



VIT[®]

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

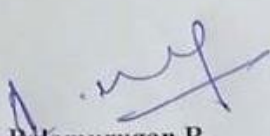
School of Computer Science and Engineering (SCOPE)

Fall Semester 2023-24

CSE4020: MACHINE LEARNING

LAB MANUAL

Faculty: Prof. Balamurugan R, Prof. Abdulgaffar H, Prof. Jayakumar K, and Prof. Aarth S L


Prof. Balamurugan R

(Machine Learning Lab Coordinator)


Prof. Rajkumar S

HoD, Dept. of Analytics

Head of the Department

**Department of Analytics
School of Computer Science & Engineering (SCOPE)
Vellore Institute of Technology (VIT)
Deemed to be University under section 3 of the UGC Act, 1956
Vellore – 632 014, Tamil Nadu, India**

CHALLENGING ASSIGNMENTS

S.NO	TITLE	PAGE NO
1	Data exploration and preprocessing in machine learning	1
2	Implement a tree-based algorithm to classify the flower species	4
3	Implement Logistic Regression for predict client subscription	7
4	Implement classification using Multilayer perceptron with back-propagation	13
5	Implement Support Vector Machines for bank note authentication	18
6	Implement K Nearest Neighbors algorithm to classify flower species	21
7	Implement Random Forest algorithm to classify flower species	24
8	Implement Ada boost algorithm to classify flower species	27
9	Implement K-means Clustering to Find Natural Patterns in Data	29
10	Implement Principle Component Analysis for Dimensionality Reduction	32

Assignment 1

Problem Statement:

Suppose you have height and weight data for a group of people. For example: Heights are in feet, like 6.5, and weight is in grams, like 80000. In many machine learning situations, you want to normalize the data — scale the data so that the values in different columns have roughly the same magnitude so that large values (like the weight) don't overwhelm smaller values (like the heights). Create a raw data of minimum 40 records of height and weight in above mentioned format and use Min-Max Normalization to normalize the weights in the range from as well as use Z-score to normalize the weights.

Concept to be applied:

Feature scaling becomes necessary when dealing with datasets containing features that have different ranges, units of measurement, or orders of magnitude. In such cases, the variation in feature values can lead to biased model performance or difficulties during the learning process.

There are several common techniques for feature scaling, including standardization, normalization, and min-max scaling. These methods adjust the feature values while preserving their relative relationships and distributions.

Min-Max Scaling is a simple method of normalization that scales the values between 0 and 1. It works by subtracting the minimum value from each value in the dataset and then dividing by the range of the dataset (i.e., maximum value minus minimum value).

$$X_{norm} = (X - X.min()) / (X.max() - X.min())$$

where `X` is the original dataset, `X_min` is the minimum value of `X`, and `X_max` is the maximum value of `X`.

Z-score is a variation of scaling that represents the number of standard deviations away from the mean. You would use z-score to ensure your feature distributions have mean = 0 and std = 1. It's useful when there are a few outliers, but not so extreme that you need clipping.

$$X_{norm} = (X - X.mean()) / X.std()$$

where `X` is the original dataset, `X_mean` is the mean value of `X`, and `X_std` is the standard deviation of `X`.

Procedure/Steps:

Step 1: First load a dataset and extract the numerical features we want to normalize.

Step 2: Create an instance of StandardScaler and use its fit_transform method to normalize the data.

Step 3: Print the normalized data.

Code:

```
import random
random.seed(42) # for reproducibility
# Generate random height and weight data for 40 records
data = []
for _ in range(40):
    height = round(random.uniform(4.5, 7.0), 2) # Height in feet (between 4.5 and 7.0 feet)
    weight = random.randint(50000, 100000) # Weight in grams (between 50000 and 100000 grams)
    data.append((height, weight))

# Display the raw data
print("Raw Data:")
print("Height (feet)  Weight (grams)")
for record in data:
    print(f"{record[0]:<14} {record[1]:<14}")
```

#Normalizing the weights using Min-Max normalization and Z-score normalization

```
import numpy as np

# Extract weights from the data
weights = np.array([record[1] for record in data])

# Min-Max Normalization
min_weight = np.min(weights)
max_weight = np.max(weights)

minmax_normalized_weights = (weights - min_weight) / (max_weight - min_weight)

# Z-score Normalization
mean_weight = np.mean(weights)
stddev_weight = np.std(weights)

zscore_normalized_weights = (weights - mean_weight) / stddev_weight

# Display the normalized weights
print("\nNormalized Weights (Min-Max Normalization):")
for i, weight in enumerate(minmax_normalized_weights):
    print(f"Record {i+1}: {weight:.4f}")

print("\nNormalized Weights (Z-score Normalization):")
for i, weight in enumerate(zscore_normalized_weights):
    print(f"Record {i+1}: {weight:.4f}")
```

Input:

Raw Data:
Height (feet) Weight (grams)

6.1	51639
6.35	66049
5.06	98265
4.76	98540
6.73	55697
5.98	52082
4.57	64328
5.08	89453
4.57	63031
6.29	95962
5.86	64446

Output:

Normalized Weights (Min-Max Normalization):

Record 1: 0.0252

Record 2: 0.3247

Record 3: 0.9943

Record 4: 1.0000

Record 5: 0.1096

Record 6: 0.0344

Record 7: 0.2890

Record 8: 0.8111

Normalized Weights (Z-score Normalization):

Record 1: -1.5740

Record 2: -0.6133

Record 3: 1.5344

Record 4: 1.5528

Record 5: -1.3034

Record 6: -1.5444

Record 7: -0.7280

Record 8: 0.9470

Assignment 2

Problem Statement:

Implement Decision tree classifier for iris dataset (load_iris) and evaluate the algorithm with precision, recall sensitivity and F1-score.

Concept to be applied:

Decision Trees are a type of Supervised Machine Learning (that is you explain what the input is and what the corresponding output is in the training data) where the data is continuously split according to a certain parameter. The tree can be explained by two entities, namely decision nodes and leaves. The leaves are the decisions or the final outcomes. And the decision nodes are where the data is split.

There are two main types of Decision Trees:

1. **Classification trees** (Yes/No types)

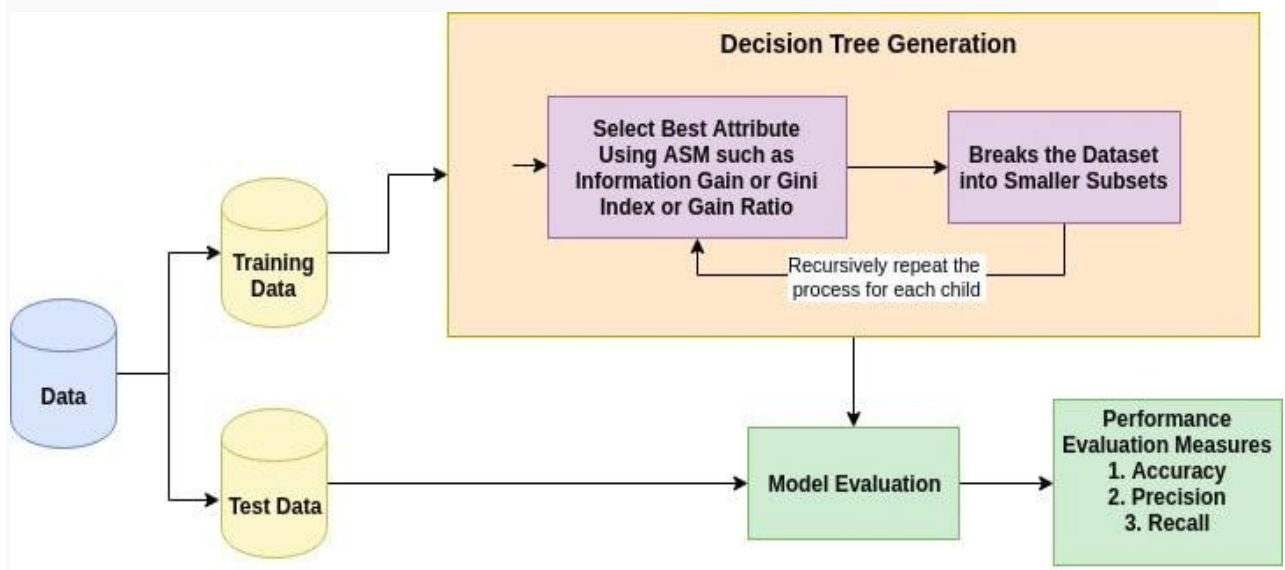
What we've seen above is an example of classification tree, where the outcome was a variable like 'fit' or 'unfit'. Here the decision variable is **Categorical**.

2. **Regression trees** (Continuous data types)

Here the decision or the outcome variable is **Continuous**, e.g. a number like 123.

Working

Now that we know what a Decision Tree is, we'll see how it works internally. There are many algorithms out there which construct Decision Trees, but one of the best is called as **ID3 Algorithm**. ID3 Stands for **Iterative Dichotomiser 3**.



Procedure/Steps:

Step 1: Load required packages and the dataset using Pandas

Step 2: Take a look at the shape of the dataset

Step 3: Define the features and the target

Step 4: Split the dataset into train and test sets using sklearn

Step 5: Build the model with the help of the decision tree classifier function

Step 6: Visualize the decision tree

Step 7: Predict the values

Step 8: Compare y_test and y_pred

Step 9: Find the confusion matrix and other metric parameters for this decision tree classification model

Code:

#Importing required libraries

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn import metrics
```

#Loading the iris data

```
data = load_iris()
print('Classes to predict: ', data.target_names)
print('Classes to predict: ', data.feature_names)
```

#Extracting data attributes

```
X = data.data
```

#Extracting target/ class labels

```
y = data.target
```

```
print('Number of examples in the data:', X.shape)
```

#First four rows in the variable 'X'

```
print(X[:4])
```

#Using the train_test_split to create train and test sets.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=None, train_size = 0.80)
```

```

#Importing the Decision tree classifier from the sklearn library.
#from sklearn.tree import DecisionTreeClassifier

clf = DecisionTreeClassifier(criterion='gini')

#Training the decision tree classifier.

clf.fit(X_train, y_train)

#Predicting labels on the test set.

y_pred = clf.predict(X_test)

#Importing the accuracy metric from sklearn.metrics library

from sklearn.metrics import accuracy_score
print('Accuracy Score on train data: ', accuracy_score(y_true=y_train, y_pred=clf.predict(X_train))*100)
print('Accuracy Score on test data: ', accuracy_score(y_true=y_test, y_pred=y_pred)*100)

from sklearn.metrics import classification_report, confusion_matrix
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))

```

Input:

Classes to predict: ['setosa' 'versicolor' 'virginica']
Classes to predict: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
Number of examples in the data: (150, 4)
[[5.1 3.5 1.4 0.2]
[4.9 3. 1.4 0.2]
[4.7 3.2 1.3 0.2]
[4.6 3.1 1.5 0.2]]

Output:

Accuracy Score on train data: 100.0
Accuracy Score on test data: 96.66666666666667
[[13 0 0]
[0 9 1]
[0 0 7]]

	precision	recall	f1-score	support
0	1.00	1.00	1.00	13
1	1.00	0.90	0.95	10
2	0.88	1.00	0.93	7

accuracy			0.97	30
macro avg	0.96	0.97	0.96	30
weighted avg	0.97	0.97	0.97	30

Assignment 3

Problem Statement:

Implement Logistic regression for loan datasets and evaluate the algorithm with precision, recall sensitivity and F1-score.

Concept to be applied:

Logistic Regression is a classification algorithm. It is used to predict a binary outcome (1 / 0, Yes / No, True / False) given a set of independent variables. To represent binary / categorical outcome, we use dummy variables.

Let's understand it further using an example:

We are provided a sample of 1000 customers. We need to predict the probability whether a customer will buy (y) a **particular magazine or not**. As you can see, we've a categorical outcome variable, we'll use logistic regression.

To start with logistic regression, I'll first write the simple linear regression equation with dependent variable enclosed in a link function:

$$g(y) = \beta_0 + \beta(\text{Age}) \quad \text{---- (a)}$$

Note: For ease of understanding, I've considered 'Age' as independent variable.

In logistic regression, we are only concerned about the **probability of outcome dependent variable** (success or failure). As described above, $g()$ is the link function. This function is established using two things: Probability of Success(p) and Probability of Failure($1-p$). p should meet following criteria:

1. It must always be positive (since $p \geq 0$)
2. It must always be less than equals to 1 (since $p \leq 1$)

Now, we'll simply satisfy these 2 conditions and get to the core of logistic regression. To establish link function, we'll denote $g()$ with ' p ' initially and eventually end up deriving this function.

Since **probability must always be positive**, we'll put the linear equation in exponential form. For any value of slope and dependent variable, exponent of this equation **will never be negative**.

$$p = \exp(\beta_0 + \beta(\text{Age})) = e^{(\beta_0 + \beta(\text{Age}))} \text{ ----- (b)}$$

To make the probability less than 1, we must divide p by a number greater than p. This can simply be done by:

$$p = \exp(\beta_0 + \beta(\text{Age})) / \exp(\beta_0 + \beta(\text{Age})) + 1 = e^{(\beta_0 + \beta(\text{Age}))} / e^{(\beta_0 + \beta(\text{Age}))} + 1 \text{ --- (c)}$$

Using (a), (b) and (c), we can redefine the probability as:

$$p = e^y / 1 + e^y \text{ --- (d)}$$

where p is the probability of success. *This (d) is the Logit Function*

If p is the probability of success, 1-p will be the probability of failure which can be written as:

$$q = 1 - p = 1 - (e^y / 1 + e^y) \text{ --- (e)}$$

where q is the probability of failure

On dividing, (d) / (e), we get,

$$\frac{p}{1 - p} = e^y$$

After taking log on both side, we get,

$$\log \left(\frac{p}{1-p} \right) = y$$

log(p/1-p) is the link function. Logarithmic transformation on the outcome variable allows us to model a non-linear association in a linear way.

After substituting value of y, we'll get:

$$\log \left(\frac{p}{1-p} \right) = \beta_o + \beta(\text{Age})$$

This is the equation used in Logistic Regression. Here **(p/1-p) is the odd ratio**. Whenever the log of odd ratio is found **to be positive**, the probability of success is always **more than 50%**. A typical logistic model plot is shown below. You can see probability never goes below 0 and above 1.

Procedure/Steps:

Step 1. Import required libraries

Step 2. Load the data, visualize and explore it

Step 3. Clean the data

STEP 4. Deal with any outliers

STEP 5. Split the data into a training set and testing set

STEP 6. Fit a logistic regression model using sklearn

STEP 7. Apply the model on the test data and make a prediction

STEP 8. Evaluate the model accuracy using the confusion matrix

STEP 9. Create the model and obtain the regression coefficients using statsmodel

STEP 10. The essential thing is, Interpret the regression coefficient in terms of the odds

Code:

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

df = pd.read_csv("/content/logit.csv")
df.info()
df.head(5)
plt.scatter(df.age, df.charges, color='red', marker='+')
plt.xlabel("Age of person")
plt.ylabel("Bought Insurance 1=Bought 0=Did not Buy")
x = df.iloc[:, 0:1].values
#x=df.drop("charges", axis=1)
y = df.iloc[:, -1].values

#Split the dataset into train and test sets (70:30)
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3, random_state=42)
print(x_test)

reg = LogisticRegression()
reg.fit(x_train, y_train)
yPrediction = reg.predict(x_test) #Predict the test set
print()
print(y_test)

plt.scatter(x_test, y_test, color='green', marker='*')
plt.scatter(x_test, yPrediction, color='blue', marker='.')

ins_accuracy = accuracy_score(y_test, yPrediction)
print('insurance score:', ins_accuracy * 100)

print(confusion_matrix(y_test, yPrediction))
print(classification_report(y_test, yPrediction))
o = reg.predict_proba(x_test)
print(o)

yPrediction1 = reg.predict([[96.9]])
print()
print(yPrediction1)
```

Input:

	A	B
1	age	charges
2	18	0
3	28	0
4	33	1
5	32	0
6	31	0
7	46	1
8	37	1
9	37	1
10	60	1

Output:

insurance score: 100.0

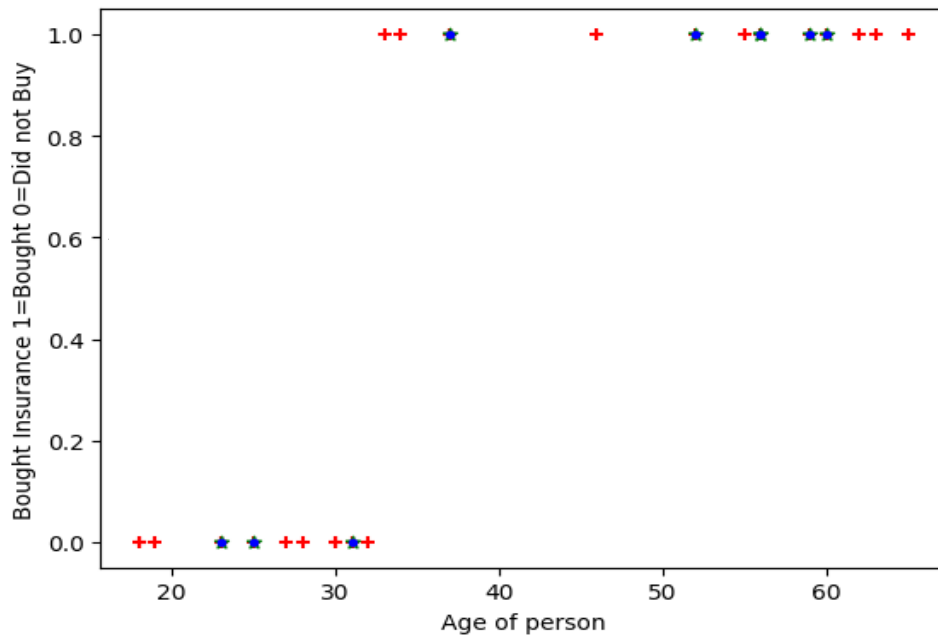
[[3 0]

[0 6]]

	precision	recall	f1-score	support
0	1.00	1.00	1.00	3
1	1.00	1.00	1.00	6
accuracy			1.00	9
macro avg	1.00	1.00	1.00	9
weighted avg	1.00	1.00	1.00	9

[[9.99983660e-01 1.63396068e-05]
[2.18511875e-10 1.00000000e+00]
[6.44755195e-03 9.93552448e-01]
[2.22200036e-12 1.00000000e+00]
[2.26485497e-14 1.00000000e+00]
[9.99837990e-01 1.62010150e-04]
[8.63545057e-01 1.36454943e-01]
[7.10542736e-14 1.00000000e+00]
[2.22200036e-12 1.00000000e+00]]

[1]



Assignment 4

Problem Statement:

Develop and fine-tune a Multi-Layer Perceptron (MLP) neural network model with back-propagation using scikit-learn, perform hyperparameter tuning, and assess its performance using various performance measures for a specific machine learning task.

Concept to be applied

1. **Data Preparation:** Begin by acquiring and preprocessing a dataset suitable for your machine learning task. Ensure proper data splitting into training, validation, and test sets. Perform data preprocessing, such as scaling or encoding categorical features, if necessary.
2. **MLP Model Design:** Design the architecture of your MLP neural network using scikit-learn's `MLPClassifier` or `MLPRegressor`. Specify the number of hidden layers, the number of neurons in each layer, activation functions, and output layer configuration. Ensure that the architecture is appropriate for the chosen task (e.g., classification or regression).
3. **Back-Propagation Implementation:** Utilize scikit-learn's MLP model to perform back-propagation and train your model. Configure parameters such as the solver (e.g., 'adam' or 'sgd'), learning rate, and batch size.
4. **Hyperparameter Tuning:** Optimize the model's performance through hyperparameter tuning using scikit-learn's `GridSearchCV` or `RandomizedSearchCV`. Tune hyperparameters such as learning rate, batch size, the number of hidden layers, the number of neurons per layer, regularization techniques (e.g., alpha for L2 regularization), and activation functions.
5. **Performance Measures:** Assess the performance of your tuned MLP model using performance measures available in scikit-learn. Metrics for classification tasks include accuracy, precision, recall, F1-score, ROC AUC, and confusion matrices. For regression tasks, use metrics such as mean squared error (MSE), mean absolute error (MAE), and R-squared (R2). Compute these metrics on the validation and test datasets to evaluate model performance comprehensively.

6. **Training and Validation:** Train your tuned MLP model using the training data, and monitor its training process for convergence. Validate the model's performance using the validation dataset and make necessary adjustments.
7. **Testing and Final Evaluation:** Evaluate your tuned MLP model on the test dataset to assess its generalization ability. Compare the results obtained on the test set with those on the validation set to ensure consistency.

Code:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
# Load the dataset
data = pd.read_csv("Heart Attack.csv")
# Encode the 'class' column (target) to numerical values
label_encoder = LabelEncoder()
data['target'] = label_encoder.fit_transform(data['target'])
# Split the data into features (X) and target (y)
X = data.drop(columns=['target'])
y = data['target']
# Split the data into training (70%), validation (15%), and test (15%) sets
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)
# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_val = scaler.transform(X_val)
X_test = scaler.transform(X_test)
from sklearn.neural_network import MLPClassifier
# Create an MLP Classifier model
mlp = MLPClassifier(
    hidden_layer_sizes=(64, 32), # Specify the number of neurons in hidden layers
    activation='relu', # Activation function for hidden layers
    solver='adam', # Optimization solver
    alpha=0.0001, # L2 regularization parameter
    max_iter=1000, # Maximum number of iterations
    random_state=42
)
```



```

mlp.fit(X_train, y_train)

from sklearn.model_selection import GridSearchCV

# Define hyperparameters to search
param_grid = {
    'hidden_layer_sizes': [(64, 32), (128, 64), (32, 16)],
    'alpha': [0.0001, 0.001, 0.01],
}

# Create a GridSearchCV object
grid_search = GridSearchCV(estimator=mlp, param_grid=param_grid, cv=5, scoring='accuracy',
n_jobs=-1)

# Fit the GridSearchCV to the data
grid_search.fit(X_train, y_train)

# Get the best hyperparameters
best_params = grid_search.best_params_
best_mlp = grid_search.best_estimator_

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score,
roc_auc_score, confusion_matrix

# Predictions on the validation set
y_val_pred = best_mlp.predict(X_val)

# Calculate performance metrics
accuracy = accuracy_score(y_val, y_val_pred)
precision = precision_score(y_val, y_val_pred)
recall = recall_score(y_val, y_val_pred)
f1 = f1_score(y_val, y_val_pred)
roc_auc = roc_auc_score(y_val, y_val_pred)
conf_matrix = confusion_matrix(y_val, y_val_pred)

# Print the performance metrics
print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1-Score: {f1}")
print(f"ROC AUC: {roc_auc}")
print(f"Confusion Matrix:\n{conf_matrix}")

import numpy as np

# Combine training and validation data for final training
X_train_final = np.vstack((X_train, X_val))
y_train_final = np.concatenate((y_train, y_val))

# Train the final model
final_mlp = MLPClassifier(

```

```

hidden_layer_sizes=best_params['hidden_layer_sizes'],
activation='relu',
solver='adam',
alpha=best_params['alpha'],
max_iter=1000,
random_state=42
)
final_mlp.fit(X_train_final, y_train_final)
# Predictions on the test set
y_test_pred = final_mlp.predict(X_test)
# Calculate performance metrics on the test set
test_accuracy = accuracy_score(y_test, y_test_pred)
test_precision = precision_score(y_test, y_test_pred)
test_recall = recall_score(y_test, y_test_pred)
test_f1 = f1_score(y_test, y_test_pred)
test_roc_auc = roc_auc_score(y_test, y_test_pred)
test_conf_matrix = confusion_matrix(y_test, y_test_pred)
# Print the final evaluation metrics on the test set
print("Final Evaluation on Test Set:")
print(f"Accuracy: {test_accuracy}")
print(f"Precision: {test_precision}")
print(f"Recall: {test_recall}")
print(f"F1-Score: {test_f1}")
print(f"ROC AUC: {test_roc_auc}")
print(f"Confusion Matrix:\n{test_conf_matrix}")

```

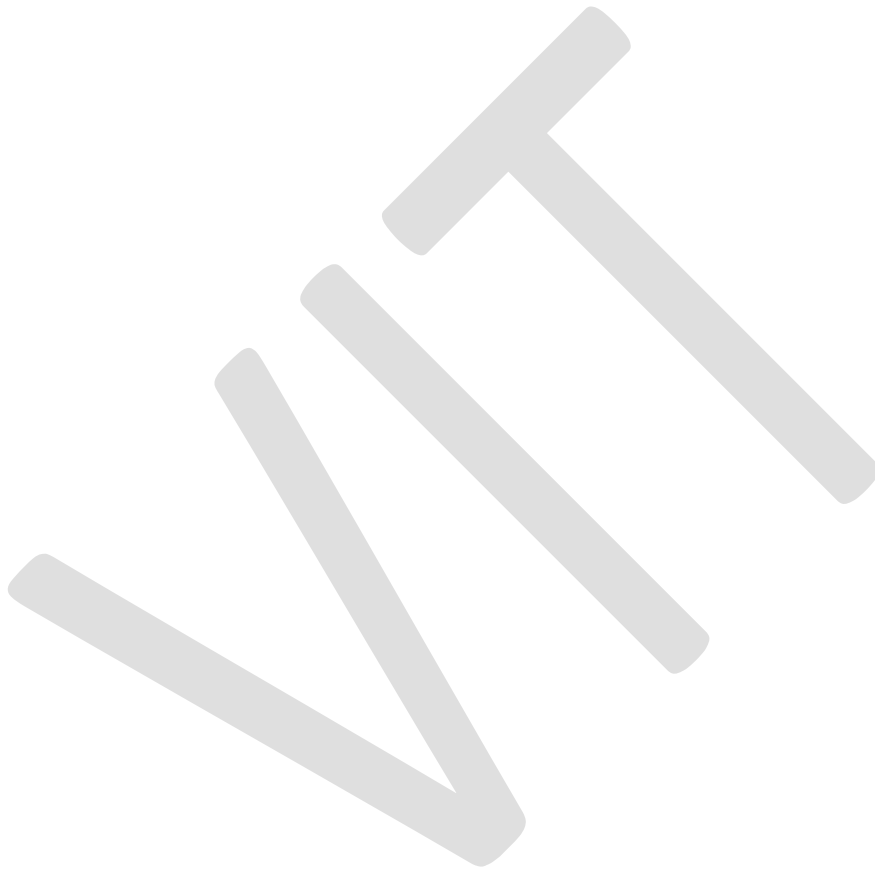
Output:

```

Accuracy: 0.987012987012987
Precision: 1.0
Recall: 0.967741935483871
F1-Score: 0.9836065573770492
ROC AUC: 0.9838709677419355
Confusion Matrix:
[[92  0]
 [ 2 60]]
Final Evaluation on Test Set:
Accuracy: 1.0
Precision: 1.0

```

Recall: 1.0
F1-Score: 1.0
ROC AUC: 1.0
Confusion Matrix:
[[67 0]
[0 87]]



Assignment 5

Problem Statement:

Let us Consider data set banknote authentication have four baseline variables, variance, skewness, curtosis, and entropy were obtained for each of $n = 1372$ bank notes, as well as the response of interest, a qualitative measure of category after baseline. Build Support Vector Machine model and computer the error using various error measures.

- Figure out if any preprocessing such as scaling would help here.
- Tune your model to reach with highest accuracy score.
- Find the optimal samples as training and test data size.

Concept to be applied:

Support Vector Machine or SVM is a supervised and linear Machine Learning algorithm most commonly used for solving classification problems and is also referred to as Support Vector Classification.

The objective of SVM is to draw a line that best separates the two classes of data points.

SVM generates a line that can cleanly separate the two classes. How clean, you may ask. There are many possible ways of drawing a line that separates the two classes, however, in SVM, it is determined by the margins and the support vectors.

The margin is the area separating the two dotted green lines as shown in the image above. The more the margin the better the classes are separated. The support vectors are the data points through which each of the green lines passes through. These points are called support vectors as they contribute to the margins and hence the classifier itself. These support vectors are simply the data points lying closest to the border of either of the classes which has a probability of being in either one.

The SVM then generates a hyperplane which has the maximum margin, in this case the black bold line that separates the two classes which is at an optimum distance between both the classes.

In case of more than 2 features and multiple dimensions, the line is replaced by a hyperplane that separates multidimensional spaces.

Procedure/Steps

Step 1: Import the Libraries-

Step 2: Load the Dataset

Step 3: Split Dataset into X and Y

Step 4: Split the X and Y Dataset into the Training set and Test set

Step 5: Perform Feature Scaling

Step 6: Fit SVM to the Training set

Step 7: Predict the Test Set Results

Step 8: Make the Confusion Matrix

Step 9. Visualise the Test set results

Code:

```
import pandas as pd
bankdata = pd.read_csv("/content/bill_authentication.csv")
print(bankdata.head(5))
X = bankdata.drop('Class', axis=1)
y = bankdata['Class']

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.20, random_state=42)

from sklearn.svm import SVC
svclassifier = SVC(kernel='linear')
svclassifier.fit(X_train, y_train)

y_pred = svclassifier.predict(X_test)

from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
print(accuracy_score(y_test,y_pred))

print(confusion_matrix(y_test,y_pred))
print(classification_report(y_test,y_pred))
```

#for polynomial kernel trick

```
from sklearn.svm import SVC
svclassifier = SVC(kernel='poly')
svclassifier.fit(X_train, y_train)

y_pred = svclassifier.predict(X_test)

from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
print(accuracy_score(y_test,y_pred))

print(confusion_matrix(y_test,y_pred))
print(classification_report(y_test,y_pred))
```

Input:

	Variance	Skewness	Curtosis	Entropy	Class
0	3.62160	8.6661	-2.8073	-0.44699	0
1	4.54590	8.1674	-2.4586	-1.46210	0
2	3.86600	-2.6383	1.9242	0.10645	0
3	3.45660	9.5228	-4.0112	-3.59440	0
4	0.32924	-4.4552	4.5718	-0.98880	0

Output:

0.9854545454545455

[[146 2]

[2 125]]

	precision	recall	f1-score	support
--	-----------	--------	----------	---------

0	0.99	0.99	0.99	148
---	------	------	------	-----

1	0.98	0.98	0.98	127
---	------	------	------	-----

accuracy			0.99	275
----------	--	--	------	-----

macro avg	0.99	0.99	0.99	275
-----------	------	------	------	-----

weighted avg	0.99	0.99	0.99	275
--------------	------	------	------	-----

0.9745454545454545

[[141 7]

[0 127]]

	precision	recall	f1-score	support
--	-----------	--------	----------	---------

0	1.00	0.95	0.98	148
---	------	------	------	-----

1	0.95	1.00	0.97	127
---	------	------	------	-----

accuracy			0.97	275
----------	--	--	------	-----

macro avg	0.97	0.98	0.97	275
-----------	------	------	------	-----

weighted avg	0.98	0.97	0.97	275
--------------	------	------	------	-----

Assignment 6

Problem Statement:

Implement K Nearest Neighbors algorithm for iris dataset and evaluate the algorithm with accuracy, precision, recall and F1-score.

Concept to be applied:

K-nearest neighbors (KNN) is a type of supervised learning algorithm used for both regression and classification. KNN tries to predict the correct class for the test data by calculating the distance between the test data and all the training points. Then select the K number of points which is close to the test data. The KNN algorithm calculates the probability of the test data belonging to the classes of 'K' training data and class holds the highest probability will be selected. In the case of regression, the value is the mean of the 'K' selected training points.

Procedure/Steps:

1. Load the data.
2. Initialise the value of k.
3. For getting the predicted class, iterate from 1 to total number of training data points.
 1. Calculate the distance between test data and each row of training dataset. Here we will use Euclidean distance as our distance metric since it's the most popular method. The other distance function or metrics that can be used are Manhattan distance, Minkowski distance, Chebyshev, cosine, etc. If there are categorical variables, hamming distance can be used.
 2. Sort the calculated distances in ascending order based on distance values.
 3. Get top k rows from the sorted array.
 4. Get the most frequent class of these rows.
 5. Return the predicted class.

Code:

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report, confusion_matrix
```

```
#To import the dataset and load it into our pandas dataframe, execute the following code:
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
# Assign column names to the dataset
names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'Class']
```

```
# Read dataset to pandas dataframe
dataset = pd.read_csv(url, names=names)
```

```
#from sklearn.datasets import load_iris
#dataset = load_iris()
```

To see what the dataset actually looks like, execute the following command:

```
dataset.head()
```

The next step is to split our dataset into its attributes and labels. To do so, use the following code:

```
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, -1].values
```

To create training and test splits, execute the following script:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20)
```

The following script performs feature scaling:

```
scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)
X_test = scaler.fit_transform(X_test)
```

Training and Predictions:

```
classifier = KNeighborsClassifier(n_neighbors=5)
classifier.fit(X_train, y_train)

y_pred = classifier.predict(X_test)
```

Evaluating the Algorithm:

```
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
y_pred = classifier.predict([[4,3.5,3,6.5]])
```

Input:

iris dataset

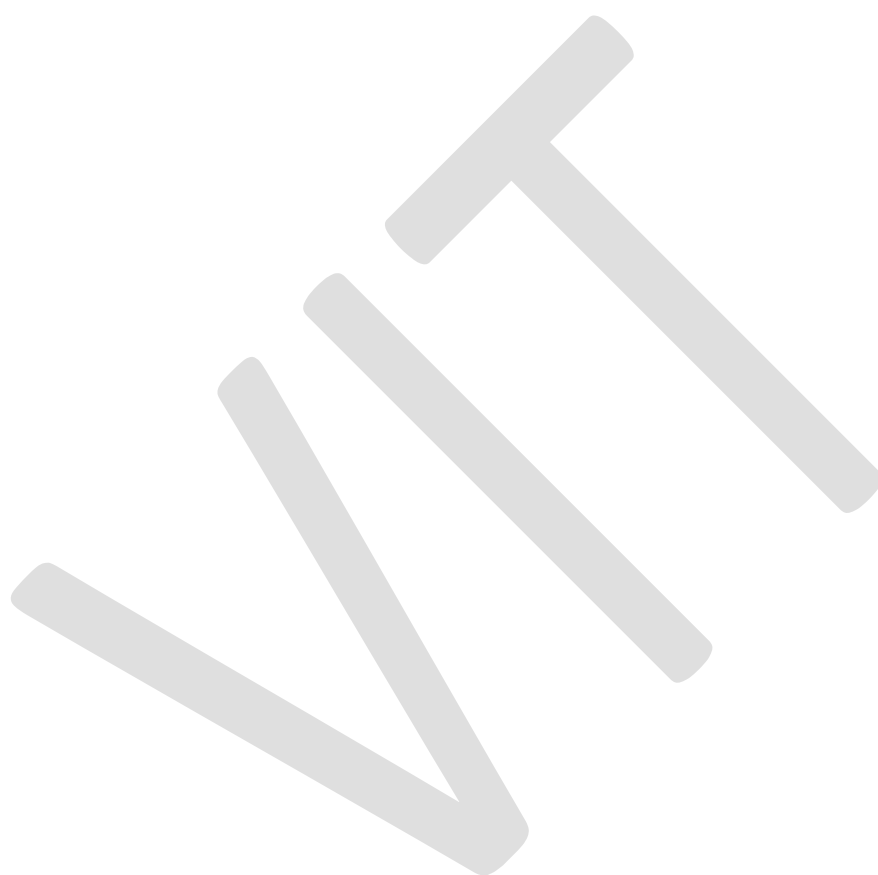
Output:

Classification Report:

	precision	recall	f1-score	support
Iris-setosa	1.00	1.00	1.00	20
Iris-versicolor	0.91	1.00	0.95	21
Iris-virginica	1.00	0.89	0.94	19
accuracy		0.97		60
macro avg	0.97	0.96	0.97	60

weighted avg 0.97 0.97 0.97 60

Accuracy: 0.9666666666666667



Assignment 7

Problem Statement:

Implement Random Forest algorithm for iris dataset and evaluate the algorithm with accuracy, precision, recall and F1-score.

Concept to be applied:

A Random Forest Algorithm is a supervised machine learning algorithm that is extremely popular and is used for Classification and Regression problems in Machine Learning. We know that a forest comprises numerous trees, and the more trees more it will be robust. Similarly, the greater the number of trees in a Random Forest Algorithm, the higher its accuracy and problem-solving ability. Random Forest is a classifier that contains several decision trees on various subsets of the given dataset and takes the average to improve the predictive accuracy of that dataset. It is based on the concept of ensemble learning which is a process of combining multiple classifiers to solve a complex problem and improve the performance of the model.

The following are the basic steps involved when executing the random forest algorithm:

1. Pick a number of random records, it can be any number, such as 4, 20, 76, 150, or even 2.000 from the dataset (called N records). The number will depend on the width of the dataset, the wider, the larger N can be. This is where the random part in the algorithm's name comes from!
2. Build a decision tree based on those N random records;
3. According to the number of trees defined for the algorithm, or the number of trees in the forest, repeat steps 1 and 2. This generates more trees from sets of random data records;
4. After step 3, comes the final step, which is predicting the results:
 - a. In case of classification: each tree in the forest will predict the category to which the new record belongs. After that, the new record is assigned to the category that wins the majority vote.
 - b. In case of regression: each tree in the forest predicts a value for the new record, and the final prediction value will be calculated by taking an average of all the values predicted by all the trees in the forest.

Each tree fit on a random subset of features will necessarily have no knowledge of some other features, which is rectified by ensembling, while keeping the computational cost lower.

Procedure/Steps

1. Import Required Libraries
2. Load and Prepare Data
3. Read the Data
4. Split the Data
5. Initialize and Train the Random Forest Classifier

6. Fit the Model
7. Make Predictions
8. Evaluate the Model
9. Fine-Tune Hyperparameters (Optional)
10. Final Model Training (Optional)

Code:

```
# importing required libraries
# importing Scikit-learn library and datasets package
from sklearn import datasets

# Loading the iris plants dataset (classification)
iris = datasets.load_iris()
# dividing the datasets into two parts i.e. training datasets and test datasets
X, y = datasets.load_iris( return_X_y = True)

# Splitting arrays or matrices into random train and test subsets
from sklearn.model_selection import train_test_split
# i.e. 70 % training dataset and 30 % test datasets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.30)
# importing random forest classifier from assemble module
from sklearn.ensemble import RandomForestClassifier
import pandas as pd
# creating dataframe of IRIS dataset
data = pd.DataFrame({'sepalength': iris.data[:, 0], 'sepalwidth': iris.data[:, 1],
                    'petallength': iris.data[:, 2], 'petalwidth': iris.data[:, 3],
                    'species': iris.target})

# creating a RF classifier
clf = RandomForestClassifier(n_estimators = 100)

# Training the model on the training dataset
# fit function is used to train the model using the training sets as parameters
clf.fit(X_train, y_train)

# performing predictions on the test dataset
y_pred = clf.predict(X_test)

# metrics are used to find accuracy or error
from sklearn import metrics
print()

# using metrics module for accuracy calculation
print("ACCURACY OF THE MODEL: ", metrics.accuracy_score(y_test, y_pred))
# importing random forest classifier from assemble module
from sklearn.ensemble import RandomForestClassifier
# Create a Random forest Classifier
clf = RandomForestClassifier(n_estimators = 100)

# Train the model using the training sets
clf.fit(X_train, y_train)
# using the feature importance variable
import pandas as pd
```

```
feature_imp = pd.Series(clf.feature_importances_, index =
iris.feature_names).sort_values(ascending = False)
feature_imp
```

Input:

iris dataset

Output:

Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	16
1	0.86	1.00	0.92	12
2	1.00	0.88	0.94	17
accuracy			0.96	45
macro avg	0.95	0.96	0.95	45
weighted avg	0.96	0.96	0.96	45

Accuracy: 0.9555555555555556

Assignment 8

Problem Statement:

Implement Ada Boost algorithm for iris dataset and evaluate the algorithm with accuracy, precision, recall and F1-score.

Concept to be applied:

AdaBoost, short for Adaptive Boosting, is a machine learning algorithm. AdaBoost technique follows a decision tree model with a depth equal to one. AdaBoost is nothing but the forest of stumps rather than trees. AdaBoost works by putting more weight on difficult to classify instances and less on those already handled well. AdaBoost algorithm is developed to solve both classification and regression problem.

Idea behind AdaBoost:

- Stumps (one node and two leaves) are not great in making accurate classification, so it is nothing but a weak classifier/ weak learner. Combination of many weak classifier makes a strong classifier, and this is the principle behind the AdaBoost algorithm.
- Some stumps get more performance or classify better than others.
- Consecutive stump is made by taking the previous stumps mistakes into account.

Procedure/Steps:

Step 1: Import the libraries.

Step 2: Load the Dataset

Step 3: Split dataset

Step 4: Build the AdaBoost model.

Step 5: Predict the values.

Code:

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier
from sklearn import datasets
iris= datasets.load_iris()
X = iris.data
Y = iris.target

le=LabelEncoder()
y=le.fit_transform(Y)

Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, random_state=0)
```

```

gnb = GaussianNB()
rf = RandomForestClassifier(n_estimators=10)

base_methods=[rf, gnb, dtc]
for bm in base_methods:
    print("Method: ", bm)
    ada_model=AdaBoostClassifier(base_estimator=bm)
    ada_model=ada_model.fit(Xtrain,ytrain)
    ytest_pred=ada_model.predict(Xtest)
    print(ada_model.score(Xtest, ytest))
    print(confusion_matrix(ytest, ytest_pred))

```

```

Method: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
    max_depth=None, max_features='auto', max_leaf_nodes=None,
    min_impurity_decrease=0.0, min_impurity_split=None,
    min_samples_leaf=1, min_samples_split=2,
    min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=1,
    oob_score=False, random_state=None, verbose=0,
    warm_start=False)

```

Input:

iris dataset

Output:

Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	16
1	1.00	0.88	0.93	16
2	0.87	1.00	0.93	13
accuracy			0.96	45
macro avg	0.96	0.96	0.95	45
weighted avg	0.96	0.96	0.96	45

Accuracy: 0.9555555555555555

Assignment 9

Problem Statement:

Illustrate the K-means clustering to cluster the data points for at least five epochs properly.

- Use the elbow method to determine the optimal number of clusters.
- Visualize the clusters.
- Plot the centroids of each cluster.

Procedure/Steps:

1. Import the dataset into our workspace using pandas.
2. Define the set of independent attributes. Since it is dependent attribute.
3. Next, plot the graph of elbow method to find the optimal number of clusters.
4. Train the k-means clustering model with the optimal number of clusters as input.
5. Print the results of each input as predicted by our model, which is the cluster they belong to.
6. Finally, visualize the clusters and their centroids by plotting a scatter plot.

Code:

```
#Importing Libraries

import pandas as pd

import numpy as np
import matplotlib.pyplot as plt

#Importing the Datasets
dataset = pd.read_csv('shopping-data.csv')
X = dataset.iloc[:, 3:].values

#Elbow method to find the optimal number of clusters
from sklearn.cluster import KMeans
wcss = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters=i, init='k-means++', max_iter=300, n_init=10).fit(X)
    wcss.append(kmeans.inertia_)
plt.plot(range(1, 11), wcss)
plt.title('The Elbow Method')
plt.xlabel('Number of Clusters')
plt.ylabel('WCSS')
plt.show()

#Applying Kmeans to the dataset
kmeans = KMeans(n_clusters=5, init='k-means++', max_iter=300, n_init=10); y_kmeans = kmeans.fit_predict(X)

#Printing out the cluster each input belongs to
y_kmeans

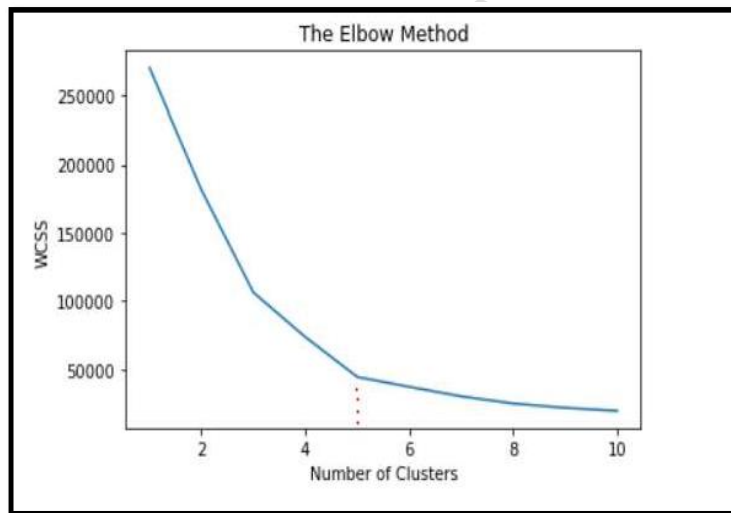
# Visualising the clusters
plt.scatter(X[y_kmeans == 0, 0], X[y_kmeans == 0, 1], s = 100, c = 'red', label = 'Standard Customers')
plt.scatter(X[y_kmeans == 1, 0], X[y_kmeans == 1, 1], s = 100, c = 'blue', label = 'Careless Customers')
plt.scatter(X[y_kmeans == 2, 0], X[y_kmeans == 2, 1], s = 100, c = 'cyan', label = 'Target Customers')
```

```

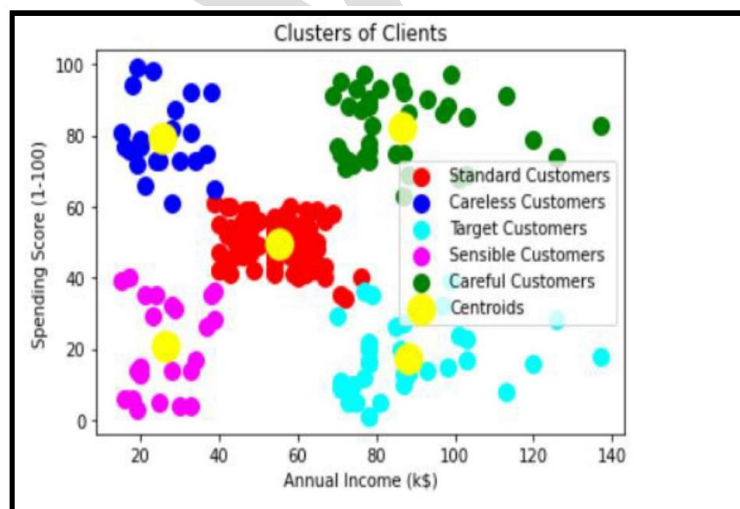
plt.scatter(X[y_kmeans == 3, 0], X[y_kmeans == 3, 1], s = 100, c = 'magenta', label = 'Sensible
Customers')
plt.scatter(X[y_kmeans == 4, 0], X[y_kmeans == 4, 1], s = 100, c = 'green', label = 'Careful
Customers')
plt.scatter(kmeans.cluster_centers_[0, 0], kmeans.cluster_centers_[0, 1], s = 300, c = 'yellow', label =
'Centroids')
plt.title('Clusters of Clients') plt.xlabel ('Annual Income (k$)') plt.ylabel ('Spending Score (1-100)')
plt.legend()
plt.show()

```

Output:



We can here see that graph for Within Cluster sum of squares (WCSS) and Number of clusters take a bend when number of clusters is 5. Hence, we assume that the optimal number of clusters are 5.



Here different clusters are marked as blue, red, pink, cyan and green. The yellow dot over each cluster represents its centroid. We can categorise these clusters as:

- Blue Cluster corresponds to careless customers as they have low income but high spending.
- Pink Cluster as Sensible customers, becoz they have low income and low spending.
- Red Clusters are standard cluster that suggest they have median income and median spending.
- The cyan coloured cluster correspond to Target Customers, as they have high income but low spending, the shopping company can give them offers and attractions as they are capable of spending more.
- Finally, the Green coloured clusters are Careful customers. They have high income and thus high spending as well.

Assignment 10

Problem Statement:

Let us consider data set mushroom have 22 baseline variables like cap-shape, cap-color, odor and so on were obtained for each of $n = 8124$ mushrooms, as well as the response of interest, a qualitative measure of category after baseline. Implement Principal Component Analysis for the given dataset.

Concepts to be applied:

Dimensionality Reduction is a statistical/ML-based technique wherein we try to reduce the number of features in our dataset and obtain a dataset with an optimal number of dimensions.

One of the most common ways to accomplish Dimensionality Reduction is Feature Extraction, wherein we reduce the number of dimensions by mapping a higher dimensional feature space to a lower-dimensional feature space. The most popular technique of Feature Extraction is Principal Component Analysis (PCA).

As stated earlier, Principal Component Analysis is a technique of feature extraction that maps a higher dimensional feature space to a lower-dimensional feature space. While reducing the number of dimensions, PCA ensures that maximum information of the original dataset is retained in the dataset with the reduced no. of dimensions and the co-relation between the newly obtained Principal Components is minimum. The new features obtained after applying PCA are called Principal Components and are denoted as **PC i** ($i=1,2,3...n$). Here, (Principal Component-1) PC1 captures the maximum information of the original dataset, followed by PC2, then PC3 and so on.

Procedure/Steps:

Step 1: Import necessary libraries

Step 2: Load the dataset

Step 3: Standardize the features

Step 4: Check the Co-relation between features without PCA (Optional)

Step 5: Applying Principal Component Analysis

Step 6: Checking Co-relation between features after PCA

Code:

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
m_data = pd.read_csv('/content/mushrooms.csv')

# Machine learning systems work with integers, we need to encode these
```

```

# string characters into ints

encoder = LabelEncoder()

# Now apply the transformation to all the columns:
for col in m_data.columns:
    m_data[col] = encoder.fit_transform(m_data[col])

X_features = m_data.iloc[:,1:23]
y_label = m_data.iloc[:, 0]
# Scale the features
scaler = StandardScaler()
X_features = scaler.fit_transform(X_features)
# Visualize
pca = PCA()
pca.fit_transform(X_features)
pca_variance = pca.explained_variance_

plt.figure(figsize=(6, 4))
plt.bar(range(22), pca_variance, alpha=0.5, align='center', label='individual variance')
plt.legend()
plt.ylabel('Variance ratio')
plt.xlabel('Principal components')
plt.show()
pca2 = PCA(n_components=17)
pca2.fit(X_features)
x_3d = pca2.transform(X_features)

plt.figure(figsize=(6,4))
plt.scatter(x_3d[:,0], x_3d[:,5], c=m_data['class'])
plt.show()
pca3 = PCA(n_components=2)
pca3.fit(X_features)
x_3d = pca3.transform(X_features)

plt.figure(figsize=(6,4))
plt.scatter(x_3d[:,0], x_3d[:,1], c=m_data['class'])
plt.show()

```

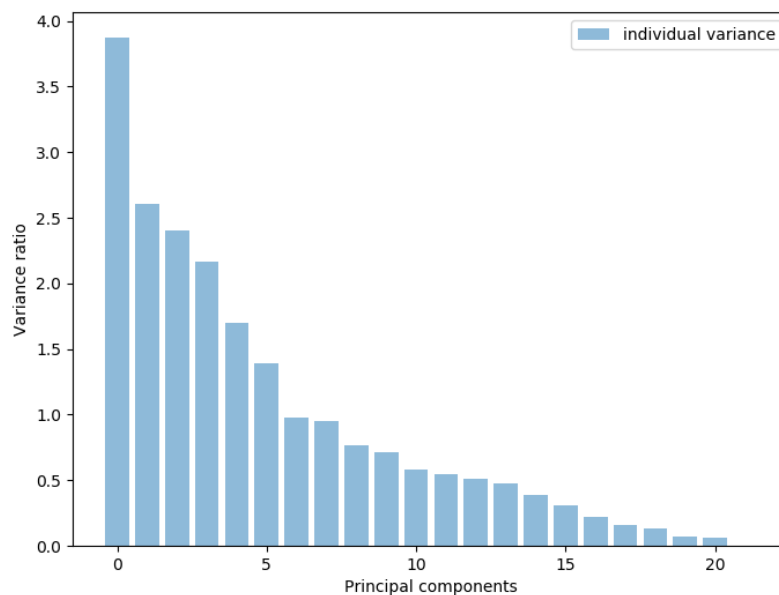
Input:

Data columns (total 23 columns):

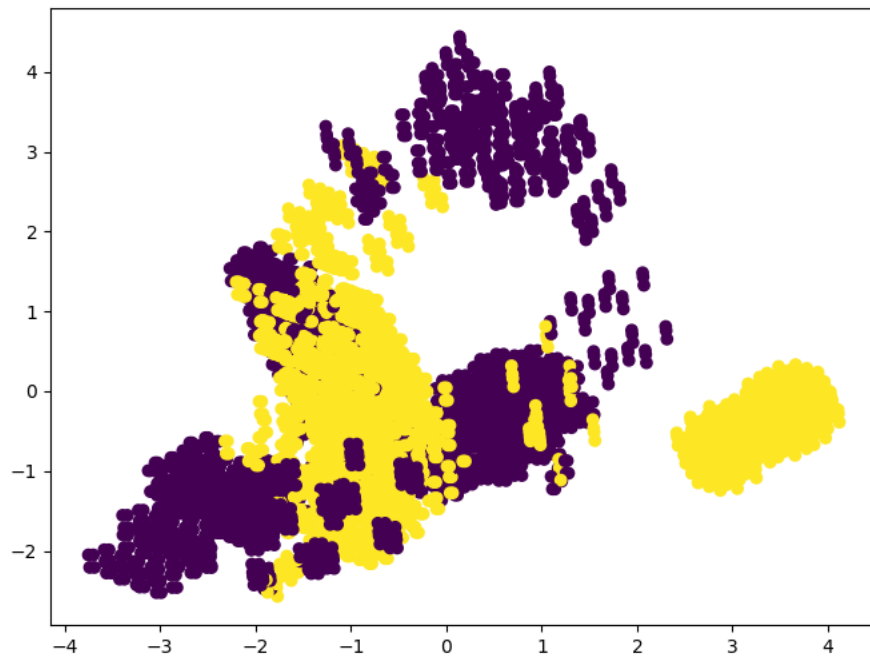
#	Column	Non-Null Count	Dtype
0	class	8124 non-null	object
1	cap-shape	8124 non-null	object
2	cap-surface	8124 non-null	object
3	cap-color	8124 non-null	object
4	bruises	8124 non-null	object
5	odor	8124 non-null	object

6	gill-attachment	8124 non-null	object
7	gill-spacing	8124 non-null	object
8	gill-size	8124 non-null	object
9	gill-color	8124 non-null	object
10	stalk-shape	8124 non-null	object
11	stalk-root	8124 non-null	object
12	stalk-surface-above-ring	8124 non-null	object
13	stalk-surface-below-ring	8124 non-null	object
14	stalk-color-above-ring	8124 non-null	object
15	stalk-color-below-ring	8124 non-null	object
16	veil-type	8124 non-null	object
17	veil-color	8124 non-null	object
18	ring-number	8124 non-null	object
19	ring-type	8124 non-null	object
20	spore-print-color	8124 non-null	object
21	population	8124 non-null	object
22	habitat	8124 non-null	object

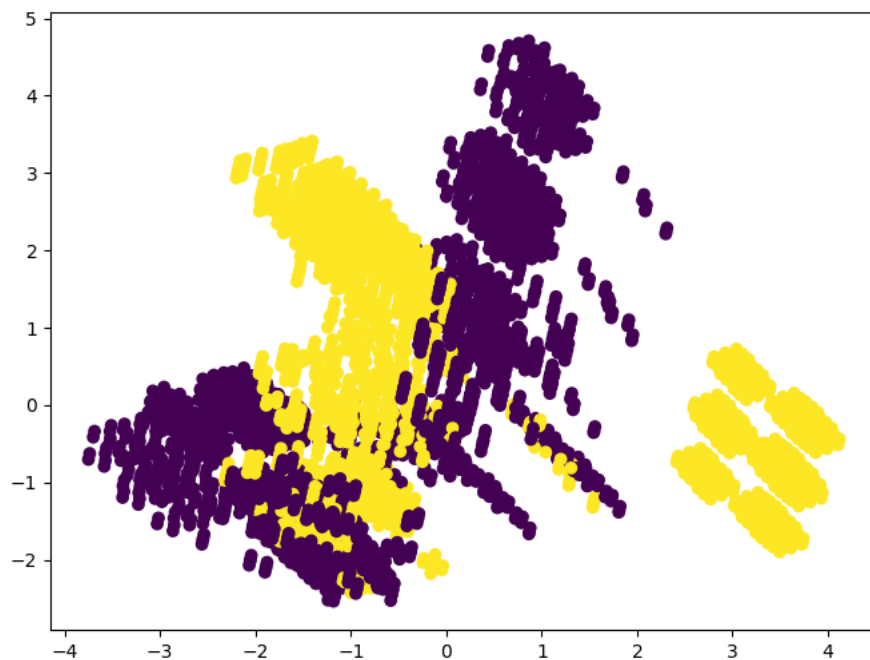
Output:



Use PCA to get the list of features and this plot depicts which features have the most explanatory power, or have the most variance. These are the principle components. It looks like around 17 or 18 of the features explain the majority, almost 95% of our data.



This plot a scatter plot of the data point classification based on these 17 features



This for the top 2 features and see how the classification changes