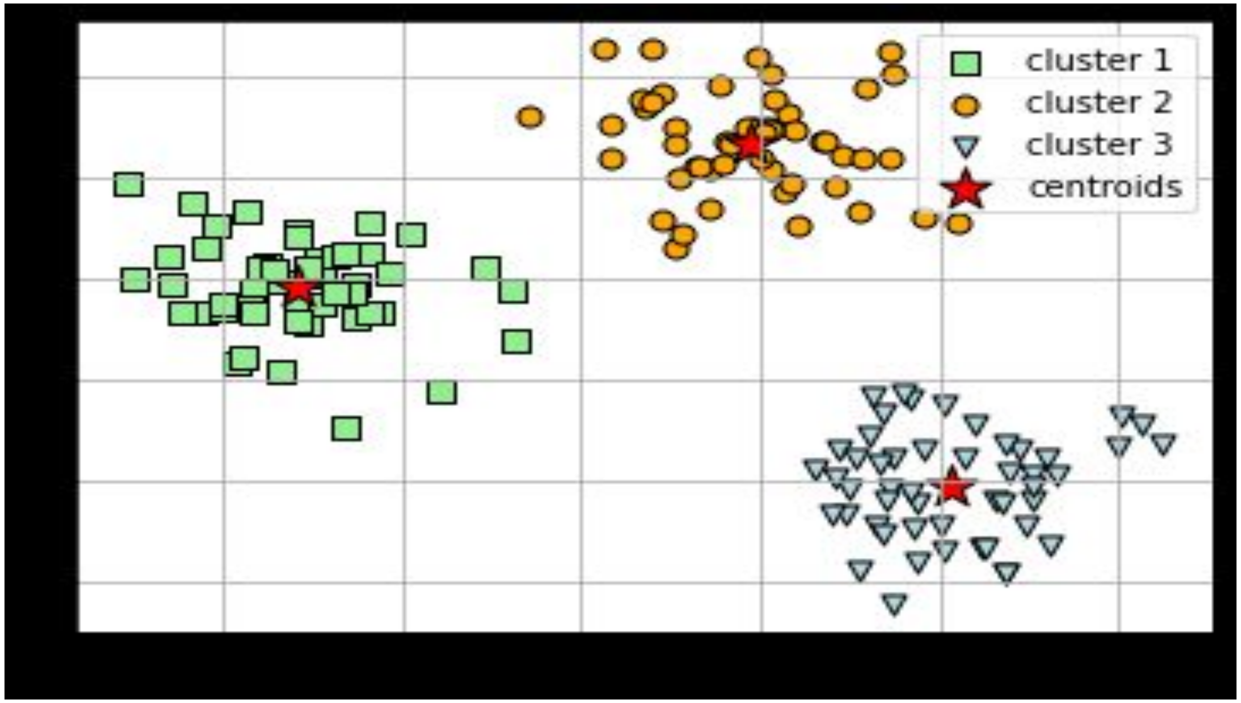
K-Means Clustering Algorithm

```
plt.scatter(
X[y_km == 2, 0], X[y_km == 2, 1],
s=50, c='lightblue',
marker='v', edgecolor='black',
label='cluster 3'
)

# plot the centroids
plt.scatter(
km.cluster_centers_[:, 0],
km.cluster_centers_[:, 1],
s=250, marker='*',
c='red', edgecolor='black',
label='centroids'
)
plt.legend(scatterpoints=1)
plt.grid()
plt.show()
```

Figure Looks like

- This is the clustering pattern.



K-Means Clustering

- In the resulting scatterplot, we can see that k-means placed the three centroids at the center of each sphere, which looks like a reasonable grouping given this dataset.