1- datu courses

Prior knowledge test (pre-assessing)

Here are 20 questions containing general concepts and code about machine learning.

Q1	Data Loading	question	C

Which option has a package that can not be used for data analysis.
A. scipy, numpy, matplotlib
B. numpy, matplotlib, pandas
C. matplotlib, pandas, pyro
D. scikit-learn, numpy, seaborn

Q2	Data preprocessing	question	C

When performing regression, which of the following ways is the correct way to preprocess data?
A. Normalize >> PCA >> handle outliers
B. PCA >> Normalize >> handle missing
C. handle outlier >> Normalize >> handle missing
D. None of the above

Q3	Data preprocessing	question	A

Which of the following snippet is not correct to handle missing data

A. df = pd.DataFrame(dict)
df.fillna()
B. data.replace(to_replace = np.nan, value = 0)
C. df = pd.DataFrame(dict)
df.dropna()
D. median = dataset[missing_feature].median()
dataset[missing_feature] = dataset[missing_feature].replace(to_replace=0, value=median)

Q4	Data preprocessing	question	C

Which of the following approaches does not need preprocessing?
A. Naive bayes
B. Logistic regression
C. Decision tree
D. None of the above

Q5	Feature Selection	question	B

Which of the following approaches is not true for selecting important features?
A. Information gain with decision tree or random forest
B. K-fold cross validation
C. p-value from linear regression
D. Correlation and Lasso

Q6	Feature Selection	**question**	**A**

What is the %68 rule in normal distribution?
A. 68 per cent of the data is around the mode
B. 68 per cent of the data is within the 3-sigma confidence interval.
C. The skewness is towards the mean
D. All the above

Q7	Classification	**question**	**D**

Which of the following models is suitable for classifying categorical output?
A. Logistic Regression
B. Decision tree, Random forest
C. KNN, Naive bayes
D. All the above

Q8	Classification	**question**	**D**

Which of the following methods are not suitable for binary classification?

A. Logistic Regression (LR)/ Linear Discriminant Analysis (LDA)

B. K-Nearest Neighbors (KNN)/ Classification and Regression Trees (CART).

C. Gaussian Naive Bayes (NB)/ Support Vector Machines (SVM).

D. Linear regression (LR) / Decision tree (DT)

Q9	Classification	**question**	**B**

Which one is wrong for hyper parameter tuning for the best model (model_best).

```
param_grid = {
    'C': [1.0, 10.0, 50.0],
   $1: ['linear', 'rbf', 'poly', 'sigmoid'],
    'shrinking': [True, False],
   $2: ['auto', 1, 0.1],
```

```
    'coef0': [0.0, 0.1, 0.5]
}
grid_search = GridSearchCV(model_best, param_grid, cv=10,$3='accuracy')
grid_search.fit($4, train_set_labels)
```

A. $1 = 'kernel'
B. $2 = 'alpha'
C. $3 = scoring
D. $4 = train_set_scaled

Q10	Classification	**question**	**D**

Which statement is true about SVM and logistic regression?
A. unlike logistic regression, SVM can form non linear decision surfaces and can be coupled with the kernel trick
B. Logistic regression is computationally cheaper
C. SVM project data in higher dimension and use kernel to apply nonlinear separation, while Logistic regression is nor linear separation
D. unlike logistic regression, classifier in SVM depends only on a subset of points

Q11	Regression	**question**	**D**

What is the cause of error in regression?
A. bias error
B. variance error
C. irreducible error
D. All of the above

Q12	Regression	**question**	**C**

Which of the following assumptions is not a necessary condition in linear regression?

A. Multivariate normality

B. Homoscedasticity

C. Multicollinearity

D. Linear relationship and no auto-correlation

Q13	Optimization	**question**	**A**

Which of the following approaches can not be used to avoid overfitting?
A. reducing the learning rate of the optimizer.
B. Adding a regularization term to the objective function (e.g. Lasso)
C. Reducing the variance of the model by making the model more simple.
D. K-Fold cross validation

Q14	Optimization	question	A

Which of the following statements is wrong?
A. SGD is faster compared to batch GD but it has higher variance.
B. Mini-batch GD is less likely to diverge compared to SGD in non-convex problems.
C. One of the concerns in deep networks is vanishing gradient.
D. One of the concerns in deep networks is exploding gradients.

Q15	Performance Metrics	question	B

Suppose you want to create a confusion matrix for actual and predicted values. What is the correct python code?
A. from sklearn import metrics
confusion_matrix = metrics.confusion_matrix(predicted, actual)
B. from sklearn import metrics
confusion_matrix = metrics.confusion_matrix(actual, predicted)
C. from sklearn import metrics
confusion_matrix = confusion_matrix.metrics(actual, predicted)
D. from sklearn import metrics
confusion_matrix = confusion_matrix.metrics(predicted, actual)

Q16	Performance Metrics	question	C

In the given snippet, which one is wrong to prediction on validation,
model = SVC($2)
model.$1(X_train, Y_train)
predictions = model.$3($4)
A. $2 = gamma='auto'
B. $1 = fit
C. $4 = Y_validation
D. $3 = predict

Q17	Evaluation	question	D

Given y_true as the grand-truth and y_pred as the predicted, which of the following is the correct python script for AUC-ROC plot?
A.from sklearn.metrics import roc_auc_score
fpr, tpr, thresholds = roc_auc_score(y_pred,y_true)
B.from sklearn.metrics import accuracy_score
roc = accuracy_score(true_y,y_pred)

C.from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_pred, true_y)
D.from sklearn.metrics import roc_auc_score
auc = roc_auc_score(y_true, y_pred)

Q18	Evaluation	**question**	**D**

Which of the following methods is not a valid method for evaluating a classifier?
A. Akaike Information Criteria (AIC)
B. Receiver operating characteristics (ROC curve)
C. Confusion Matrix
D. None of the the above

Q19	Analysis	**question**	**C**

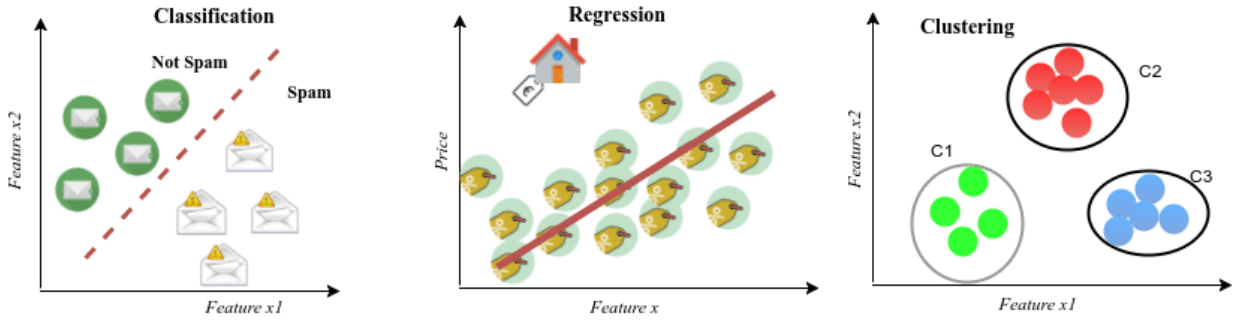
To deal with the class imbalance in a classification problem, which is wrong?

A. Using class weights
B. Using Sampling or SMOTE
C. Choosing non-likelihood-based models
D. Choosing Focal Loss

Q20	Analysis	**question**	**A**

Which one is a hyper parameter of the decision tree?
A. Min_samples
B. Min_depth
C. Max_leaves
D. Max_samples_leaf

Classification



Machine learning classification challenges demand the classification of a given data set into two or more categories.

There are several different sorts of classification problems:

1) Binary classification divides data into two categories: yes/no, good/bad, high/low, suffers from a specific ailment or not, and so on.
2) Classification on a multinomial scale: Organizes data into three or more categories; Document categorization, product categorization, and malware categorization are all examples of classification.

Classification issues are supervised learning problems in which the training data set includes data from both independent and response variables (label). Some of the following algorithms are used to train the classification models: Logistic regression, decision tree, Naive bayes, support vector machine, artificial neural network

Here are some real world example of classification problem:

Customer behaviour prediction: Customers can be divided into groups based on their purchasing habits, online shop browsing habits, and other factors.

Image classification: To categorize photos into distinct categories, a multinomial classification model can be developed.

Malware classification: A multinomial classification can be used to categorize new/emerging-malware based on similar malware traits.

Image sentiment analysis: Machine learning binary classification models based on machine learning algorithms can be created to classify whether an image has a good or negative emotion/sentiment.

Customer behaviour assessment for promotional offers: In the context of a given business scenario such as upselling, cross-selling, and so on, a binary classification model can be used to determine whether an account is customer-friendly or not.

Validation of deductions: A binary classification model can be used to determine if a deduction claimed by the buyer on a certain invoice is valid or invalid.

Course name: **classification**

Problem: **0 (Expert learning)**

title: **classifying Iris-data set**

Section 1	Problem description

We want to distinguish between three different species of iris — setosa, versicolor, and virginica, based on their measurements of their sepals and petals?

Iris Versicolor **Iris Setosa** **Iris Virginica**

Section 1	**Q1**	**solution**

Based on the question, this is a multi-class classification problem. We have three different classes which need to be grouped based on their features (sepals and petals length). The dataset contains 150 observations of iris flowers. There are four columns of measurements of the flowers in centimeters. The fifth column is the species of the flower observed. All observed flowers belong to one of three species. You can learn more about this dataset on Wikipedia.

Section 2	**Python requirement**

Section 2	Q1	question/answers

2-1- What are the required packages to be installed:

1) scipy, numpy, matplotlib, pandas, sklearn

2) numpy, matplotlib, pandas, seaborn

3) matplotlib, pandas, sklearn, numpy

4) scikit-learn, pandas, pandas, seaborn

Section 2	Q1	solution

Answer2-1 **:3,** We need to install packages to have access to the library.

We do not need the seaborn visualization, as we are using matplotlib for the same results. We also do not need the scipy package for this problem. You can check the version of the installed package using the following code.

```
# Python version
import sys
print('Python: {}'.format(sys.version))
# numpy
import numpy
print('numpy: {}'.format(numpy.__version__))
# matplotlib
import matplotlib
print('matplotlib: {}'.format(matplotlib.__version__))
# pandas
import pandas
print('pandas: {}'.format(pandas.__version__))
# scikit-learn
import sklearn
print('sklearn: {}'.format(sklearn.__version__))
```

Section 2	Q2	question/answers

2-2- which one is not necessary to import as requirement libraries

1)
```
from pandas import read_csv
from pandas.plotting import scatter_matrix
from matplotlib import pyplot
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
```
2)
```
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
```
3)
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
```
4)
```
from sklearn import preprocessing, metrics, cross_validation
from sklearn import svm
from sklearn.linear_model import LinearRegression
from sklearn import tree
```

Section 2	Q2	solution

anwer2-2) 4 The code is as follows:

```
# Load libraries
from pandas import read_csv
from pandas.plotting import scatter_matrix
from matplotlib import pyplot
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
```

```
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
```

Section 3	**Dataset Analysis**

Section 3	Q1	question/answers

3-1- load the dataset

```
url =
"https://raw.githubusercontent.com/jbrownlee/Datasets/master/iris.csv"
names = ['sepal-length', 'sepal-width', 'petal-length',
'petal-width', 'class']
1)
dataset = read_csv(url, names=names)
2)
dataset = read_file(url, names=names)
3)
dataset = read(url, names)
4)
dataset = read_file(url, title=names)
```

Section 3	Q1	solution

Answer 3-1) 1. The final code is as follows

```
# Load dataset
url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/iris.csv"
names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'class']
dataset = read_csv(url, names=names)
```

Section 3	Q2	question/answers

3-2- Which one is not true about the data?

1)150 samples, with 4 attributes (same units, all numeric)

2)Balanced class distribution (50 samples for each class)

3)The first 10 data are `Iris-setosa`

4)The mean of sepal width is 5.843

Section 3	Q2	solution

Answer 3-2) 4. To check for the first option, we can use this code

```
print(dataset.shape)
```

For the second option:

```
print(dataset.groupby('class').size())
```

For the third option:

```
 print(dataset.head(10))
```

For the fourth option

```
print(dataset.describe())
```

The mean of sepal width is 3.05400

Section 3	Q3	question/answers

3-3- Which of the following codes does not show some distribution of the data?

1)
```
scatter_matrix(dataset)
pyplot.show()
```
2)
```
dataset.hist()
pyplot.show()
```
3)
```
dataset.plot(kind='box', subplots=True, layout=(2,2), sharex=False,
sharey=False)
pyplot.show()
```
4)
```
pyplot.scatter(dataset)
pyplot.show()
```

Section 3	Q4	Solution

Answer3-3) 4. For pyplot scatter option, you need to pass x,y seperately. Feel free to check these results in our visualization toolbox.

Section 3	Q4	question/answers

3-4- which of the following is not required in this dataset:

1) Handling missing data
2) Separate validation/test/train
3) Set-up test harness
4) Build prediction models and accuracy measures

Section 3	Q4	Solution

Answer 3-4) 1. As seen by the dataset there is no missing data in the system. We can count the number of missing using this code:

dataset`.isna().sum()`

Section 3	Q5	question/answers

3-5- handle missing data

```
1) df = pd.DataFrame(dict)
df.fillna(0)
2) data.replace(to_replace = np.nan, value = 0)

3) df = pd.DataFrame(dict)
df.dropna()
4) The code does not require handling missing data.
```

Section 3	Q5	Solution

Answer 3-5) 4, Since there is no missing data, no option is required.

Section 4	**Model**

Section 4	Q1	question/answers

4-1) Fill the $ signs to create a validation dataset

```
array = dataset.values
X = array[:,$1]
y = array[:,$2]
X_train, X_validation, Y_train, Y_validation = train_test_split(X, y,$3,
random_state=1)
```

1) $1 = 0:4, $2 = 4,$3 = test_size=0.20
2) $1 = 0:3, $2 = 4,$3 = test_size=0.80

3) $1 = 0:3, $2 = 5,$3 = test_size=0.80
4) $1 = 0:4, $2 = 5,$3 = test_size=0.20

Section 4	Q1	solution

Answer 4-1) 1, the code is as follows:

```
# Split-out validation dataset
array = dataset.values
X = array[:,0:4]
y = array[:,4]
X_train, X_validation, Y_train, Y_validation = train_test_split(X, y, test_size=0.20,
random_state=1)
```

Section 4	Q2	question/answers

4-2- which of the following methods are not suitable as classification here?

1) Logistic Regression (LR)
   Linear Discriminant Analysis (LDA)
2) K-Nearest Neighbors (KNN).
   Classification and Regression Trees (CART).
3) Gaussian Naive Bayes (NB).
   Support Vector Machines (SVM).
4) Linear regression (LR)
   Decision tree (DT)

Section 4	Q2	Solution

Answer 4-6) 4, Linear regression is for regression not for classification problems.

Section 4	Q3	question/answers

4-3- create a logistic regression model and add it to the model list, which one is wrong

```
models = []
models.$1(($2, $3(solver='liblinear',$4)))
```

1) $1 = append
2) $2 = 'LR'
3) $3 = LogisticRegression
4) $4 = multi_class='lr'

Section 4	Q3	Solution

Answer 4-3) 4, the code is as follow

```
models.append(('LR', LogisticRegression(solver='liblinear', multi_class='ovr')))
```

Section 4	Q4	question/answers

4-4- create a linear discriminant model

```
models.$1(($2,$3($4)))
```

1) $1=append
2) $2= 'LDA'
3) $3= LinearDiscriminantAnalysis
4) $4 = dataset

Section 4	Q4	solution

Answer 4, no input needed

```
models.append(('LDA', LinearDiscriminantAnalysis()))
```

Section 4	Q5	question/answers

4-5- Create KNN model,which is wrong

```
models.$1(($2,$3($4)))
    5)  $1=append
    6)  $2= 'KNN'
    7)  $3= KNeighborsClassifier
    8)  $4 = dataset
```

Section 4	Q5	solution

Answer 4) there is no input needed

```
models.append(('KNN', KNeighborsClassifier()))
```

Section 4	Q6	question/answers

4-6- create decision tree model,which in wrong

```
models.$1(($2,$3($4)))
```

1)$1=`append`

2)$2 = `'CART'`

3)$3 = `DecisionTreeClassifier`

4)$4 = dataset

Section 4	Q6	solution

Answer 4, there is no input needed

```
models.append(('CART', DecisionTreeClassifier()))
```

Section 4	Q7	question/answers

4-7- create naive bayes model, which one wrong

```
models.$1(($2,$3($4)))
```

1)$1=`append`

2)$2 = `'NB'`

3)$3=`GaussianNB`

4)$4 = dataset

Section 4	Q7	solution

Answer 4) there is not input needed

```
models.append(('NB', GaussianNB()))
```

Section 4	Q8	question/answers

4-8- create support vection machine model, which is wrong

```
models.$1(($2, SVC($3=$4)))
```

1)$1 = `append`

2)$2 =`'SVM'`

3)$3 = `gamma`

4)$4 = `'min'`

Section 4	Q8	solution

Answer 4) the option is `'auto'`

```
models.append(('SVM', SVC(gamma='auto')))
```

Section 4	Q9	question/answers

4-9- create stratified k-fold, which is wrong

```
kfold =$4($1,$2,$3)

1)$1 =n_splits=10

2)$2 = random_state=1

3)$3 = shuffle=True

4)$4 = =KFold
```

Section 4	Q9	solution

```
Answer 4. The final code
```

```
kfold = StratifiedKFold(n_splits=10, random_state=1, shuffle=True)
```

Section 4	Q10	question/answers

4-10- compare evaluate model, complete the code what is wrong

```
results = []

names = []

for name, model in models:

       kfold = <from last question>

       cv_results = cross_val_score($1, X_train, Y_train, cv=kfold,$2)

       results.$4(cv_results)

       names.append(name)

       print('%s: %f (%f)' % (name, $3, cv_results.std()))
```

1) $1 = `model`
2) $2= scoring='accuracy'
3) $3 = cv_results.mean()
4) $4 = current

Section 4	Q10	solution

Answer 4-10) 4 (current should be append)

```
results = []
names = []
for name, model in models:
       kfold = StratifiedKFold(n_splits=10, random_state=1, shuffle=True)
       cv_results = cross_val_score(model, X_train, Y_train, cv=kfold,
scoring='accuracy')
       results.append(cv_results)
       names.append(name)
       print('%s: %f (%f)' % (name, cv_results.mean(), cv_results.std()))
```

Section 4	Q11	question/answers

4-11- which method has the highest accuracy?

1) SVM
2) CART
3) KNN
4) LDA

Section 4	Q11	solution

Answer 4-15) 1 accuracy is 98%

Section 5	**Prediction**

Section 5	Q1	question/answers

5-1- In the code bellow to do prediction on validation, which is wrong

model = SVC($2)

model.$1(X_train, Y_train)

predictions = model.$3($4)

1) $1 = fit
2) $2 = gamma='auto'
3) $3 = predict
4) $4 = Y_validation

Section 5	Q1	solution

Answer 5-1) 4, current is X_validation. The code is:

```
# Make predictions on validation dataset
model = SVC(gamma='auto')
model.fit(X_train, Y_train)
predictions = model.predict(X_validation)
```

Section 5	Q2	question/answers

5-2) compute accuracy on validation

1)0.99

2)0.98

3)0.91

4)0.96

Section 5	Q2	solution

5-2) 4

print(accuracy_score(Y_validation, predictions))

```
>0.9666666666666667
```

Section 5	Q3	question/answers

5-3) about computing number of misclassified in each category, which not true?

1) Misclassified class 1 =0
2) one class 2 was classified as 3
3) 6 class 3 were computed correclty
4) Two misclassified in total

Section 5	Q3	solution

Answer 5-3)4, only 1 misclassified
print(confusion_matrix(Y_validation, predictions))

```
[[11  0  0]
[ 0 12  1]
[ 0  0  6]]
```

Section 5	Q4	question/answers

4-4) which one about the precision, recall, f1, f2 score is wrong?

1) Precision `Iris-virginica` = `0.86`
2) `Recall Iris-versicolor = 0.92`
3) `F1-score  Iris-setosa =1`
4) `Weighted average f1-score = 0.95`

Section 5	Q4	answer

Answer 4-4) 4, the weight average f1 score is 0.97

print(classification_report(Y_validation, predictions))

```
>                 precision    recall  f1-score   support

    Iris-setosa       1.00      1.00      1.00        11
Iris-versicolor       1.00      0.92      0.96        13
Iris-virginica        0.86      1.00      0.92         6

       accuracy                           0.97        30
      macro avg       0.95      0.97      0.96        30
   weighted avg       0.97      0.97      0.97        30
```

Section 6	**Final code**

```
# Block 1: (section 2) Import
from pandas import read_csv
from matplotlib import pyplot
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import StratifiedKFold
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC

# Block 2: (section 3) Data analysis
url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/iris.csv"
names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'class']
dataset = read_csv(url, names=names)
# Split-out validation dataset
array = dataset.values
X = array[:,0:4]
y = array[:,4]
X_train, X_validation, Y_train, Y_validation = train_test_split(X, y, test_size=0.20,
random_state=1, shuffle=True)

# Block 3: (section 4) Modeling
models = []
models.append(('LR', LogisticRegression(solver='liblinear', multi_class='ovr')))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
models.append(('SVM', SVC(gamma='auto')))

# Block 4: (section 5) Prediciton
results = []
names = []
for name, model in models:
        kfold = StratifiedKFold(n_splits=10, random_state=1, shuffle=True)
```

```
        cv_results = cross_val_score(model, X_train, Y_train, cv=kfold, scoring='accuracy')
        results.append(cv_results)
        names.append(name)
        print('%s: %f (%f)' % (name, cv_results.mean(), cv_results.std()))
# Compare Algorithms
pyplot.boxplot(results, labels=names)
pyplot.title('Algorithm Comparison')
pyplot.show()
```

Course name: **classification**

Problem: **1 (Binary classification)**

title: **classifying** PIMA

The **Pima** are a group of **Native Americans** living in Arizona. A genetic predisposition allowed this group to survive normally to a diet poor of carbohydrates for years. In the recent years, because of a sudden shift from traditional agricultural crops to processed foods, together with a decline in physical activity, made them develop **the highest prevalence of type 2 diabetes** and for this reason they have been subject of many studies. The other dataset with the same property are:

- Haberman Breast Cancer (Haberman)
- German Credit (German)

Section 1	**Problem description**

The dataset includes data from **768** women with **8** characteristics, in particular:

1. Number of times pregnant
2. Plasma glucose concentration a 2 hours in an oral glucose tolerance test
3. Diastolic blood pressure (mm Hg)
4. Triceps skin fold thickness (mm)
5. 2-Hour serum insulin (mu U/ml)
6. Body mass index (weight in kg/(height in m)^2)
7. Diabetes pedigree function
8. Age (years)

The last column of the dataset indicates if the person has been diagnosed with diabetes (1) or not (0)

The original dataset is available at **UCI Machine Learning Repository** and can be downloaded from this address:
http://archive.ics.uci.edu/ml/datasets/Pima+Indians+Diabetes

Section 1	Q1	question

1-1- What is the machine learning task?

1) **Supervised binary classification**
2) **Supervised multi-label classification**
3) **Unsupervised binary clustering**
4) **Unsupervised multi-category grouping**

Section 1	Q1	solution

1) The type of dataset and problem is a classic **supervised binary classification**. Given a number of elements all with certain characteristics (features), we want to build a machine learning model to identify people affected by type 2 diabetes.

To solve the problem we will have to analyze the data, do any required transformation and normalization, apply a machine learning algorithm, train a model, check the performance of the trained model and iterate with other algorithms until we find the most performant for our type of dataset.

Section 1	Q1	Feedback

F1	Get more information about the different tasks in this link.
F2	Look back to the problem description. Is the prediction output discrete or continuous? How many output results are there?
F3	If the dataset has label/output, then the ML model is supervised. If the number of discrete outputs are more than two, that is a multi-label problem.

Section 2	Python requirements

Section 2	Q1	question

1-1) which import is not required

```
1)
from sklearn.cluster import KMeans # Our clustering algorithm
from sklearn.decomposition import PCA # Needed for dimension reduction
from sklearn.preprocessing import StandardScaler
2)
from pandas import read_csv
import seaborn as sns
import matplotlib.pyplot as plt
3)
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler as Scaler
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.svm import LinearSVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeRegressor
from sklearn import model_selection
from sklearn.model_selection import GridSearchCV

4)
import os
import pandas as pd
import numpy as np
from numpy import unique
```

Section 2	Q1	Solution

1-1) 1, we are not using `PCA, kmeans. The code is as follows`

```
import os
import pandas as pd
import numpy as np
from numpy import unique
```

```
from pandas import read_csv
import seaborn as sns
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler as Scaler
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.svm import LinearSVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeRegressor
from sklearn import model_selection
from sklearn.model_selection import GridSearchCV
```

Section 2	**Q1**	**Feedback**

F1	Answer this question after you went through all segments to know which packages are required.
F2	Based on the task in the previous question, select a choice which has packages that are not related.
F3	In the sklearn package, we need different models for our task. We need a package for data processing and visualization.

Section 3	**Data Analysis**

Section 3	**Q1**	**question**

3-1- load the dataset. Which one is not true:

```
# load the dataset
url =
'https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.csv
'
```

```
dataset =$1(url $2)
dataset.$3 = [
    "NumTimesPrg", "PlGlcConc", "BloodP",
    "SkinThick", "TwoHourSerIns", "BMI",
    "DiPedFunc", $4, "HasDiabetes"]
```

1) $1 = `pd.read_csv`

2) $2 = `, header=None`

3)$3 = `columns`

4)$4 = `"Race"`

Section 3	Q2	Solution

4) $4 = `"Age"`

```
# load the dataset
url =
'https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.csv
'
dataset =pd.read_csv(url , header=None)
dataset.columns = ["NumTimesPrg", "PlGlcConc", "BloodP",
    "SkinThick", "TwoHourSerIns", "BMI",
    "DiPedFunc", "Age", "HasDiabetes"]
```

Section 3	Q1	Feedback

F1	Please refer to this link to load dataset.
F2	Read_csv from panda package is used to load csv dataset. By default this will ignore the first line which is the header. Features names can be added as dataset columns.
F3	As the dataset does not have a header, we need to add the features name in order into the dataset columns.

Section 3	Q2	question

3-2- Which statement is not true about the dataset?

1) The number of data points are 768

2) There are 8 features

3) there are two classes

4) The dataset is balanced.

Section 3	**Q2**	**Solution**

4) The dataset is not balanced. Arround 65% of class 0, and 35% of class 1. The code to check is as follows

```
# get the values
values = dataset.values
X, y = values[:, :-1], values[:, -1]
# gather details
n_rows = X.shape[0]
n_cols = X.shape[1]
classes = unique(y)
n_classes = len(classes)
# summarize
print('N Examples: %d' % n_rows)
print('N Inputs: %d' % n_cols)
print('N Classes: %d' % n_classes)
print('Classes: %s' % classes)
print('Class Breakdown:')
# class breakdown
breakdown = ''
for c in classes:
	total = len(y[y == c])
	ratio = (total / float(len(y))) * 100
	print(' - Class %s: %d (%.5f%%)' % (str(c), total, ratio))
```

Section 3	**Q2**	**Feedback**

F1	Use the EDA toolbox to check this.

F2	The number of rows in a dataset are the total data points, and columns are the number of features. The number of unique outputs determines the class number.
F3	A balanced dataset is when the number of samples for different classes are roughly equal. In a panda dataset, one can use the following code to check `dataset.groupby(<output>).size()`

Section 3	**Q3**	**question**

3-3-Which statement is not true?

1) The greater the age of a patient is, the greater the probabilities that the patient can develop type 2 diabetes.

2) The greater the BMI of a patient is, the greater the probabilities that the patient can develop type 2 diabetes.

3) PlGlcConc level has the most effect on developing type 2 diabetes among other features.

4) TwoHourSerIns value has the least effect on developing type 2 diabetes among other features.

Section 3	**Q3**	**Solution**

3-3) 4. The least effect is SkinThick with a correlation value of 0.074752. To obtain these values, we need to compute correlation matrices.

The correlation matrix is an important tool to understand the correlation between the different characteristics. The values range from -1 to 1 and the closer a value is to 1 the better correlation there is between two characteristics. Let's calculate the correlation matrix for our dataset.

```
corr = dataset.corr()

# showing the visulization
```

```
sns.heatmap(corr, annot = True)
```

Section 3	Q3	Feedback

F1	Use the visualization toolbox, and check for grouping plots.
F2	The correlation matrix is an important tool to understand the correlation between two characteristics. The values range from -1 to 1 and the closer a value is to 1 the greater the two quantities relations.
F3	By using the correlation matrix with the following code, if the correlation between output and a feature is positive, increasing the feature will result in increasing the output. While the value closer to zero, means the feature is the least related. `corr = dataset.corr()`

Section 3	Q4	question

3-4- Which of the following features does not have missing values?

1) SkinThick

2) BloodP

3) BMI

4) Age

Section 3	Q4	Solution

3-4) 4, Age does not have null values. You can check by the histogram plot, or the visualization tool available in the website. These are the list of features with missing values:

```
dataset.hist(bins=50, figsize=(20, 15))

plt.show()

Features_missing = ['BMI', 'BloodP', 'PlGlcConc', 'SkinThick','TwoHourSerIns']
```

Section 3	Q4	Feedback

F1	Please use the EDA toolbox > histogram plot.
F2	Missing values are either empty entry, np.nan, or zero. The latest is used only if zero value does not provide meaningful information.
F3	If the dataset missing values are imputed with zeros, histogram plots can detect outliers. `dataset.hist()`

Section 3	Q5	question

3-5- handle missing data. Which one is correct?

```
1) Features_missing = <from previous question>
for feature in Features_missing:
      median = dataset[feature].median()
      dataset[feature] = dataset[feature].replace(to_replace=0, value=median)
```

```
2) Features_missing = <from previous question>
for feature in Features_missing:
      dataset[feature].replace(to_replace = 0, value = np.nan)
      mean = dataset[feature].mean()
      dataset[feature].fillna(mean, inplace = True)
```

```
3) dataset.replace(to_replace = 0, value = np.nan)
dataset.dropna()
```

```
4) No missing data. This step is optional
```

Section 3	Q5	Solution

3-5)1

We have noticed from the previous analysis that some patients have missing data for some of the features. Machine learning algorithms don't work very well when the data is missing so we have to find a solution to "clean" the data we have.

The easiest option could be to eliminate all those patients with null/zero values, but in this way we would eliminate a lot of important data.

Another option is to calculate the **median** value for a specific column and substitute that value everywhere (in the same column) we have zero or null. Let's see how to apply this second method.

```
Features_missing = ['BMI', 'BloodP', 'PlGlcConc','SkinThick','TwoHourSerIns']
for feature in Features_missing:
      median = dataset[feature].median()
      # Substitute it in the BMI column of the dataset where values are 0
      dataset[feature] = dataset[feature].replace(to_replace=0, value=median)
```

Section 3	Q5	Feedback

F1	Check this link to handle missing data.
F2	There are two approaches to handle missing data: remove/impute. If only one feature is missing, it is better not to remove the whole dataset.
F3	The highest frequency of a feature, or mean is not the most robust approach to impute a dataset, as the null values can affect them.

Section 4	Model

Section 4	Q1	question

4-1- Split the dataset to test/train and input/output. Which one is wrong:

```
train_set, test_set = train_test_split(dataset,$3)

# Separate labels from the rest of the dataset

train_set_labels = train_set["HasDiabetes"].$1

train_set = train_set.$2("HasDiabetes",$4)

test_set_labels = test_set["HasDiabetes"].$1

test_set = test_set.$2("HasDiabetes", $4)

1)$1 = copy()

2)$2 = drop

3)$3 = test_size=0.95, random_state=42

4)$4 = axis=1
```

Section 4	Q1	Solution

4-1) 3, around 20-30% of the data should go to the test dataset.

Now that we have transformed the data we need to split the dataset in two parts: a training dataset and a test dataset. Splitting the dataset is a very important step for supervised machine learning models. Basically we are going to use the first part to train the model (ignoring the column with the pre assigned label), then we use the trained model to make predictions on new data (which is the test dataset, not part of the training set) and compare the predicted value with the pre assigned label.

The full code is as follows.

```
train_set, test_set = train_test_split(dataset, test_size=0.2, random_state=42)

# Separate labels from the rest of the dataset

train_set_labels = train_set["HasDiabetes"].copy()
```

```
train_set = train_set.drop("HasDiabetes", axis=1)

test_set_labels = test_set["HasDiabetes"].copy()

test_set = test_set.drop("HasDiabetes", axis=1)
```

Section 4	Q1	Feedback

F1	To avoid overfitting, the dataset should be divided into train/test. More information in this link
F2	We need to separate the output from the features in two dataframes, labels and test/train_set. A portion of data should be allocated to evaluate performance of the model to unseen data.
F3	Train_set portion should be relatively higher compared to the test_set as the model is only exposed to the train_set while learning. We can provide seed for separation to have reproducible results.

Section 4	Q2	question

4-2- Feature scaling. Which one is incorrect

```
scaler = $1

scaler.$2($3)

train_set_scaled = scaler.$4(train_set)

test_set_scaled = scaler.$4(test_set)
```

1)$1 = `Scaler()`

2)$2 = `fit`

3)$3 =`test_set`

4)$4 = `transform`

Section 4	Q2	Solution

4-2) 3, it should be applied to the (`train_set`)

One of the most important data transformations we need to apply is the features scaling. Basically most of the machine learning algorithms don't work very well if the features have a different set of values. In our case for example the Age ranges from 20 to 80 years old, while the number of times a patient has been pregnant ranges from 0 to 17. For this reason we need to apply a proper transformation.

```
scaler = Scaler()

scaler.fit(train_set)

train_set_scaled = scaler.transform(train_set)

test_set_scaled = scaler.transform(test_set)
```

Section 4	Q2	Feedback

F1	Scaling all the features to a comparable range is a data wrangling step. More information this link.
F2	In preprocessing stage, data normalization/data whitening is essential as most of the machine learning algorithms don't work very well if the features have a different set of values. The linear transformation should apply the same for train/test_set
F3	Firstly, we need to find the appropriate scaling for the training. Then apply this mapping for both train and test dataset before learning.

Section 4	Q3	question

4-3) which one is not true in model selection:

```
models = []

1) models.append(('LR', LogisticRegression()))

models.append(('KNN', KNeighborsClassifier()))
```

```
2) models.append(('NB', GaussianNB()))

models.append(('SVC', SupportVectorClassifier()))

3) models.append(('LSVC', LinearSVC()))

models.append(('RFC', RandomForestClassifier()))

4) models.append(('DTR', DecisionTreeRegressor()))
```

Section 4	Q3	Solution

4-3) 2, the correct is as follows:

```
models = []

models.append(('LR', LogisticRegression()))

models.append(('KNN', KNeighborsClassifier()))

models.append(('NB', GaussianNB()))

models.append(('SVC', SVC()))

models.append(('LSVC', LinearSVC()))

models.append(('RFC', RandomForestClassifier()))

models.append(('DTR', DecisionTreeRegressor()))
```

Section 4	Q3	Feedback

F1	Please refer to `linear_model`, `neighbors`, `naive_bayes`, `svm`, `tree`, `ensemble` packages of `sklearn` library
F2	Create a list of different models in a tuple with the models name, and the model class.
F3	All models in sklearn do not need input. Check for the correct name of the functions.

Section 4	Q4	question

4-4) Find the best model: which one is wrong

```
seed = 7
results = []
names = []
X = train_set_scaled
Y = train_set_labels

for name, model in models:
    kfold = model_selection.$1(
       $2=10, random_state=seed, shuffle=True)
    cv_results = model_selection.$4(
        model,$3, cv=kfold, scoring='accuracy')
    results.append(cv_results)
    names.append(name)
    msg = "%s: %f (%f)" % (
        name, cv_results.mean(), cv_results.std())
    print(msg)
```

```
1) $1= cv

2) $2=n_splits

3) $3 = X, Y

4) $4 = cross_val_score
```

Section 4	Q4	Solution

4-4)1, the correct is `KFold`

To compare multiple algorithms with the same dataset, there is a very nice utility in sklearn called **model_selection**. We create a list of algorithms and then we score them using the same comparison method. At the end we pick the one with the best score.

The code is as follows:

```
# Prepare the configuration to run the test
seed = 7
results = []
names = []
X = train_set_scaled
```

```
Y = train_set_labels

# Every algorithm is tested and results are
# collected and printed
for name, model in models:
    kfold = model_selection.KFold(
        n_splits=10, random_state=seed, shuffle=True)
    cv_results = model_selection.cross_val_score(
        model, X, Y, cv=kfold, scoring='accuracy')
    results.append(cv_results)
    names.append(name)
    msg = "%s: %f (%f)" % (
        name, cv_results.mean(), cv_results.std())
    print(msg)
```

Section 4	**Q4**	**Feedback**

F1	To compare multiple algorithms with the same dataset, there is a very nice utility in sklearn called **model_selection**. More information link.
F2	To have a fair comparison of all approaches, k-fold cross validation is used to verify accuracy in training. For each mode, first create a k-fold cross validation, then pass that class to the modul to compute the results.
F3	By adding `kfold` for dataset, we pass it to the `cross_val_score` and compute the accuracy for each model, given the inputs/output.

Section 4	**Q5**	**question**

4-5) Which statement is false in terms of accuracy:

1) Naive bayes have the largest range of change among other methods.

2) The most robust approach, with the highest worst performance is the DTR

3) On average the least performed method is naive bayes.

4) On average the highest performing method is SVC.

Section 4	Q5	solution

4-5) 2

The highest least performance, showing the robustness, is determined by the lower bound of the performance quartile.

We can check the result with the following code:

```
fig = plt.figure()

fig.suptitle('Algorithm Comparison')

ax = fig.add_subplot(111)

plt.boxplot(results)

ax.set_xticklabels(names)

plt.show()
```

Section 4	Q5	feedback

F1	Please refer to the boxplot of the results using this link.
F2	The performance of an approach should be determined with a different validation set to guarantee the validity of the measurement. This will be determined by the average performance as the mean of different results, the upper and lower bound as the 1.5 IQR and the min and max as the range. More information on the quartile can be found here.
F3	Robustness of an approach is defined based on the performance in the worst case which is defined by lower bound of the IQR. The range however, is defined by the difference in min and max values.

Section 4	Q6	question

4-6) Hyper parameter tuning for the best model. Which one is wrong:

```
param_grid = {
    'C': [1.0, 10.0, 50.0],
   $1: ['linear', 'rbf', 'poly', 'sigmoid'],
    'shrinking': [True, False],
   $2: ['auto', 1, 0.1],
```

```
    'coef0': [0.0, 0.1, 0.5]
}

model_best = <from previous question>

grid_search = GridSearchCV(
    model_best, param_grid, cv=10,$3='accuracy')
grid_search.fit($4, train_set_labels)
```

1)$1 = `'kernel'`

2)$2 = `'alpha'`

3)$3 = `scoring`

4)$4 = `train_set_scaled`

Section 4	Q6	Solution

4-6) 2, the correct is `'gamma'`

The default parameters for an algorithm are rarely the best ones for our dataset. Using sklearn we can easily build a parameters grid and try all the possible combinations. At the end we inspect the `best_estimator_` property and get the best ones for our dataset.

```
param_grid = {
   'C': [1.0, 10.0, 50.0],
   'kernel': ['linear', 'rbf', 'poly', 'sigmoid'],
   'shrinking': [True, False],
   'gamma': ['auto', 1, 0.1],
   'coef0': [0.0, 0.1, 0.5]
}


model_best = SVC()

grid_search = GridSearchCV(model_best, param_grid, cv=10,scoring='accuracy')
grid_search.fit(train_set_scaled, train_set_labels)
```

Section 4	Q6	Feedback

F1	For more information on the grid based cross-validation, please refer to this

	link.
F2	Grid search validation is a method of hyper parameter tuning. We define the candidate hyper-parameters including the model structure, learning rate, and epochs. Then we conduct the search on the best model. During search we need to set the method for measuring the performances.
F3	In `param_grid` we define the hyper parameter set (more info link). We define the best model `model_best` then use `GridSearchCV` to pass the mode, grid, number of cv, and scoring. We finally use `grid_search.fit` to apply the this models to the appropriate training dataset.

Section 4	**Q7**	**question**

4-7- what is the best accuracy in grid search?

1)`0.76872964`

2)`0.757271`

3)`0.763802`

4)`0.755632`

Section 4	**Q7**	**Solution**

`4-7) 1`

The code is:

```
grid_search.best_score_
```

Section 4	**Q7**	**Feedback**

F1	For more information on the property of grid based cross-validation, please refer to this link.

F2	With grid search CV we now have access to so many different internal functionality which stores best accuracy, best parameters, and best estimator.
F3	By utilizing the `grid_search`, we have access to internal properties including **best_estimator_, best_score_, best_params_, best_index_, scorer_ , etc**. This is accessible via `grid_search.<internal funcitonality>`

Section 4	**Q8**	**question**

4-8) train with the best hyper parameters

```
model = grid_search.$1

X = np.append($2, test_set_scaled, axis=0)
Y = np.append($3, test_set_labels, axis=0)
model.fit($4)

1)$1 = best_estimator_
2)$2 = train_set_scaled
3)$3 = train_set_labels
4)$4 = dataset
```

Section 4	**Q8**	**Solution**

4-8) 4, the answer `X, Y`
The code:

```
# Create an instance of the algorithm using parameters
# from best_estimator_ property
model = grid_search.best_estimator_

# Use the whole dataset to train the model
X = np.append(train_set_scaled, test_set_scaled, axis=0)
Y = np.append(train_set_labels, test_set_labels, axis=0)

# Train the model
model.fit(X, Y)
```

Section 4	**Q8**	**Feedback**

F1	To fit a function with s sklearn mode, one can see this SVM as an example. link
F2	After finding the best model, and the best hyper-parameter, now we can use all the test/train models to train a final best performing model.
F3	First we need to find the best estimator from the `grid_search`. Then we augment the separated train and test set for the features and output vectors. Finally we train the model with `model.fit`

Section 5	**Prediction**

Corresponding library:

Section 5	**Q1**	**question**

5-1) what is the probability of the person having type 2 `diabetes` for the following features:

```
[6, 168, 72, 35, 0, 43.6, 0.627, 65]
```

1) 0

2) 0.25

3) 0.75

4) 1

Section 5	**Q1**	**solution**

5-1) 4 the code is as follows:

```
# We create a new (fake) person having the three most correated values high

new_df = pd.DataFrame([[6, 168, 72, 35, 0, 43.6, 0.627, 65]])

# We scale those values like the others

new_df_scaled = scaler.transform(new_df)

# We predict the outcome
```

```
prediction = model.predict(new_df_scaled)

# A value of "1" means that this person is likely to have type 2 diabetes

print(prediction)
```

Section 5	**Q1**	**feedback**

F1	After defining the sample input in a panda dataframe, we can use the following link as an example to predict a sample output result.
F2	For a new sample within the same distribution, and features, we can create a panada dataframe (link). Then we can use the final model to predict the outcome to this sample input.
F3	In binary classification, with sklearn we can predict the result of a sample input with `pd.DataFrame`. Using the `model.predict` of the scaled input will result the output of the sample.

The final code (also available in github as google colab):

Block 1: (section 2) Import

```
import os
import pandas as pd
import numpy as np
from numpy import unique

from pandas import read_csv
import seaborn as sns
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler as Scaler
from sklearn.linear_model import LogisticRegression
```

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.svm import LinearSVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeRegressor
from sklearn import model_selection
from sklearn.model_selection import GridSearchCV
```

Block 2: (section 3) Data analysis

```
# load the dataset
url =
'https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabe
tes.csv'
dataset =pd.read_csv(url , header=None)
dataset.columns = [
    "NumTimesPrg", "PlGlcConc", "BloodP",
    "SkinThick", "TwoHourSerIns", "BMI",
    "DiPedFunc", "Age", "HasDiabetes"]

# get the values
values = dataset.values
X, y = values[:, :-1], values[:, -1]
# gather details
n_rows = X.shape[0]
n_cols = X.shape[1]
classes = unique(y)
n_classes = len(classes)
# summarize
print('N Examples: %d' % n_rows)
print('N Inputs: %d' % n_cols)
print('N Classes: %d' % n_classes)
print('Classes: %s' % classes)
print('Class Breakdown:')
# class breakdown
breakdown = ''
for c in classes:
      total = len(y[y == c])
      ratio = (total / float(len(y))) * 100
      print(' - Class %s: %d (%.5f%%)' % (str(c), total, ratio))

corr = dataset.corr()

# showing the visulization

sns.heatmap(corr, annot = True)
```

```
dataset.hist(bins=50, figsize=(20, 15))

plt.show()

Features_missing = ['BMI', 'BloodP', 'PlGlcConc','SkinThick','TwoHourSerIns']
for feature in Features_missing:
      median = dataset[feature].median()
      # Substitute it in the BMI column of the dataset where values are 0
      dataset[feature] = dataset[feature].replace(to_replace=0, value=median)
```

Block 3: (section 4) Modeling

```
train_set, test_set = train_test_split(dataset, test_size=0.2, random_state=42)

# Separate labels from the rest of the dataset

train_set_labels = train_set["HasDiabetes"].copy()

train_set = train_set.drop("HasDiabetes", axis=1)

test_set_labels = test_set["HasDiabetes"].copy()

test_set = test_set.drop("HasDiabetes", axis=1)

scaler = Scaler()

scaler.fit(train_set)

train_set_scaled = scaler.transform(train_set)

test_set_scaled = scaler.transform(test_set)

models = []

models.append(('LR', LogisticRegression()))

models.append(('KNN', KNeighborsClassifier()))

models.append(('NB', GaussianNB()))

models.append(('SVC', SVC()))

models.append(('LSVC', LinearSVC()))

models.append(('RFC', RandomForestClassifier()))

models.append(('DTR', DecisionTreeRegressor()))

# Prepare the configuration to run the test
seed = 7
results = []
names = []
X = train_set_scaled
Y = train_set_labels
```

```
# Every algorithm is tested and results are
# collected and printed
for name, model in models:
    kfold = model_selection.KFold(
        n_splits=10, random_state=seed, shuffle=True)
    cv_results = model_selection.cross_val_score(
        model, X, Y, cv=kfold, scoring='accuracy')
    results.append(cv_results)
    names.append(name)
    msg = "%s: %f (%f)" % (
        name, cv_results.mean(), cv_results.std())
    print(msg)

fig = plt.figure()

fig.suptitle('Algorithm Comparison')

ax = fig.add_subplot(111)

plt.boxplot(results)

ax.set_xticklabels(names)

plt.show()



param_grid = {
   'C': [1.0, 10.0, 50.0],
   'kernel': ['linear', 'rbf', 'poly', 'sigmoid'],
   'shrinking': [True, False],
   'gamma': ['auto', 1, 0.1],
   'coef0': [0.0, 0.1, 0.5]
}

model_best = SVC()

grid_search = GridSearchCV(
    model_best, param_grid, cv=10,scoring='accuracy')
grid_search.fit(train_set_scaled, train_set_labels)


grid_search.best_score_



# Create an instance of the algorithm using parameters
# from best_estimator_ property
svc = grid_search.best_estimator_
```

```
# Use the whole dataset to train the model
X = np.append(train_set_scaled, test_set_scaled, axis=0)
Y = np.append(train_set_labels, test_set_labels, axis=0)

# Train the model
svc.fit(X, Y)
```

Block 4: (section 5) Prediction

```
# We create a new (fake) person having the three most correated values high

new_df = pd.DataFrame([[6, 168, 72, 35, 0, 43.6, 0.627, 65]])

# We scale those values like the others

new_df_scaled = scaler.transform(new_df)

# We predict the outcome

prediction = svc.predict(new_df_scaled)

# A value of "1" means that this person is likley to have type 2 diabetes

print(prediction)
```

Course name: **classification**

Problem: **2 (Multi-label classification)**

title: **Wine quality prediction**

Section 1	Problem description

The two datasets from UCI are related to red and white variants of the Portuguese "Vinho Verde" wine. The reference [Cortez et al., 2009].

The classes are ordered and not balanced (e.g. there are much more normal wines than excellent or poor ones). Two datasets were combined and few values were randomly removed. The goal is to find the wine quality level given the input variables.

Attribute Information:

Input variables (based on physicochemical tests):

1 - fixed acidity

2 - volatile acidity

3 - citric acid

4 - residual sugar

5 - chlorides

6 - free sulfur dioxide

7 - total sulfur dioxide

8 - density

9 - pH

10 - sulphates

11 - alcohol

Output variable (based on sensory data):

12 - quality (score between 0 and 10)

The original dataset is available at **Kaggle** and can be downloaded from this address:

https://www.kaggle.com/datasets/rajyellow46/wine-quality

Section 1	Q1	question

1-1- What is the machine learning task?

1) **Supervised binary classification**
2) **Supervised multi-label classification**
3) **Unsupervised binary clustering**
4) **Unsupervised multi-category grouping**

Section 1	Q1	solution

1) These datasets can be viewed as classification or regression tasks. But since we are looking to find the level of quality, it is a multi-label classification problem.

To solve the problem we will have to analyze the data, do any required transformation and normalization, apply a machine learning algorithm, train a model, check the performance of the trained model and iterate with other algorithms until we find the most performant for our type of dataset.

Section 1	Q1	Feedback

F1	Get more information about the different tasks in this link.
F2	Look back to the problem description. Is the prediction output discrete or continuous? How many different output results are there?
F3	If the dataset has label/output, then the ML model is supervised. If the number of discrete outputs are more than two, that is a multi-label problem.

Section 2	Python requirements

Section 2	Q1	question

1-1)

```
import numpy as np
import pandas as pd
import <?> as plt
import seaborn as sb
```

Section 2	Q1	Solution

1-1) `matplotlib.pyplot`

We use matplotlib.pyplot for visualization purposes.

Section 2	Q1	Feedback

F1	Answer this question after you went through all segments to know which packages are required.
F2	The missing package is for visualization purposes. Find an appropriate module to include in your code.
F3	Matplotlib is a useful tool for visualization. Look at the link to know how to include them in your code.

Section 2	Q2	question

1-1)

```
from sklearn.model_selection import train_test_split
from <?> import MinMaxScaler
from sklearn import metrics
from sklearn.svm import SVC
from xgboost import XGBClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier



import warnings
warnings.filterwarnings('ignore')
```

Section 2	Q1	Solution

1-1) `sklearn.preprocessing`

We need to use the preprocessing package of sklearn to in order to normalize the input features.

Section 2	Q1	Feedback

F1	The missing package is required for normalizing the input/output features.
F2	Find a module in Sklearn package which has the MinMaxScaler function.
F3	For preprocessing, we use sklearn to normalize the inputs. Use this link to find the appropriate function to include.

Section 3	Data Analysis

Section 3	Q1	question

3-1-

```
# Load the dataset
url =
'https://raw.githubusercontent.com/reubengazer/Wine-Quality-Analysis/master/winequalityN.csv'
df =<?>(url)
print(df.head())
```

Section 3	Q2	Solution

4) `pd.read_csv`

We need to use the panda library to read dataset from URL.

Section 3	Q1	Feedback

F1	To load a dataset from URL, we can use panada library.
F2	Please refer to this link to load dataset.
F3	Read_csv from panda package is used to load csv dataset. By default this will ignore the first line which is the header. Features names can be added as dataset columns.

Section 3	Q2	question

3-2- Which statement is not true about the dataset?

1) The number of data points are 6496

2) There are 11 features

3) there are 10 classes

4) The first quartile value of pH is 3.14

Section 3	Q2	Solution

4) The first quartile value (25%) of pH is 3.11, which can be computed with the pd.describe

```
df.info()
df.describe().T
```

Section 3	Q2	Feedback

F1	Use the EDA toolbox to check this. The number of rows in a dataset are the total data points, and columns are the number of features. The number of unique outputs determines the class number.
F2	The second quartile is the median. The first and third quartile are data points that the number of values less than those point are 25%, 75% of the total data data.
F3	You can use the info and describe modules in python to check the answer.

Section 3	Q3	question

3-4- Which of the following features have the most number of missing values?

1) Fixed acidity

2) Volatile acidity

3) alcohol

4) There is no missing value

Section 3	Q3	Solution

3-4) 1

```
df.isnull().sum()
```

The results are:

```
type                   0
fixed acidity         10
volatile acidity       8
citric acid            3
residual sugar         2
chlorides              2
free sulfur dioxide    0
total sulfur dioxide   0
density                0
pH                     9
sulphates              4
alcohol                0
quality                0
dtype: int64
```

Section 3	Q3	Feedback

F1	Please use the EDA toolbox > histogram plot.
F2	Missing values are either empty entry, np.nan, or zero. The latest is used only if zero value does not provide meaningful information.
F3	Use the isnull function of pandas to see how many null values you have. `df.isnull().sum()`

Section 3	Q4	question

3-5-

```
for col in df.columns:
  if df[col].isnull().sum() > 0:
    df[col] = df[col].<?>(df[col].mean())

df.isnull().sum().sum()
```

Section 3	Q4	Solution

3-5)`fillna`

We have noticed from the previous analysis that there are missing data for some of the features. Machine learning algorithms don't work very well when the data is missing so we have to find a solution to "clean" the data we have.

The easiest option could be to eliminate all those data with null/zero values, but in this way we would eliminate a lot of important data.

Another option is to calculate the **mean** value for a specific column and substitute that value everywhere (in the same column) we have zero or null.

Section 3	Q4	Feedback

F1	Check this link to handle missing data.
F2	There are two approaches to handle missing data: remove/impute. If only one feature is missing, it is better not to remove the whole dataset.
F3	You need to impute the dataset, change the null values with the column mean.

Section 3	Q5	question

3-3-Which statement is not true?

1) The dataset is imbalanced.

2) Chloride is the least-sparse feature.

3) pH is normally distributed with the mean around 3.2

4) All features are linearly independent.

Section 3	Q5	Solution

3-3) 4. We need to plot the histogram.

```
df.hist(bins=20, figsize=(10, 10))
plt.show()
```

As we can see, the histogram of the last column is equally distributed. The chloride has 3000 value for it's histogram, which shows it is more dense. For the linear dipendensy, let us use this code:

```
plt.figure(figsize=(12, 12))
sb.heatmap(df.corr() > 0.7, annot=True, cbar=False)
plt.show()
```

We can see two features "free sulfur dioxide" and "total sulfur dioxide" are redundant (their linear correlation is bigger than 0.7).

Section 3	Q5	Feedback

F1	Use the visualization toolbox, and check for histogram/ group plots.
F2	The correlation matrix is an important tool to understand the correlation between two characteristics. The values range from -1 to 1 and the closer a value is to 1 the greater the two quantities relations.

F3	If a linear correlation between two values are high (greater than 0.7), then these two quantities are highly correlated and can be omitted.

Section 3	Q6	question

3-3-

```
Feature1 = 'total sulfur dioxide'
Feature2 = 'chlorides'
f = df.drop(<?>, axis=1)
```

Section 3	Q6	Solution

3-3) `Feature1`

As we know the `'total sulfur dioxide'` and `'free sulfur dioxide'` are redundant, we can drop one of them.

Section 3	Q6	Feedback

F1	Please fill the placeholder with one of the `Feature1` or `Feature2`
F2	Use the visualization toolbox, and check for group plots to see if any of these variables `Feature1` or `Feature2` need to be removed.
F3	By using the heatmap in the previous question, which of these features are redundant, meaning it is highly correlated with other variable.

Section 3	Q7	question

3-3-

```
# Make an output of good (quality >5) and bad (quality <5)

df['best quality'] = [1 if x > 5 else 0 for x in  df.<?>]
```

Section 3	Q7	Solution

3-3) `quality`

We create a new output feature, which is 1 when the quality is greate than 5 and 0 when it is not.

Section 3	Q7	Feedback

F1	We can change the multi-class classification to binary classification by creating a new output feature that is two classes.
F2	Try to look over different values of the wine quality column, assign a new column to 1 if it is greater than 5 and 0 if it is less than 5.
F3	Different columns in a panda dataframe can be accessed, the same as it does for methods in classes. Look at this link for more information.

Section 3	Q8	question

3-3-

```
# Change the text type column to numerical value (0,1)

<?>({'white': 1, 'red': 0}, inplace=True)
```

Section 3	Q8	Solution

3-3) `df.replace`

We need to replace the ‘white’ value with 1 and the ‘red’ value to 0 in the type column.

Section 3	Q8	Feedback

F1	You need to change string values to numerical values.
F2	You should replace the ‘white’ value with 1 and the ‘red’ value to 0 in the type column.
F3	Using panda functions, we can substitute any string/numerical values with any string/numerical values. This can be done in place, where the output size is the same as the input size. Look at this link for more information.

Section 4	Model

Section 4	Q1	question

4-1-

```
features = df.drop(<?>, axis=1)
target = df['best quality']

xtrain, xtest, ytrain, ytest = train_test_split(
     features, target, test_size=0.2, random_state=40)

xtrain.shape, xtest.shape
```

Section 4	Q1	Solution

4-1) `['quality', 'best quality']`

We need to get remove the “quality” column and the created “best quality” for the features. We split the dataset to 20% for test and rest train.

Section 4	Q1	Feedback

F1	To avoid overfitting, the dataset should be divided into train/test. More information in this link. The dataset should include of features as inputs and target as output.
F2	The features should include the relevant items of input columns.
F3	You need to drop the columns that are considered as output.

Section 4	Q2	question

4-2-

```
norm = MinMaxScaler()
xtrain = norm.<?>
xtest = norm.transform(xtest)
```

Section 4	Q2	Solution

4-2) `fit_transform(xtrain)`

One of the most important data transformations we need to apply is the features scaling. Basically most of the machine learning algorithms don't work very well if the features have a different set of values.

Section 4	Q2	Feedback

F1	Normalising the data before training help us to achieve stable and fast training of the model. . More information at this link.
F2	In the preprocessing stage, data normalization/data whitening is essential as most of the machine learning algorithms don't work very well if the features have a different set of values. The linear transformation should apply the same for train/test_set
F3	Firstly, we need to find the appropriate scaling for the training. Then apply this mapping for both train and test dataset before learning.

Section 4	Q3	question

4-3)

```
models = [LogisticRegression(), XGBClassifier(), SVC(kernel='rbf'),
KNeighborsClassifier(), RandomForestClassifier()]

for i in range(len(models)):
     # learn the model
     <?>(xtrain, ytrain)
```

Section 4	Q3	Solution

4-3) `models[i].fit`

Section 4	Q3	Feedback

F1	Please refer to `linear_model`, `neighbors`, `svm`, `tree`, `ensemble` packages of `sklearn` library
F2	For the created list of models, we need to learn for the train input and labels.
F3	All models in sklearn use the same scheme to train. Look linear model as an example in this link.

Section 4	Q4	question

4-4)

```
 print(f'{models[i]} : ')
 print('Training Accuracy : ', metrics.roc_auc_score(ytrain,
models[i].predict(xtrain)))
 print('Validation Accuracy : ', metrics.roc_auc_score(
   <?>))
 print()
```

Section 4	Q4	Solution

4-4)`ytest, models[i].predict(xtest)`

We need to compute the validation accuracy from the ROC plot.

Section 4	Q4	Feedback

F1	To compute the training/validation error, one can use ROC plot in classification tasks. More information link.
F2	For validation accuracy, we need to have the prediction of the model to the test inputs and the corresponding test dataset label.
F3	Each model can be evaluate for the test dataset by `model_i.predict`. By passing the test dataset and labels ,`ytest`, to the ROC, we can capture the evaluation accuracy.

Section 4	Q5	question

4-5) Which statement is false in classification task:

1) Logistic regression has the worst results

2) All train dataset in random forest are correctly predicted.

3) Random forest has the best results.

4) XGBoost has the least discrepancy between train and test validation

Section 4	Q5	solution

4-5) 4

According to the result of the previous question, we know that the least change in train/test is for logistic regression with `0.70197 - 0.6937 = 0.008`difference.

Section 4	Q5	feedback

F1	Please refer to the ROCplot of the results using this link.
F2	The worst result is the minimum evaluation error, while the best results is the maximum evaluation among the used models.
F3	The discrepancy between train and test is a measure of over fitting. The difference between train/test validation can reflect it.

Section 4	Q6	question

4-5)

```
evals = [metrics.roc_auc_score(ytest, models[i].predict(xtest)) for i in
range(len(models))]
# find the best model
best_model_index = max(range(len(evals)), key=evals.__getitem__)
# confusion matrix of the test results
metrics.plot_confusion_matrix(models[best_model_index],<?>)
plt.show()
```

Section 4	Q6	solution

4-5) `xtest, ytest`

We are going to find the confusion_matrix of the best model. We need to path x,y values of test.

Section 4	Q6	feedback

F1	Please refer to the confusion matrix of the results using this link.
F2	After we found the best model in terms of test accuracy, we are going to find the confusion matric value of the test and train dataset.
F3	Pass the value of x,y for test dataset to plot the confusion matrix.

Section 4	Q7	question

4-6) Hyper parameter tuning for the best model. Which one is wrong:

```
grid_search = GridSearchCV(models[best_model_index], param_grid,
cv=10,scoring='accuracy')
grid_search.fit(<?>, train_set_labels)
```

Section 4	Q7	Solution

4-6) NOT INCLUDE

We are not using any Grid search method for the model selection, as we have not defined the parameters that we want to search on.

Section 4	Q7	Feedback

F1	Grid search validation is a method of hyper parameter tuning. For more information on the grid based cross-validation, please refer to this link.
F2	We define the candidate hyper-parameters including the model structure (more info link)., learning rate, and epochs. Then we conduct the search on the best model. During search we need to set the method for measuring the performances.
F3	Grid search hyperparameter tuning is an optional step. For this, we need to have the parameter set for grid search, otherwize we cannot use it.

Section 4	Q8	question

4-5) Which statement is false in this classification of the best model:

1) There are 213 samples that are wrongly classified

2) The weighted average of the recall is 0.81

3) The f1-score of the poor wine class is 0.77

4) The precision of the high quality wine is 85%

Section 4	Q8	solution

4-5) 2

There are 126+87 = 213 samples in the random forest (best classifier) that are wrongly classified. Using the following code, we can compute the rest

```
print(metrics.classification_report(ytest,models[best_model_index].predict(xtest)))
```

Section 4	Q8	feedback

F1	Please refer to the classification report of the results using this link.
F2	The total number of mis-classificaion are the summation of the off-diagonal in the confusion matrix.
F3	The Precision, recall, and f-1 score are computed based on the rate of true/false positive/negative and can be accessed via the `classification_report` of the test target and prediction

Section 5	Prediction

Section 5	Q1	question

5-1)

```
# We create a new value of wine features
new_df = pd.DataFrame([[0,5,1.5,1.5,3,0.08,2,0.99,3.1,0.6,12.1]])
```

```
prediction = models[best_model_index].predict(<?>)
print(prediction)
```

Section 5	**Q1**	**solution**

5-1) `norm.transform(new_df)`

In order to find the prediction to a new value (in the same distribution of train), we first need to transform the data with the normalizer and the pass it to the model predict

Section 5	**Q1**	**feedback**

F1	To predict new data, we need to make sure the data has the same distribution as the training.
F2	We need to normalize the raw input value before we pass it to the model predict.
F3	We can use the same `norm.transform` for the new dataset, and pass the transformed dataset to the prediction model.

The final code (also available in github as google colab):

Block 1: (section 2) Import

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sb
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn import metrics
from sklearn.svm import SVC
from xgboost import XGBClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier


import warnings
warnings.filterwarnings('ignore')
```

Block 2: (section 3) Data analysis

```
# Load dataset
url =
'https://raw.githubusercontent.com/reubengazer/Wine-Quality-Analysis/maste
r/winequalityN.csv'
df  =pd.read_csv(url)
print(df.head())
```

```
df.info()
```

```
df.describe().T
```

```
df.isnull().sum()
```

```
for col in df.columns:
 if df[col].isnull().sum() > 0:
   df[col] = df[col].fillna(df[col].mean())
df.isnull().sum().sum()
```

```
df.hist(bins=20, figsize=(10, 10))
plt.show()
```

```
plt.figure(figsize=(12, 12))
sb.heatmap(df.corr() > 0.7, annot=True, cbar=False)
plt.show()
```

```
df = df.drop('total sulfur dioxide', axis=1)
```

```
df['best quality'] = [1 if x > 5 else 0 for x in df.quality]
```

```
df.replace({'white': 1, 'red': 0}, inplace=True)
```

Block 3: (section 4) Modeling

```
features = df.drop(['quality', 'best quality'], axis=1)

target = df['best quality']

xtrain, xtest, ytrain, ytest = train_test_split(

   features, target, test_size=0.2, random_state=40)

xtrain.shape, xtest.shape
```

```
norm = MinMaxScaler()

xtrain = norm.fit_transform(xtrain)

xtest = norm.transform(xtest)
```

```
models = [LogisticRegression(), XGBClassifier(), SVC(kernel='rbf'),
KNeighborsClassifier(), RandomForestClassifier()]


for i in range(len(models)):
 models[i].fit(xtrain, ytrain)


 print(f'{models[i]} : ')
 print('Training Accuracy : ', metrics.roc_auc_score(ytrain,
models[i].predict(xtrain)))
 print('Validation Accuracy : ', metrics.roc_auc_score(
   ytest, models[i].predict(xtest)))
 print()
```

```
evals = [metrics.roc_auc_score(ytest, models[i].predict(xtest)) for i in
range(len(models))]
best_model_index = max(range(len(evals)), key=evals.__getitem__)
metrics.plot_confusion_matrix(models[best_model_index], xtest, ytest)
plt.show()
```

```
print(metrics.classification_report(ytest,


models[best_model_index].predict(xtest)))
```

Block 4: (section 5) Prediction

```
# We create a new (fake) person having the three most correated values
high

new_df = pd.DataFrame([[0,5,1.5,1.5,3,0.08,2,0.99,3.1,0.6,12.1]])

# We scale those values like the others

new_df_scaled = norm.transform(new_df)

prediction = models[best_model_index].predict(new_df_scaled)

print(prediction)
```

Course name: **classification**

Problem: **3 (Binary classification)**

title: **Health Insurance Cross Sell**

Section 1	**Problem description**

An insurance policy is an arrangement by which a company undertakes to provide a guarantee of compensation for specified loss, damage, illness, or death in return for the payment of a specified premium. A premium is a sum of money that the customer needs to pay regularly to an insurance company for this guarantee.

Building a model to predict whether a customer would be interested in Vehicle Insurance is extremely helpful for the company because it can then accordingly plan its communication strategy to reach out to those customers and optimize its business model and revenue.

Now, in order to predict whether the customer would be interested in Vehicle insurance, you have information about demographics (gender, age, region code type), Vehicles (Vehicle Age, Damage), Policy (Premium, sourcing channel) etc. So the dataset is as follows:

Output: Response of insurance (0 or 1)

Features:

1. id (int)
2. Gender (str)
3. Age (int)
4. Driving_License (int)
5. Region_Code (float)
6. Previously_Insured (int)
7. Vehicle_Age (str)
8. Vehicle_Damage 1 (str)
9. Annual_Premium (float)
10. Policy_Sales_Channel (float)
11. Vintage (int)

The last column of the dataset indicates the response of the insurance to be either approved or rejected

dataset is obtained from Kaggle in this Link

https://www.kaggle.com/datasets/anmolkumar/health-insurance-cross-sell-prediction?select=train.csv

Section 2	Python requirements

Section 2	Q1	question

Import the libraries here

Section 2	Q1	solution

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from sklearn.metrics import precision_score, recall_score, accuracy_score,
f1_score
import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline
```

Section 2	Q1	feedback

F1	Answer this question after you went through all segments to know which packages are required.
F2	You need the following packages: 1) data handling 2)working with arrays 3) used for data visualization purposes 4) pre-implemented functions to perform tasks from data preprocessing to model development and evaluation
F3	For the data preprocessing and model development look at this link to know how to include them in your code.

Section 3	**Data Analysis**

Section 3	**Q1**	**question**

The dataset is available in this link. Please load the dataset and view it's first 5 elements to get some idea about the data.

Section 3	**Q1**	**solution**

```
# Load dataset
url =
'https://raw.githubusercontent.com/datu-ca/ML/main/datasets/classification
/health_insurance/train.csv?token=GHSAT0AAAAAAB4SCUVR4VWD5IGEHXAW6LWYY7FZD
DQ'
df  =pd.read_csv(url)
print(df.head())
```

Section 3	**Q1**	**feedback**

F1	Please refer to this link to load dataset.
F2	You can use the url link directly in the panda read_csv. It will automatically download the dataset in your temp folder.
F3	Read_csv from panda package is used to load csv dataset. By default this will ignore the first line which is the header. Features names can be added as dataset columns.

Section 3	Q2	question

Try to see the type of all dataset and some statistical information like their mean/limits/quantile

Section 3	Q2	solution

```
df.info()

df.describe().T
```

Section 3	Q2	feedback

F1	You can use the EDA toolbox to check this.
F2	Getting general information about data is vital before applying machine learning models. Try to find panda libraries which will provide you some information about data.

F3	Use this link1, and link2 to find your code.

Section 3	Q3	question

Check if there is any missing data

Section 3	Q3	Solution

```
df.isnull().sum()
```

Section 3	Q3	feedback

F1	You can use the EDA toolbox > histogram plot.
F2	Missing values are either empty entry, np.nan, or zero. You can use the panda library to check if you have any null variables.
F3	You can use this link to find the syntax of your code.

Section 3	Q4	question

handle missing data

Section 3	Q4	Solution

```
# no missing data
```

Section 3	Q4	Feedback

F1	Check this link to handle missing data.
F2	There are two approaches to handle missing data: remove/impute. If only one feature is missing, it is better not to remove the whole dataset. The highest frequency of a feature, or mean is not the most robust approach to impute a dataset, as the null values can affect them.
F3	Make sure if you have missing data. If you don't you can simply ignore this section. If you do, use impute by replacing missing values with the average of it's group.

Section 3	Q5	question

If there is any non-numerical feature, try to convert categorical variables to Indicator columns

Section 3	Q5	Solution

```
# Convert categorical variables to Indicator columns
df2 = pd.get_dummies(df,drop_first=True)
df2.head()
```

Section 3	Q5	feedback

F1	For machine learning models, we need numerical values to evaluate the model. Hence, any non numerical value should be mapped to some values.
F2	If you do not have non-numerical values skip this stage. If you do, try to map those values to numerical ones. Make sure the distance in the numerical space makes sense for the actual feature. E.g. if it is color, it makes sense to have blue and green have closer values compared to blue and red, due to their wavelength.
F3	Use this link to change the categorical values automatically to numerical ones.

Section 3	Q6	question

Try to find non-informative variables:

1) Any variable that has a high correlation (>0.75) with another variable
2) Column that heuristically do not carry any information about the output

Section 3	Q6	solution

```
plt.figure(figsize=(12, 12))
sns.heatmap(df2.corr() > 0.75, annot=True, cbar=False)
```

```
plt.show()
# there is none
# Heuristically column id does not have informative knowledge
```

Section 3	Q6	feedback

F1	In machine learning, If there are some variables that logically does affect output or are not important, they can be safely deleted. Variables with high correlation are redundant and can reduce the performance.
F2	Use correlation method in panda to find the relative correlation with each two variables. You can also use heatmap for better visualization . Drop any columns that are not important using the panda method.
F3	Use this link to drop non important features. Use this link to find the correlation.

Section 4	Model

Section 4	Q1	question

Create the feature and target variables store as X, y

Section 4	Q1	solution

```
X = df2.iloc[:,[1,2,3,4,5,6,7,9,10,11,12]]
y = df2.iloc[:,8]
```

Section 4	Q1	feedback

F1	Choose the locations of the features and the output to x,y
F2	In panda, you can determine the column index to store that specific data for all data points.
F3	Use this link to find the syntax for chunking the dataframe.

Section 4	Q2	question

Check if your target variable is uniformly distributed

Section 4	Q2	solution

```
y.value_counts().plot(kind='bar',figsize=(3,3),title="Insurance sell")

plt.xlabel("Insurance Response")

plt.ylabel("Frequency")

y_pos=50000

x_pos=0

for i, v in enumerate(y.value_counts()):

   plt.text(i+x_pos, y_pos, v, horizontalalignment='center',
verticalalignment='center', fontsize=20)

plt.show()
```

Section 4	Q2	feedback

F1	Having uniform distribution of target variable is essential, since an imbalance dataset can result in bias. More info
F2	Plot a histogram like plot where each x-axis represent different values of target and y-axis represent the count of samples.
F3	For a panda series, you can use value_count to amortize number of occurrence. For more information use this link

Section 4	Q3	question

Scale your feature variables.

Section 4	Q3	solution

```
# define standard scaler

scaler = StandardScaler()

X= pd.DataFrame(scaler.fit_transform(X))
```

Section 4	Q3	feedback

F1	All features should have the scale between -1,1 or 0,1 so some features can not dominate the others. Try to scale all features. More information this link.
F2	You can apply the scaler fit to the panda dataframe as a function.

F3	Use this link to find the class of scaling in sklear. Then apply it to the features (x) dataframe.

Section 4	Q4	question

Split the dataset to train and test set for features and target variables

Section 4	Q4	solution

```
# Splitting dataset into training and test

X_train, X_test, y_train, y_test=  train_test_split(X,y,test_size=0.2)
```

Section 4	Q4	feedback

F1	To avoid overfitting, the dataset should be divided into train/test.
F2	We need to separate the output from the features in two dataframes, labels and test/train_set. A portion of data should be allocated to evaluate performance of the model to unseen data.
F3	Train_set portion should be relatively higher compared to the test_set as the model is only exposed to the train_set while learning. We can provide seed for separation to have reproducible results. More information in this link

Section 4	Q5	question

Try to fit a logistic regression model to the actual data and compute its confusion matrix

Section 4	Q5	solution

```
#  logistics regression

clf = LogisticRegression(random_state=0).fit(X_train, y_train)

y_pred=clf.predict(X_test)

# Plotting the Confusion Matrix

plot_confusion_matrix(clf,X_test, y_test)
```

Section 4	Q5	feedback

F1	We need to create the class of model and pass the train x,y. Then we can pass the test feature to predict the y test.
F2	Use this link to plot the confusion matrix of the test x,y.
F3	Please refer to this link to create the logistic regression class. They call the prediction method and find the confusion matrix given the test feature and label.

Section 4	Q6	question

Find accuracy, precision, recall and F1 score

Section 4	Q6	solution

```
print("Accuracy",accuracy_score(y_test, y_pred))

print("Precision", precision_score(y_test, y_pred))

print("Recall",recall_score(y_test, y_pred))

print("F1 score",f1_score(y_test, y_pred))
```

Section 4	Q6	feedback

F1	You need to look in sklearn package to find these metrics.
F2	For accuracy follow this link, for precision this link
F3	For recall follow this link, for F1 score this link

Section 4	Q7	question

Compute the revised fit by applying cost sensitive training

Section 4	Q7	solution

```
clf_cw = LogisticRegression(random_state=0,
class_weight='balanced').fit(X_train, y_train)

y_pred= clf_cw.predict(X_test)
```

```
print("Accuracy",accuracy_score(y_test, y_pred))

print("Precision", precision_score(y_test, y_pred))

print("Recall",recall_score(y_test, y_pred))

print("F1 score",f1_score(y_test, y_pred))
```

Section 4	Q7	feedback

F1	You can revise the logisticregression class to consider the unbalanced data.
F2	In the logistic regression model, you can change the default value of class_weight to consider imbalance data.
F3	Look for class_weight in link to set it balanced.

Section 4	Q8	question

Compute the revised fit by applying random under sampling. Then compute accuracy, precision, recall and F1 score

Section 4	Q8	solution

```
# pip install imblearn

from imblearn.under_sampling import RandomUnderSampler
```

```
# Create an instance of RandomUnderSampler, fit the training data #and
apply Logistics regression

rus= RandomUnderSampler()

X_train, y_train = rus.fit_resample(X_train, y_train)

clf_rus=  LogisticRegression(random_state=0).fit(X_train, y_train)

# predict the test data

y_pred= clf_rus.predict(X_test)

# print the model performance metrics

print("Accurcay",accuracy_score(y_test, y_pred))

print("Precision", precision_score(y_test, y_pred))

print("Recall",recall_score(y_test, y_pred))

print("F1 score",f1_score(y_test, y_pred))
```

Section 4	Q8	feedback

F1	As a preprocessing step, the undersampling can be applied to the train x,y before applying to the logistic regression.
F2	Using fit_sample method in imblearn, you can undersample the category of the data that are more compared to others.
F3	Use this link to fit the train feature and target before fitting into the logistic regression.

Section 4	Q9	question

Compute the revised fit by applying random over sampling. Then compute accuracy, precision, recall and F1 score

Section 4	Q9	solution

```
from imblearn.over_sampling import RandomOverSampler

# Create an instance of RandomOverSampler, fit the training data #and
apply Logistics regression

ros= RandomOverSampler(sampling_strategy='auto')

X_train, y_train = ros.fit_resample(X_train, y_train)

clf_ros=  LogisticRegression(random_state=0).fit(X_train, y_train)

# predict the test data

y_pred= clf_ros.predict(X_test)

# print the model performance metrics

print("Accurcay",accuracy_score(y_test, y_pred))

print("Precision", precision_score(y_test, y_pred))

print("Recall",recall_score(y_test, y_pred))

print("F1 score",f1_score(y_test, y_pred))
```

Section 4	Q9	feedback

F1	As a preprocessing step, the undersampling can be applied to the train x,y before applying to the logistic regression. Then recompute all the meters from previous section.
F2	Using fit_resample method in imblearn, you can undersample the category of the data that are more compared to others.
F3	Please refer to this link to sample the train feature and target before fitting into the logistic regression.

Section 4	Q10	question

Compute the revised fit by applying Synthetic Minority Over-sampling Technique (SMOTE). Then compute accuracy, precision, recall and F1 score

Section 4	Q10	solution

```
from imblearn.over_sampling import SMOTE

##Create an instance of SMOTE, fit the training data and apply #Logistics regression

sm = SMOTE(random_state=27, sampling_strategy='minority', k_neighbors=5)

X_train, y_train = sm.fit_resample(X_train, y_train)

clf_sm = LogisticRegression(random_state=0).fit(X_train, y_train)

# predict the test data

y_pred= clf_sm.predict(X_test)

# print the model performance metrics
```

```
print("Accurcay",accuracy_score(y_test, y_pred))

print("Precision", precision_score(y_test, y_pred))

print("Recall",recall_score(y_test, y_pred))

print("F1 score",f1_score(y_test, y_pred))
```

Section 4	Q10	feedback

F1	As a preprocessing step, the SMOTE can be applied to the train x,y before applying to the logistic regression.
F2	Using fit_resample method in imblearn, you can undersample the category of the data that are more compared to others.
F3	Please refer to this link to sample the train feature and target before fitting into the logistic regression.

Section 5	Prediction

Section 5	Q1	question

Make the prediction of the best model for following input

id	Age	Driving_License	Region_Code	Previously_Insured	Annual_Premium	Policy_Sales_Channel	Vintage	Gender	Vehicle_Age	Vehicle damager
3745	45	yes	150	yes	23864.0	153.0	125	Male	0.7	yes

Section 5	Q1	solution

First need to convert it to our dataset

Age	Driving_License	Region_Code	Previously_Insured	Annual_Premium	Policy_Sales_Channel	Vintage	Gender_male	Vehicle_Age<1	Vehicle_Age>2	Vehicle damager
45	1	150	1	23864.0	153.0	125	1	1	0	yes

```
# We create a new (fake) person having the three most correated values high

new_df = pd.DataFrame([[  45, 1,  15.0, 1,  23864.0,  153.0,  125 ,1, 1 ,0, 1]])

# We scale those values like the others

prediction= clf_rus.predict(new_df)

print(prediction)
```

Section 5	Q1	feedback

F1	First you need to convert the dataframe to your redefined format.
F2	Find the model you want to use and apply the predict method to get the prediction result for the input. Remember to scale/map the test input to match your training distribution.
F3	Create new dataframe for test using `pd.DataFrame` then use your leaned model and apply the predict. More info in this link.

Twitter sentiment analysis

Blocks category:

Function, Math, DATA, Variable, Control, UI, train, predict

Function:

def myFunction(): arg	**Defining function**
myFunction();	**Calling function**
//	**comment**
import	**Importing libraries**

Math:

+*/-	**arithmetic**
== != <> & | !	**assign/comparison**
randn	**Random number**

Data:

pd.read_csv	**Tabular data**
dataloader	**Dataloader for DL**
opencv	**Import image**

open	**Open a file**

Variable:

Dictionary, int/double/float, set	**Define variable**
log	**Logging variable**

Control:

train_test_split()	train/test split
while/if/for	**Loops, conditions**

UI:

print	**Show on console**
df.describe/info	**Get info from dataframe**
plot	**Plotting results**

Train:

model.fit	**Train a model**
sampling	**Apply rus/ros**
Hyper parameters	**Set hyper parameters**

Prediction:

model.predict	**Predict a learned model**
Accuracy, F1,F2, AOI	**metrics**

Twitter sentiment analysis blocks:

Imports (different colors)

`# utilities ()`
`import re`
`import pickle`
`import numpy as np`
`import pandas as pd`

`# plotting`
`import seaborn as sns`
`from wordcloud import WordCloud`
`import matplotlib.pyplot as plt`

`# nltk` `from nltk.stem import WordNetLemmatizer`

`# sklearn`
`from sklearn.svm import LinearSVC`
`from sklearn.naive_bayes import BernoulliNB`
`from sklearn.linear_model import LogisticRegressionfrom`
`sklearn.model_selection import train_test_split`
`from sklearn.feature_extraction.text import TfidfVectorizer`
`from sklearn.metrics import confusion_matrix, classification_report`

```
import time
import nltk
```

```

```

data

```
DATASET_COLUMNS  = ["sentiment", "ids", "date", "flag", "user", "text"]
DATASET_ENCODING = "ISO-8859-1"
dataset = pd.read_csv('/content/drive/MyDrive/twitter.csv',
encoding=DATASET_ENCODING, names=DATASET_COLUMNS, engine='python')
```

```

```

```

```

Data manipulation (pandas)

```
# Removing the unnecessary columns.
dataset = dataset[['sentiment','text']]
```

```
# Replacing the values to ease understanding.
dataset['sentiment'] = dataset['sentiment'].replace(4,1)
```

```

```

```

```

Console

```
#printing unique values
print(dataset['sentiment'].unique())
```

```
# Plotting the distribution for dataset.
ax = dataset.groupby('sentiment').count().plot(kind='bar',
title='Distribution of data',
                                               legend=False)
ax.set_xticklabels(['Negative','Positive'], rotation=0)

# Storing data in lists.
text, sentiment = list(dataset['text']), list(dataset['sentiment'])
```

```
print(f'Text Preprocessing complete.')
print(f'Time Taken: {round(time.time()-t)} seconds')
```

```
plt.figure(figsize = (20,20))
wc = WordCloud(max_words = 1000 , width = 1600 , height = 800,
               collocations=False).generate(" ".join(data_neg))
plt.imshow(wc)
```

```
wc = WordCloud(max_words = 1000 , width = 1600 , height = 800,
               collocations=False).generate(" ".join(data_pos))
plt.figure(figsize = (20,20))
plt.imshow(wc)
```

```
print('No. of feature_words: ', len(vectoriser.get_feature_names()))
```

```
# Print the evaluation metrics for the dataset.
   print(classification_report(y_test, y_pred))
```

```
   sns.heatmap(cf_matrix, annot=labels, fmt="", xticklabels =
categories, yticklabels = categories, cmap = 'Blues')
plt.xlabel("Predicted values", fontdict = {'size':14}, labelpad = 10)
   plt.ylabel("Actual values"   , fontdict = {'size':14}, labelpad = 10)
   plt.title ("Confusion Matrix", fontdict = {'size':18}, pad = 20)
```

```
```

Variables

```
# Defining dictionary containing all emojis with their meanings.
emojis = {':)': 'smile', ':-)': 'smile', ';d': 'wink', ':-E': 'vampire',
':(': 'sad',
          ':-(': 'sad', ':-<': 'sad', ':P': 'raspberry', ':O':
'surprised',
          ':-@': 'shocked', ':@': 'shocked',':-$': 'confused', ':\\':
'annoyed',
          ':#': 'mute', ':X': 'mute', ':^)': 'smile', ':-&': 'confused',
'$_$': 'greedy',
          '@@': 'eyeroll', ':-!': 'confused', ':-D': 'smile', ':-0':
'yell', 'O.o': 'confused',
          '<(-_-)>': 'robot', 'd[-_-]b': 'dj', ":'-)": 'sadsmile', ';)':
'wink',
          ';-)': 'wink', 'O:-)': 'angel','O*-)': 'angel','(:-D':
'gossip', '=^.^=': 'cat'}
## Defining set containing all stopwords in english.
stopwordlist = ['a', 'about', 'above', 'after', 'again', 'ain', 'all',
'am', 'an',
             'and','any','are', 'as', 'at', 'be', 'because', 'been',
'before',
             'being', 'below', 'between','both', 'by', 'can', 'd', 'did',
'do',
             'does', 'doing', 'down', 'during', 'each','few', 'for',
'from',
```

```
            'further', 'had', 'has', 'have', 'having', 'he', 'her',
'here',
            'hers', 'herself', 'him', 'himself', 'his', 'how', 'i',
'if', 'in',
            'into','is', 'it', 'its', 'itself', 'just', 'll', 'm', 'ma',
            'me', 'more', 'most','my', 'myself', 'now', 'o', 'of', 'on',
'once',
            'only', 'or', 'other', 'our', 'ours','ourselves', 'out',
'own', 're',
            's', 'same', 'she', "shes", 'should', "shouldve",'so',
'some', 'such',
            't', 'than', 'that', "thatll", 'the', 'their', 'theirs',
'them',
            'themselves', 'then', 'there', 'these', 'they', 'this',
'those',
            'through', 'to', 'too','under', 'until', 'up', 've', 'very',
'was',
            'we', 'were', 'what', 'when', 'where','which','while',
'who', 'whom',
            'why', 'will', 'with', 'won', 'y', 'you', "youd","youll",
"youre",
            "youve", 'your', 'yours', 'yourself', 'yourselves']
```

```
# Create Lemmatizer and Stemmer.
   wordLemm = WordNetLemmatizer()
```

```
# Defining regex patterns.
   urlPattern        = r"((http://)[^ ]*|(https://)[^ ]*|( www\.)[^ ]*)"
   userPattern       = '@[^\s]+'
   alphaPattern      = "[^a-zA-Z0-9]"
   sequencePattern   = r"(.)\1\1+"
   seqReplacePattern = r"\1\1"
```

```
tweetwords = ''
```

```
categories  = ['Negative','Positive']
   group_names = ['True Neg','False Pos', 'False Neg','True Pos']
```

```
   group_percentages = ['{0:.2%}'.format(value) for value in
cf_matrix.flatten() / np.sum(cf_matrix)]
```

```
labels = [f'{v1}\n{v2}' for v1, v2 in
zip(group_names,group_percentages)]
   labels = np.asarray(labels).reshape(2,2)
```

```
path = pre_path + 'vectoriser-ngram-(1,2).pickle'
```

```
path = pre_path + 'Sentiment-LR.pickle'
```

```
text = ["I hate twitter",
           "May the Force be with you.",
           "Mr. Stark, I don't feel so good"]
```

```
data = []
```

functions

```
def preprocess(textdata):

    return processedText
```

```
processedtext = preprocess(text)
```

```
def model_Evaluate(model):
```

```
def load_models(pre_path, Model_type):
   return vectoriser, model
```

```
def predict(vectoriser, model, text):
    return df
```

```
df = predict(vectoriser, model, text)
```

```
def load_models(path, Model_type):
    return vectoriser, model
```

```
```

```
```

controls

```
for tweet in textdata:
```

```
for emoji in emojis.keys():
```

```
for word in tweet.split():
```

```
if len(word)>1:
```

```
if Model_type == 'LR':
elif Model_type == 'BNB':
else: # SVC
```

```
for text, pred in zip(text, sentiment):
```

```
for text, pred in zip(text, sentiment):
```

Strings

```
tweet = tweet.lower()
```

```
# Replace all URls with 'URL'
tweet = re.sub(urlPattern,' URL',tweet)
```

```
tweet = tweet.replace(emoji, "EMOJI" + emojis[emoji])
```

```
# Replace @USERNAME to 'USER'.
       tweet = re.sub(userPattern,' USER', tweet)
```

```
 # Replace all non alphabets.
       tweet = re.sub(alphaPattern, " ", tweet)
```

```
# Replace 3 or more consecutive letters by 2 letter.
       tweet = re.sub(sequencePattern, seqReplacePattern, tweet)
```

```
# Lemmatizing the word.
               word = wordLemm.lemmatize(word)
               tweetwords += (word+' ')
```

```
data_neg = processedtext[:800000]
```

```
data_pos = processedtext[800000:]
```

utility

```
processedText.append(tweetwords)
```

```
nltk.download('wordnet')
nltk.download('omw-1.4')
```

```
t = time.time()
```

```
X_train, X_test, y_train, y_test = train_test_split(processedtext,
sentiment,
                                                    test_size = 0.05,
random_state = 0)
```

```
vectoriser = TfidfVectorizer(ngram_range=(1,2), max_features=500000)
```

```
vectoriser.fit(X_train)
```

```
X_train = vectoriser.transform(X_train)
X_test  = vectoriser.transform(X_test)
```

```
# Predict values for Test dataset
   y_pred = model.predict(X_test)
```

```
# Compute and plot the Confusion matrix
```

```
    cf_matrix = confusion_matrix(y_test, y_pred)
```

```
BNBmodel = BernoulliNB(alpha = 2)
```

```
BNBmodel.fit(X_train, y_train)
```

```
model_Evaluate(BNBmodel)
```

```
SVCmodel = LinearSVC()
```

```
SVCmodel.fit(X_train, y_train)
```

```
model_Evaluate(SVCmodel)
```

```
LRmodel = LogisticRegression(C = 2, max_iter = 1000, n_jobs=-1)
```

```
LRmodel.fit(X_train, y_train)
```

```
model_Evaluate(LRmodel)
```

```
file = open('vectoriser-ngram-(1,2).pickle','wb')
pickle.dump(vectoriser, file)
file.close()
file = open(path, 'rb')
```

```
vectoriser = pickle.load(file)
```

```
file.close()
```

```
file = open(path, 'rb')
model = pickle.load(file)
file.close()
```

```

```

```
textdata = vectoriser.transform(preprocess(text))
sentiment = model.predict(textdata)
```

```
data = []
```

```
data.append((text,pred))
```

```
df = pd.DataFrame(data, columns = ['text','sentiment'])
```

```
df = df.replace([0,1], ["Negative","Positive"])
```

```
vectoriser, model = load_models(pre_path, Model_type ='LR')
```

```
file = open('..path/vectoriser-ngram-(1,2).pickle', 'rb')
vectoriser = pickle.load(file)
file.close()
```

```
textdata = vectoriser.transform(preprocess(text))
sentiment = model.predict(textdata)
```

```
data.append((text,pred))
```

```
df = pd.DataFrame(data, columns = ['text','sentiment'])
df = df.replace([0,1], ["Negative","Positive"])
```

Explainable ai >> x ai

first/second/third difference

Features from the logfile, Fair ai for gender

Features: access to ai or not, time spent to the

Sequence modeling

Automatic and hand chosen features

HMM

problem #, use of AI, features = score that problem

Is this fair model? male /female demographic

2- datu library

Module 1.1: machine learning	Duration <1	Type concept	Category: Introduction
Machine learning is a subset of AI that enables computers to learn and improve from data without explicit programming. It uses algorithms to analyze data, identify patterns, and make predictions. This iterative process empowers machines to adapt and improve their performance over time.			
Machine learning is a field of study and application within artificial intelligence (AI) that focuses on developing computer systems capable of automatically learning and improving from experience without being explicitly programmed. It involves the development of algorithms and models that enable machines to analyze and interpret complex data, identify patterns, and make predictions or decisions based on the observed patterns. Machine learning algorithms are designed to iteratively learn from data, allowing them to adapt and improve their performance over time. **At its core, machine learning involves the utilization of statistical techniques and mathematical models to enable machines to learn and extract meaningful information from data. The learning process typically involves three key components: data, models, and optimization. First, large volumes of relevant and diverse data are collected, which serve as the input for the learning process. Next, a machine learning model is created, which represents the relationships and patterns within the data. This model is then trained using various algorithms and techniques to optimize its parameters and improve its ability to generalize from the provided data. Once the model is trained, it can be deployed to make predictions or decisions on new, unseen data, effectively leveraging the knowledge and patterns learned from the training phase. Machine learning finds applications in various domains, including image and speech recognition, natural language processing, recommender systems, fraud detection, and autonomous vehicles, among others.**			

Module 1.2: ML python packages	Duration: 5	Type code	Category: Introduction

Machine Learning is the science of programming computers to learn from data. It eliminates the need for explicit programming by utilizing algorithms and mathematical formulas. Python is a popular language for this task, with libraries like NumPy, SciPy, scikit-learn, TensorFlow, Keras, and PyTorch providing efficient solutions for data processing and analysis. These libraries enable tasks such as matrix processing, optimization, image manipulation, classical ML algorithms, deep learning, and neural network design, making Machine Learning more accessible and efficient than ever before.

Machine Learning, as the name suggests, is the science of programming a computer by which they are able to learn from different kinds of data. A more general definition given by Arthur Samuel is – "Machine Learning is the field of study that gives computers the ability to learn without being explicitly programmed." They are typically used to solve various types of life problems.

In the older days, people used to perform Machine Learning tasks by manually coding all the algorithms and mathematical and statistical formulas. This made the processing time-consuming, tedious, and inefficient. But in the modern days, it is become very much easy and more efficient compared to the olden days with various python libraries, frameworks, and modules. Today, Python is one of the most popular programming languages for this task and it has replaced many languages in the industry, one of the reasons is its vast collection of libraries. Python libraries that are used in Machine Learning are:

Numpy
Scipy
Scikit-learn
TensorFlow
Keras
PyTorch
Pandas
Matplotlib
NumPy is a very popular python library for large multi-dimensional array and matrix processing, with the help of a large collection of high-level mathematical functions. It is very useful for fundamental scientific computations in Machine Learning. It is particularly useful for linear algebra, Fourier transform, and random number capabilities. High-end libraries like TensorFlow uses NumPy internally for manipulation of Tensors.

SciPy is a very popular library among Machine Learning enthusiasts as it contains different modules for optimization, linear algebra, integration and statistics. There is a difference between the SciPy library and the SciPy stack. The SciPy is one of the core packages that make up the SciPy stack. SciPy is also very useful for image manipulation.

Scikit-learn is one of the most popular ML libraries for classical ML algorithms. It is built on top of two basic Python libraries, viz., NumPy and SciPy. Scikit-learn supports most of the supervised and unsupervised learning algorithms. Scikit-learn can also be used for data-mining and data-analysis, which makes it a great tool who is starting out with ML.

TensorFlow is a very popular open-source library for high performance numerical computation developed by the Google Brain team in Google. As the name suggests, Tensorflow is a framework that involves defining and running computations involving tensors. It can train and run deep neural networks that can be used to develop several AI applications. TensorFlow is widely used in the field of deep learning research and application.

Keras is a very popular Machine Learning library for Python. It is a high-level neural networks API capable of running on top of TensorFlow, CNTK, or Theano. It can run seamlessly on both CPU and GPU. Keras makes it really for ML beginners to build and design a Neural Network. One of the best thing about Keras is that it allows for easy and fast prototyping.

PyTorch is a popular open-source Machine Learning library for Python based on Torch, which is an open-source Machine Learning library that is implemented in C with a wrapper in Lua. It has an extensive choice of tools and libraries that support Computer Vision, Natural Language Processing(NLP), and many more ML programs. It allows developers to perform computations on Tensors with GPU acceleration and also helps in creating computational graphs.

Pandas is a popular Python library for data analysis. It is not directly related to Machine Learning. As we know that the dataset must be prepared before training. In this case, Pandas comes handy as it was developed specifically for data extraction and preparation. It provides high-level data structures and wide variety tools for data analysis. It provides many inbuilt methods for grouping, combining and filtering data.

Matplotlib is a very popular Python library for data visualization. Like Pandas, it is not directly related to Machine Learning. It particularly comes in handy when a programmer wants to visualize the patterns in the data. It is a 2D plotting library used for creating 2D graphs and plots. A module named pyplot makes it easy for programmers for plotting as it provides features to control line styles, font properties, formatting axes, etc. It provides various kinds of graphs and plots for data visualization, viz., histogram, error charts, bar chats, etc,

Machine learning encompasses various algorithms, including supervised learning (using labeled data for prediction) and unsupervised learning (finding patterns in unlabeled data). These techniques enable tasks like classification, regression, clustering, and anomaly detection.

There are several types of machine learning algorithms that are commonly used in the field. Supervised learning is one type where the algorithm is trained using labeled examples, meaning the input data is paired with the corresponding desired output. The algorithm learns to map inputs to outputs by finding patterns in the labeled data. This type of learning is often used for tasks like classification, regression, and prediction.

Another type is unsupervised learning, where the algorithm is given unlabeled data and is tasked with finding patterns or structures within the data. Unlike supervised learning, there are no predefined outputs to learn from. Unsupervised learning algorithms focus on discovering inherent relationships, clustering similar data points, or reducing the dimensionality of the data. This type of learning is useful for tasks such as anomaly detection, data exploration, and recommendation systems.

Module 1.4: import Library	Duration =3	Type code	Category: Introduction

To utilize popular machine learning packages in Python, import scikit-learn (sklearn), Matplotlib, pandas, NumPy, and Seaborn. These libraries provide essential tools for data analysis, visualization, and machine learning algorithms.

To import popular machine learning packages such as scikit-learn (sklearn), Matplotlib, pandas, NumPy, and Seaborn, you can use the following code:

```
# importing libraries
import sklearn
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import seaborn as sns
```

The above code imports the packages and assigns them convenient aliases (sklearn, plt, pd, np, sns), which can be used to access the functionality provided by

each package. These packages offer a wide range of tools and functions for data manipulation, visualization, statistical analysis, and machine learning algorithms.

Module 2.1: introduction to data	Duration 2	Type concept	Category: data
DATA: It can be any unprocessed fact, value, text, sound, or picture that is not being interpreted and analyzed. Data is the most important part of all Data Analytics, Machine Learning, Artificial Intelligence. Without data, we can't train any model and all modern research and automation will go in vain. Big Enterprises are spending lots of money just to gather as much certain data as possible. **How we split data in Machine Learning?** **Training Data: The part of data we use to train our model. This is the data that your model actually sees(both input and output) and learns from.** **Validation Data: The part of data that is used to do a frequent evaluation of the model, fit on the training dataset along with improving involved hyperparameters (initially set parameters before the model begins learning). This data plays its part when the model is actually training.** **Testing Data: Once our model is completely trained, testing data provides an unbiased evaluation. When we feed in the inputs of Testing data, our model will predict some values(without seeing actual output). After prediction, we evaluate our model by comparing it with the actual output present in the testing data. This is how we evaluate and see how much our model has learned from the experiences feed in as training data, set at the time of training.**			

Module 2.2: data preprocessing	Duration 7	Type code	Category: data
Pre-processing refers to the transformations applied to our data before feeding it to the algorithm. Data Preprocessing is a technique that is used to convert the raw data into a clean data set. In other words, whenever the data is gathered from different sources it is collected in raw format which is not feasible for the analysis. **This article contains 3 different data preprocessing techniques for machine learning.**			

1. Rescale Data

When our data is comprised of attributes with varying scales, many machine learning algorithms can benefit from rescaling the attributes to all have the same scale.
This is useful for optimization algorithms used in the core of machine learning algorithms like gradient descent.
It is also useful for algorithms that weight inputs like regression and neural networks and algorithms that use distance measures like K-Nearest Neighbors.
We can rescale your data using scikit-learn using the MinMaxScaler class.

```
# importing libraries
import pandas
import scipy
import numpy
from sklearn.preprocessing import MinMaxScaler

# data set link
url =
"https://archive.ics.uci.edu/ml/machine-learning-databases/pima-indians-diabetes/pima-indians-diabetes.data"
# data parameters
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass',
'pedi', 'age', 'class']

# preparating of dataframe using the data at given link and
defined columns list
dataframe = pandas.read_csv(url, names = names)
array = dataframe.values

# separate array into input and output components
X = array[:,0:8]
Y = array[:,8]

# initialising the MinMaxScaler
scaler = MinMaxScaler(feature_range=(0, 1))
# learning the statistical parameters for each of the data and
transforming
rescaledX = scaler.fit_transform(X)

# summarize transformed data
numpy.set_printoptions(precision=3)
print(rescaledX[0:5,:])
```

After rescaling see that all of the values are in the range between 0 and 1.

2. Binarize Data (Make Binary)

We can transform our data using a binary threshold. All values above the threshold are marked 1 and all equal to or below are marked as 0.
This is called binarizing your data or threshold your data. It can be useful when you have probabilities that you want to make crisp values. It is also useful when feature engineering and you want to add new features that indicate something meaningful.
We can create new binary attributes in Python using scikit-learn with the Binarizer class.

```
# import libraries
from sklearn.preprocessing import Binarizer
import pandas
import numpy

# data set link
url =
"https://archive.ics.uci.edu/ml/machine-learning-databases/pima-indians-diabetes/pima-indians-diabetes.data"
# data parameters
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass',
'pedi', 'age', 'class']

# preparating of dataframe using the data at given link and
defined columns list
dataframe = pandas.read_csv(url, names = names)
array = dataframe.values

# separate array into input and output components
X = array[:, 0:8]
Y = array[:, 8]
binarizer = Binarizer(threshold = 0.0).fit(X)
binaryX = binarizer.transform(X)

# summarize transformed data
numpy.set_printoptions(precision = 3)
print(binaryX[0:5,:])
```

We can see that all values equal or less than 0 are marked 0 and all of those above 0 are marked 1.

3. Standardize Data

Standardization is a useful technique to transform attributes with a Gaussian distribution and differing means and standard deviations to a standard Gaussian distribution with a mean of 0 and a standard deviation of 1.
We can standardize data using scikit-learn with the StandardScaler class.
Code: Python code to Standardize data (0 mean, 1 stdev)

```
# importing libraries
from sklearn.preprocessing import StandardScaler
import pandas
import numpy

# data set link
url =
"https://archive.ics.uci.edu/ml/machine-learning-databases/pima-indians-diabetes/pima-indians-diabetes.data"
# data parameters
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass',
'pedi', 'age', 'class']

# preparating of dataframe using the data at given link and
defined columns list
dataframe = pandas.read_csv(url, names = names)
array = dataframe.values

# separate array into input and output components
X = array[:, 0:8]
Y = array[:, 8]
scaler = StandardScaler().fit(X)
rescaledX = scaler.transform(X)

# summarize transformed data
numpy.set_printoptions(precision = 3)
print(rescaledX[0:5,:])
```

The values for each attribute now have a mean value of 0 and a standard deviation of 1.

Module 2.3: Cleaning data	Duration 2	Type concept	Category: data

Data cleaning is a crucial step in machine learning to ensure reliable and accurate results. It involves removing duplicate and irrelevant observations, fixing structural errors, managing outliers, and handling missing data. Proper data cleaning improves the efficiency and quality of models. Various tools like Openrefine and Trifacta Wrangler facilitate the data cleaning process.

Data cleaning is one of the important parts of machine learning. It plays a significant part in building a model. It surely isn't the fanciest part of machine learning and at the same time, there aren't any hidden tricks or secrets to uncover. However, the success or failure of a project relies on proper data cleaning. Professional data scientists usually invest a very large portion of their time in this step because of the belief that "Better data beats fancier algorithms".
If we have a well-cleaned dataset, there are chances that we can get achieve good results with simple algorithms also, which can prove very beneficial at times especially in terms of computation when the dataset size is large.

Obviously, different types of data will require different types of cleaning. However, this systematic approach can always serve as a good starting point.

1. **Removal of unwanted observations**

This includes deleting duplicate/ redundant or irrelevant values from your dataset. Duplicate observations most frequently arise during data collection and Irrelevant observations are those that don't actually fit the specific problem that you're trying to solve.
Redundant observations alter the efficiency by a great extent as the data repeats and may add towards the correct side or towards the incorrect side, thereby producing unfaithful results.
Irrelevant observations are any type of data that is of no use to us and can be removed directly.

2. **Fixing Structural errors**

The errors that arise during measurement, transfer of data, or other similar situations are called structural errors. Structural errors include typos in the name of features, the same attribute with a different name, mislabeled classes, i.e. separate classes that should really be the same, or inconsistent capitalization.
For example, the model will treat America and America as different classes or values, though they represent the same value or red, yellow, and red-yellow as

different classes or attributes, though one class can be included in the other two classes. So, these are some structural errors that make our model inefficient and give poor quality results.

3. **Managing Unwanted outliers**

Outliers can cause problems with certain types of models. For example, linear regression models are less robust to outliers than decision tree models. Generally, we should not remove outliers until we have a legitimate reason to remove them. Sometimes, removing them improves performance, sometimes not. So, one must have a good reason to remove the outlier, such as suspicious measurements that are unlikely to be part of real data.

4. **Handling missing data**

Missing data is a deceptively tricky issue in machine learning. We cannot just ignore or remove the missing observation. They must be handled carefully as they can be an indication of something important. The two most common ways to deal with missing data are:

Dropping observations with missing values.

The fact that the value was missing may be informative in itself.

Plus, in the real world, you often need to make predictions on new data even if some of the features are missing!

Imputing the missing values from past observations.

Again, "missingness" is almost always informative in itself, and you should tell your algorithm if a value was missing.

Even if you build a model to impute your values, you're not adding any real information. You're just reinforcing the patterns already provided by other features.

Missing data is like missing a puzzle piece. If you drop it, that's like pretending the puzzle slot isn't there. If you impute it, that's like trying to squeeze in a piece from somewhere else in the puzzle.

So, missing data is always an informative and an indication of something important. And we must be aware of our algorithm of missing data by flagging it. By using this technique of flagging and filling, you are essentially allowing the algorithm to estimate the optimal constant for missingness, instead of just filling it in with the mean.

Some data cleansing tools

Openrefine
Trifacta Wrangler
TIBCO Clarity
Cloudingo
IBM Infosphere Quality Stage

Conclusion:

So, we have discussed four different steps in data cleaning to make the data more reliable and to produce good results. After properly completing the Data Cleaning steps, we'll have a robust dataset that avoids many of the most common pitfalls. This step should not be rushed as it proves very beneficial in

the further process.

Module 2.4: Feature scaling	Duration 5	Type code	Category: data
Feature scaling is a crucial step in data preprocessing to ensure fair treatment of independent features in machine learning. It standardizes the values to a fixed range, preventing algorithms from favoring certain features due to varying magnitudes or units. Min-Max Normalization and Standardization are popular techniques used for feature scaling.			

Feature Scaling is a technique to standardize the independent features present in the data in a fixed range. It is performed during the data pre-processing to handle highly varying magnitudes or values or units. If feature scaling is not done, then a machine learning algorithm tends to weigh greater values, higher and consider smaller values as the lower values, regardless of the unit of the values.

Example: If an algorithm is not using the feature scaling method then it can consider the value 3000 meters to be greater than 5 km but that's actually not true and in this case, the algorithm will give wrong predictions. So, we use Feature Scaling to bring all values to the same magnitudes and thus, tackle this issue.

Techniques to perform Feature Scaling
Consider the two most important ones:

Min-Max Normalization: This technique re-scales a feature or observation value with distribution value between 0 and 1.
X_{\text {new }}=\frac{X_{i}-\min (X)}{\max (x)-\min (X)}
Standardization: It is a very effective technique which re-scales a feature value so that it has distribution with 0 mean value and variance equals to 1.
X_{\text {new }}=\frac{X_{i}-X_{\text {mean }}}{\text { Standard Deviation }}
Download the dataset:
Go to the link and download Data_for_Feature_Scaling.csv

```
# Python code explaining How to
# perform Feature Scaling

""" PART 1
    Importing Libraries """

import numpy as np
```

```
import matplotlib.pyplot as plt
import pandas as pd

# Sklearn library
from sklearn import preprocessing

""" PART 2
    Importing Data """

data_set =
pd.read_csv('C:\\Users\\dell\\Desktop\\Data_for_Feature_Scaling
.csv')
data_set.head()

# here Features - Age and Salary columns
# are taken using slicing
# to handle values with varying magnitude
x = data_set.iloc[:, 1:3].values
print ("\nOriginal data values : \n",  x)


""" PART 4
    Handling the missing values """

from sklearn import preprocessing

""" MIN MAX SCALER """

min_max_scaler = preprocessing.MinMaxScaler(feature_range =(0,
1))

# Scaled feature
x_after_min_max_scaler = min_max_scaler.fit_transform(x)

print ("\nAfter min max Scaling : \n", x_after_min_max_scaler)


""" Standardisation """

Standardisation = preprocessing.StandardScaler()

# Scaled feature
x_after_Standardisation = Standardisation.fit_transform(x)

print ("\nAfter Standardisation : \n", x_after_Standardisation)
```

Module 2.5: Handle missing data	Duration 12	Type code	Category: data

Imbalanced data distribution is a common challenge in machine learning, where the class distribution is significantly skewed. Standard algorithms often favor the majority class, leading to poor performance on the minority class. Two popular techniques for handling imbalanced data are SMOTE (Synthetic Minority Oversampling Technique) for oversampling and NearMiss Algorithm for undersampling. This article provides insights and practical examples on how to choose the best technique for addressing imbalanced data.

In Machine Learning and Data Science we often come across a term called Imbalanced Data Distribution, generally happens when observations in one of the class are much higher or lower than the other classes. As Machine Learning algorithms tend to increase accuracy by reducing the error, they do not consider the class distribution. This problem is prevalent in examples such as Fraud Detection, Anomaly Detection, Facial recognition etc.

Standard ML techniques such as Decision Tree and Logistic Regression have a bias towards the majority class, and they tend to ignore the minority class. They tend only to predict the majority class, hence, having major misclassification of the minority class in comparison with the majority class. In more technical words, if we have imbalanced data distribution in our dataset then our model becomes more prone to the case when minority class has negligible or very lesser recall.

Imbalanced Data Handling Techniques: There are mainly 2 mainly algorithms that are widely used for handling imbalanced class distribution.

SMOTE
Near Miss Algorithm

SMOTE (Synthetic Minority Oversampling Technique) – Oversampling
SMOTE (synthetic minority oversampling technique) is one of the most commonly used oversampling methods to solve the imbalance problem.
It aims to balance class distribution by randomly increasing minority class examples by replicating them.
SMOTE synthesises new minority instances between existing minority instances. It generates the virtual training records by linear interpolation for the minority class. These synthetic training records are generated by randomly selecting one or more of the k-nearest neighbors for each example in the minority class. After the oversampling process, the data is reconstructed and

several classification models can be applied for the processed data.
More Deep Insights of how SMOTE Algorithm work !

Step 1: Setting the minority class set A, for each $x \in A$, the k-nearest neighbors of x are obtained by calculating the Euclidean distance between x and every other sample in set A.
Step 2: The sampling rate N is set according to the imbalanced proportion. For each $x \in A$, N examples (i.e x1, x2, …xn) are randomly selected from its k-nearest neighbors, and they construct the set A_1 .
Step 3: For each example $x_k \in A_1$ (k=1, 2, 3…N), the following formula is used to generate a new example:
$x' = x + rand(0, 1) * \mid x - x_k \mid$
in which rand(0, 1) represents the random number between 0 and 1.

NearMiss Algorithm – Undersampling
NearMiss is an under-sampling technique. It aims to balance class distribution by randomly eliminating majority class examples. When instances of two different classes are very close to each other, we remove the instances of the majority class to increase the spaces between the two classes. This helps in the classification process.
To prevent problem of information loss in most under-sampling techniques, near-neighbor methods are widely used.
The basic intuition about the working of near-neighbor methods is as follows:

Step 1: The method first finds the distances between all instances of the majority class and the instances of the minority class. Here, majority class is to be under-sampled.
Step 2: Then, n instances of the majority class that have the smallest distances to those in the minority class are selected.
Step 3: If there are k instances in the minority class, the nearest method will result in k*n instances of the majority class.
For finding n closest instances in the majority class, there are several variations of applying NearMiss Algorithm :

NearMiss – Version 1 : It selects samples of the majority class for which average distances to the k closest instances of the minority class is smallest.
NearMiss – Version 2 : It selects samples of the majority class for which average distances to the k farthest instances of the minority class is smallest.
NearMiss – Version 3 : It works in 2 steps. Firstly, for each minority class instance, their M nearest-neighbors will be stored. Then finally, the majority class instances are selected for which the average distance to the N nearest-neighbors is the largest.
This article helps in better understanding and hands-on practice on how to choose best between different imbalanced data handling techniques.

Load libraries and data file
The dataset consists of transactions made by credit cards. This dataset has 492 fraud transactions out of 284, 807 transactions. That makes it highly unbalanced, the positive class (frauds) account for 0.172% of all transactions. The dataset can be downloaded from here.

```
# import necessary modules
import pandas  as pd
import matplotlib.pyplot as plt
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix,
classification_report

# load the data set
data = pd.read_csv('creditcard.csv')

# print info about columns in the dataframe
print(data.info())
```

```
# normalise the amount column
data['normAmount'] =
StandardScaler().fit_transform(np.array(data['Amount']).reshape
(-1, 1))

# drop Time and Amount columns as they are not relevant for
prediction purpose
data = data.drop(['Time', 'Amount'], axis = 1)

# as you can see there are 492 fraud transactions.
data['Class'].value_counts()
```

Split the data into test and train sets

```
from sklearn.model_selection import train_test_split

# split into 70:30 ration
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size = 0.3, random_state = 0)

# describes info about train and test set
print("Number transactions X_train dataset: ", X_train.shape)
```

```
print("Number transactions y_train dataset: ", y_train.shape)
print("Number transactions X_test dataset: ", X_test.shape)
print("Number transactions y_test dataset: ", y_test.shape)
```

Now train the model without handling the imbalanced class distribution

```
# logistic regression object
lr = LogisticRegression()

# train the model on train set
lr.fit(X_train, y_train.ravel())

predictions = lr.predict(X_test)

# print classification report
print(classification_report(y_test, predictions))
```

The accuracy comes out to be 100% but did you notice something strange ? The recall of the minority class in very less. It proves that the model is more biased towards majority class. So, it proves that this is not the best model. Now, we will apply different imbalanced data handling techniques and see their accuracy and recall results.

Using SMOTE Algorithm

```
print("Before OverSampling, counts of label '1':
{}".format(sum(y_train == 1)))
print("Before OverSampling, counts of label '0': {}
\n".format(sum(y_train == 0)))

# import SMOTE module from imblearn library
# pip install imblearn (if you don't have imblearn in your
system)
from imblearn.over_sampling import SMOTE
sm = SMOTE(random_state = 2)
X_train_res, y_train_res = sm.fit_sample(X_train,
y_train.ravel())

print('After OverSampling, the shape of train_X:
{}'.format(X_train_res.shape))
print('After OverSampling, the shape of train_y: {}
\n'.format(y_train_res.shape))

print("After OverSampling, counts of label '1':
{}".format(sum(y_train_res == 1)))
```

```
print("After OverSampling, counts of label '0':
{}".format(sum(y_train_res == 0)))
```

Look! that SMOTE Algorithm has oversampled the minority instances and made it equal to majority class. Both categories have equal amount of records. More specifically, the minority class has been increased to the total number of majority class.
Now see the accuracy and recall results after applying SMOTE algorithm (Oversampling).

Prediction and Recall

```
lr1 = LogisticRegression()
lr1.fit(X_train_res, y_train_res.ravel())
predictions = lr1.predict(X_test)

# print classification report
print(classification_report(y_test, predictions))
```

Wow, We have reduced the accuracy to 98% as compared to previous model but the recall value of minority class has also improved to 92 %. This is a good model compared to the previous one. Recall is great.
Now, we will apply NearMiss technique to Under-sample the majority class and see its accuracy and recall results.

NearMiss Algorithm:

```
print("Before Undersampling, counts of label '1':
{}".format(sum(y_train == 1)))
print("Before Undersampling, counts of label '0': {}
\n".format(sum(y_train == 0)))

# apply near miss
from imblearn.under_sampling import NearMiss
nr = NearMiss()

X_train_miss, y_train_miss = nr.fit_sample(X_train,
y_train.ravel())

print('After Undersampling, the shape of train_X:
{}'.format(X_train_miss.shape))
print('After Undersampling, the shape of train_y: {}
\n'.format(y_train_miss.shape))

print("After Undersampling, counts of label '1':
{}".format(sum(y_train_miss == 1)))
print("After Undersampling, counts of label '0':
```

```
{}".format(sum(y_train_miss == 0)))
```

The NearMiss Algorithm has undersampled the majority instances and made it equal to majority class. Here, the majority class has been reduced to the total number of minority class, so that both classes will have equal number of records.

Prediction and Recall

```
# train the model on train set
lr2 = LogisticRegression()
lr2.fit(X_train_miss, y_train_miss.ravel())
predictions = lr2.predict(X_test)

# print classification report
print(classification_report(y_test, predictions))
```

This model is better than the first model because it classifies better and also the recall value of minority class is 95 %. But due to undersampling of majority class, its recall has decreased to 56 %.

Module 2.6: train-test split	Duration 2	Type code	Category: data

The train-test split is a crucial step in machine learning to evaluate model performance. This technique divides the dataset into training and testing sets. Python's scikit-learn library provides a convenient train_test_split function for this purpose.

When building a machine learning model, it is crucial to evaluate its performance on unseen data to ensure its generalization capabilities. The train-test split is a technique used to divide the available dataset into two subsets: the training set and the test set. The training set is used to train the model, while the test set is used to evaluate its performance.

Code example in Python using scikit-learn:

```
from sklearn.model_selection import train_test_split

# X: input features, y: target variable
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# train_test_split function takes X and y as inputs along with
the test_size parameter,
# which determines the proportion of the dataset to be
allocated to the test set (here, 20%).
# random_state ensures reproducibility by fixing the random
seed for the split.
```

In the code snippet, X represents the input features, and y represents the target variable or labels. The train_test_split function from scikit-learn is used to split the data into training and testing sets. The resulting split is assigned to X_train, X_test, y_train, and y_test variables.

Module 2.7: Categorical data	Duration 4	Type code	Category: data

In feature engineering, converting categorical variables to numerical values is crucial for ML. One-hot encoding creates binary columns for each category, while label encoding assigns numerical labels. The choice depends on data and algorithms. Code examples demonstrate one-hot encoding using pandas and label encoding using scikit-learn.

Conversion from categorical to numerical values is an important step in feature engineering for machine learning. Categorical variables represent qualitative data that cannot be directly used by ML algorithms. One common approach is one-hot encoding, where each category is converted into a binary column. Another approach is label encoding, where categories are mapped to numerical values. The choice depends on the nature of the data and the algorithm used.
Code for one-hot encoding:

```
import pandas as pd
data = pd.DataFrame({'Color': ['Red', 'Blue', 'Green', 'Red',
'Blue']})
encoded_data = pd.get_dummies(data['Color'])
print(encoded_data)
```

Code for label encoding:

```
from sklearn.preprocessing import LabelEncoder
data = pd.DataFrame({'Size': ['Small', 'Medium', 'Large', 'Medium']})
label_encoder = LabelEncoder()
encoded_data = label_encoder.fit_transform(data['Size'])
print(encoded_data)
```

Module 2.8: imbalance data	Duration 3	Type code	Category: data

Addressing imbalanced data is crucial in machine learning. Undersampling and oversampling techniques help to mitigate the issue. SMOTE is a popular oversampling method that generates synthetic samples. The provided code demonstrates the usage of SMOTE and RandomUnderSampler from the imbalanced-learn library to perform oversampling and undersampling, respectively.

Different sampling methods are commonly used to address imbalanced data in machine learning. Two widely used techniques are undersampling and oversampling. Undersampling reduces the majority class samples, while oversampling increases the minority class samples. One popular oversampling method is SMOTE (Synthetic Minority Over-sampling Technique), which generates synthetic samples based on the feature space to balance the classes.

Here is an example code using the imbalanced-learn library in Python:

```
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
from imblearn.pipeline import make_pipeline

# Create an imbalanced dataset X and y

# Undersampling using RandomUnderSampler
rus = RandomUnderSampler()
X_resampled, y_resampled = rus.fit_resample(X, y)

# Oversampling using SMOTE
smote = SMOTE()
X_resampled, y_resampled = smote.fit_resample(X, y)
```

Module 3.1: Introduction to supervised learning	Duration 2	Type concept	Category: Supervised learning

Machine learning involves improving a machine's performance in a task based on its past experiences. In the context of predicting customer behavior, a machine learns from previous data to predict whether a customer will purchase a specific product. The training process involves splitting the data into training and testing sets, where the model learns from the training data and is evaluated on the unseen testing data. Supervised learning includes classification, where discrete labels are predicted, and regression, where continuous values are predicted. The accuracy of the model is assessed based on its ability to predict the correct labels or values.

machine is said to be learning from past Experiences(data feed-in) with respect to some class of tasks if its Performance in a given Task improves with the Experience. For example, assume that a machine has to predict whether a customer will buy a specific product let's say "Antivirus" this year or not. The machine will do it by looking at the previous knowledge/past experiences i.e the data of products that the customer had bought every year and if he buys Antivirus every year, then there is a high probability that the customer is going to buy an antivirus this year as well. This is how machine learning works at the basic conceptual level.

Training the system: While training the model, data is usually split in the ratio of 80:20 i.e. 80% as training data and the rest as testing data. In training data, we feed input as well as output for 80% of data. The model learns from training data only. We use different machine learning algorithms(which we will discuss in detail in the next articles) to build our model. Learning means that the model will build some logic of its own.
Once the model is ready then it is good to be tested. At the time of testing, the input is fed from the remaining 20% of data that the model has never seen before, the model will predict some value and we will compare it with the actual output and calculate the accuracy.

Types of Supervised Learning:
A. Classification: It is a Supervised Learning task where output is having defined labels(discrete value). For example in above Figure A, Output – Purchased has defined labels i.e. 0 or 1; 1 means the customer will purchase, and 0 means that the customer won't purchase. The goal here is to predict discrete values belonging to a particular class and evaluate them on the basis of accuracy.
It can be either binary or multi-class classification. In binary classification, the model predicts either 0 or 1; yes or no but in the case of multi-class classification, the model predicts more than one class. Example: Gmail classifies mails in more than one class like social, promotions, updates, and forums.

B. Regression: It is a Supervised Learning task where output is having continuous value.
For example in above Figure B, Output – Wind Speed is not having any discrete value but is continuous in a particular range. The goal here is to predict a value as much closer to the actual output value as our model can and then evaluation is done by calculating the error value. The smaller the error the greater the accuracy of our regression model.

Module 3.2: classification	Duration 7	Type concept	Category: Supervised learning

Classification is the task of categorizing data into sub-categories, performed by machines in machine learning and statistics. It involves identifying the category to which a new observation belongs based on a training set. There are two types of classification: binary classification and multiclass classification. Various classifiers, such as logistic regression, decision tree classifiers, and support vector machines, are used for classification tasks. These techniques find applications in self-driving cars, spam email filtering, health problem detection, facial recognition, and more. Classifiers can be categorized as discriminative or generative, depending on their modeling approach.

As the name suggests, Classification is the task of "classifying things" into sub-categories. But, by a machine! If that doesn't sound like much, imagine your computer being able to differentiate between you and a stranger. Between a potato and a tomato. Between an A grade and an F. Now, it sounds interesting now. In Machine Learning and Statistics, Classification is the problem of identifying to which of a set of categories (subpopulations), a new observation belongs, on the basis of a training set of data containing observations and whose categories membership is known.
Types of Classification
Classification is of two types:

Binary Classification: When we have to categorize given data into 2 distinct classes. Example – On the basis of given health conditions of a person, we have to determine whether the person has a certain disease or not.
Multiclass Classification: The number of classes is more than 2. For Example – On the basis of data about different species of flowers, we have to determine which specie our observation belongs.

Generalized Classification Block Diagram.

X: pre-classified data, in the form of an N*M matrix.
N is the no. of observations and M is the number of features
y: An N-d vector corresponding to predicted classes for each of the N observations.
Feature Extraction: Extracting valuable information from input X using a series of transforms.
ML Model: The "Classifier" we'll train.
y': Labels predicted by the Classifier.
Quality Metric: Metric used for measuring the performance of the model.
ML Algorithm: The algorithm that is used to update weights w', which updates

the model and “learns” iteratively.

types of Classifiers (algorithms)

There are various types of classifiers. Some of them are :

Linear Classifiers: Logistic Regression
Tree-Based Classifiers: Decision Tree Classifier
Support Vector Machines
Artificial Neural Networks
Bayesian Regression
Gaussian Naive Bayes Classifiers
Stochastic Gradient Descent (SGD) Classifier
Ensemble Methods: Random Forests, AdaBoost, Bagging Classifier, Voting Classifier, ExtraTrees Classifier

Practical Applications of Classification

Google’s self-driving car uses deep learning-enabled classification techniques which enables it to detect and classify obstacles.
Spam E-mail filtering is one of the most widespread and well-recognized uses of Classification techniques.
Detecting Health Problems, Facial Recognition, Speech Recognition, Object Detection, and Sentiment Analysis all use Classification at their core.

classifiers can be categorized into two major types:

Discriminative: It is a very basic classifier and determines just one class for each row of data. It tries to model just by depending on the observed data, depends heavily on the quality of data rather than on distributions.
Example: Logistic Regression
Generative: It models the distribution of individual classes and tries to learn the model that generates the data behind the scenes by estimating assumptions and distributions of the model. Used to predict the unseen data.
Example: Naive Bayes Classifier
Detecting Spam emails by looking at the previous data. Suppose 100 emails and that too divided in 1:4 i.e. Class A: 25%(Spam emails) and Class B: 75%(Non-Spam emails). Now if a user wants to check that if an email contains the word cheap, then that may be termed as Spam.
It seems to be that in Class A(i.e. in 25% of data), 20 out of 25 emails are spam and rest not.
And in Class B(i.e. in 75% of data), 70 out of 75 emails are not spam and rest are spam.
So, if the email contains the word cheap, what is the probability of it being spam ?? (= 80%)

Module 3.3: Classification using sklearn	Duration 5	Type code	Category: Supervised learning

Multiclass classification involves training a classifier using a dataset consisting of training examples with features and corresponding labels. The dataset is split into training and test data, and classifiers like decision trees, SVM, and KNN are trained on the training data. These classifiers are then used to predict labels for the test data. Accuracy is measured, and classification results are visualized. The decision tree classifier poses questions to the dataset, creating a binary tree structure to classify the data.

Given a dataset of m training examples, each of which contains information in the form of various features and a label. Each label corresponds to a class, to which the training example belongs. In multiclass classification, we have a finite set of classes. Each training example also has n features.

In a multiclass classification, we train a classifier using our training data and use this classifier for classifying new examples.

Approach –

1. **Load dataset from the source.**
2. **Split the dataset into "training" and "test" data.**
3. **Train Decision tree, SVM, and KNN classifiers on the training data.**
4. **Use the above classifiers to predict labels for the test data.**
5. **Measure accuracy and visualize classification.**

Decision tree classifier – A decision tree classifier is a systematic approach for multiclass classification. It poses a set of questions to the dataset (related to its attributes/features). The decision tree classification algorithm can be visualized on a binary tree. On the root and each of the internal nodes, a question is posed and the data on that node is further split into separate records that have different characteristics. The leaves of the tree refer to the classes in which the dataset is split. In the following code snippet, we train a decision tree classifier in scikit-learn.

```
# importing necessary libraries
from sklearn import datasets
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split

# loading the iris dataset
iris = datasets.load_iris()
```

```
# X -> features, y -> label
X = iris.data
y = iris.target

# dividing X, y into train and test data
X_train, X_test, y_train, y_test = train_test_split(X, y,
random_state = 0)

# training a DescisionTreeClassifier
from sklearn.tree import DecisionTreeClassifier
dtree_model = DecisionTreeClassifier(max_depth = 2).fit(X_train,
y_train)
dtree_predictions = dtree_model.predict(X_test)

# creating a confusion matrix
cm = confusion_matrix(y_test, dtree_predictions)
```

Module 3.4: cross validation	Duration 5	Type code	Category: Supervised learning

Cross-validation methods such as k-fold and stratified k-fold are essential for evaluating machine learning models. K-fold divides the data into k subsets, training and testing the model on each fold. Stratified k-fold ensures class distribution balance. These methods help assess model performance on unseen data.

Cross-validation is a widely used technique in machine learning to evaluate the performance of a model on unseen data. There are various types of cross-validation methods, including k-fold cross-validation and stratified k-fold cross-validation. Here's an example code for both methods:

K-Fold Cross-Validation:

```
from sklearn.model_selection import KFold

# Splitting data into k folds
k = 5
kf = KFold(n_splits=k)

for train_index, test_index in kf.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
```

```
    # Train and evaluate the model on each fold
    model.fit(X_train, y_train)
    accuracy = model.score(X_test, y_test)
    print("Accuracy:", accuracy)
```

Stratified K-Fold Cross-Validation:

```
from sklearn.model_selection import StratifiedKFold

# Splitting data into k stratified folds
k = 5
skf = StratifiedKFold(n_splits=k)

for train_index, test_index in skf.split(X, y):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    # Train and evaluate the model on each fold
    model.fit(X_train, y_train)
    accuracy = model.score(X_test, y_test)
    print("Accuracy:", accuracy)
```

Module 3.5: evaluation metrics	Duration 3	Type concept	Category: Supervised learning

When evaluating a classification model in machine learning, metrics such as accuracy, precision, recall, and F1 score play a crucial role. These measurements assess the correctness, completeness, and overall performance of the model. Additionally, the confusion matrix provides a detailed breakdown of predictions. A solid grasp of these concepts is vital for effectively evaluating and improving classification models.

In machine learning, classification measurements are used to evaluate the performance of a classification model. Here are some commonly used

measurements:

- **Accuracy: It measures the overall correctness of the model by calculating the ratio of correctly predicted instances to the total number of instances.**
- **Precision: It represents the proportion of correctly predicted positive instances out of the total instances predicted as positive. It focuses on the correctness of positive predictions.**
- **Recall: It calculates the proportion of correctly predicted positive instances out of the total actual positive instances. It emphasizes the completeness of positive predictions.**
- **F1 Score: It is the harmonic mean of precision and recall. It provides a balanced measure between precision and recall, considering both correctness and completeness.**
- **Confusion Matrix: It is a tabular representation that shows the counts of true positive, true negative, false positive, and false negative predictions. It provides a detailed breakdown of the model's performance.**

Understanding these measurements is essential for evaluating the effectiveness of a classification model and identifying its strengths and weaknesses.

Module 3.6: AUC-ROC	Duration 4	Type code	Category: Supervised learning

AUC-ROC plot is a valuable tool for evaluating binary classifiers in ML. It visualizes the trade-off between true positive rate and false positive rate. The higher the AUC-ROC score, the better the classifier performance. The provided code demonstrates how to generate and plot the AUC-ROC curve using Python's scikit-learn library.

The AUC-ROC (Area Under the Receiver Operating Characteristic Curve) plot is a popular evaluation metric for classification tasks in machine learning. It measures the performance of a binary classifier at various classification thresholds. The curve plots the true positive rate (sensitivity) against the false positive rate (1 - specificity) for different threshold values. A higher AUC-ROC score indicates a better classifier performance.

Code for AUC-ROC plot:

```
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc
```

```
# Assuming y_true contains the true labels and y_scores contains the
predicted scores or probabilities
fpr, tpr, thresholds = roc_curve(y_true, y_scores)
roc_auc = auc(fpr, tpr)

# Plotting the ROC curve
plt.plot(fpr, tpr, label='AUC = %0.2f' % roc_auc)
plt.plot([0, 1], [0, 1], 'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC)')
plt.legend(loc='lower right')
plt.show()
```

Module 4.1: Optimization	Duration 3	Type concept	Category: optimization

Optimization techniques are fundamental in machine learning for finding optimal model parameters. Gradient Descent, Stochastic Gradient Descent, Adam, and RMSprop are commonly used algorithms. This description introduces these optimization methods, which are crucial for training accurate machine learning models.

Optimization techniques are essential in machine learning to find the optimal set of parameters that minimize the loss function and improve the model's performance. Here are some commonly used optimization algorithms:

Gradient Descent (GD): It is a first-order optimization algorithm that iteratively updates the model parameters in the direction of the negative gradient of the loss function. GD calculates the gradient using the entire training dataset, making it computationally expensive for large datasets.

Stochastic Gradient Descent (SGD): In this variant of GD, instead of using the entire training dataset, SGD randomly selects a single training sample or a small batch of samples to compute the gradient and update the parameters. This approach introduces more randomness, but it reduces the computational cost and can converge faster. However, SGD may exhibit more oscillations and noise during training.

Mini-Batch Gradient Descent: It is a compromise between GD and SGD, where the gradient is computed using a small batch of training samples. This approach provides a balance between computational efficiency and stability compared to SGD.

Adam: It is an adaptive optimization algorithm that combines the advantages of both Momentum and RMSprop methods. Adam adapts the learning rate for each parameter based on the first and second moments of the gradients. It performs well in a wide range of scenarios and is widely used in deep learning models.

RMSprop: This algorithm also adapts the learning rate but maintains an exponentially weighted average of the squared gradients. By adjusting the learning rate based on the magnitude of recent gradients, RMSprop can handle sparse gradients and converge faster in some cases.

These optimization techniques differ in terms of computational efficiency, convergence speed, and resilience to different data distributions. It is important to experiment and choose the most suitable optimization algorithm for a specific machine learning task based on factors such as dataset size, model complexity, and convergence requirements.

Module 4.2: Gradient descent	Duration 3	Type concept	Category: optimization

Gradient Descent is an optimization algorithm used for minimizing the cost function in machine learning. This article explains the different types of Gradient Descent, including Batch, Stochastic, and Mini-Batch, along with their convergence trends. The impact of the learning rate and the number of training examples on convergence is also discussed.

Gradient Descent is an optimization algorithm used for minimizing the cost function in various machine learning algorithms. It is basically used for updating the parameters of the learning model.

Types of gradient Descent:

1. **Batch Gradient Descent: This is a type of gradient descent which processes all the training examples for each iteration of gradient descent. But if the number of training examples is large, then batch gradient descent is computationally very expensive. Hence if the number of training examples is large, then batch gradient descent is not preferred. Instead, we prefer to use stochastic gradient descent or mini-batch gradient descent.**
2. **Stochastic Gradient Descent: This is a type of gradient descent which processes 1 training example per iteration. Hence, the parameters are being updated even after one iteration in which only a single example has been processed. Hence this is quite faster than batch gradient descent. But again, when the number of training examples is large, even then it processes only one example which can be additional overhead for the system as the number of iterations will be quite large.**
3. **Mini Batch gradient descent: This is a type of gradient descent which works faster than both batch gradient descent and stochastic gradient descent. Here b**

examples where b<m are processed per iteration. So even if the number of training examples is large, it is processed in batches of b training examples in one go. Thus, it works for larger training examples and that too with lesser number of iterations.

Variables used:
Let m be the number of training examples.
Let n be the number of features.

Convergence trends in different variants of Gradient Descents:

In case of Batch Gradient Descent, the algorithm follows a straight path towards the minimum. If the cost function is convex, then it converges to a global minimum and if the cost function is not convex, then it converges to a local minimum. Here the learning rate is typically held constant.

In case of stochastic gradient Descent and mini-batch gradient descent, the algorithm does not converge but keeps on fluctuating around the global minimum. Therefore in order to make it converge, we have to slowly change the learning rate. However the convergence of Stochastic gradient descent is much noisier as in one iteration, it processes only one training example.

Module 4.3: Stochastic Gradient descent	Duration 3	Type code	Category: optimization

Stochastic Gradient Descent (SGD) is an optimization algorithm that randomly selects a few samples from a dataset for each iteration, making it computationally efficient for large datasets. This article explains the concept of SGD, its advantages over Batch Gradient Descent, and the trade-off between convergence speed and noise in the descent process.

the word 'stochastic' means a system or process linked with a random probability. Hence, in Stochastic Gradient Descent, a few samples are selected randomly instead of the whole data set for each iteration. In Gradient Descent, there is a term called "batch" which denotes the total number of samples from a dataset that is used for calculating the gradient for each iteration. In typical Gradient Descent optimization, like Batch Gradient Descent, the batch is taken to be the whole dataset. Although using the whole dataset is really useful for getting to the minima in a less noisy and less random manner, the problem arises when our dataset gets big.
Suppose, you have a million samples in your dataset, so if you use a typical Gradient Descent optimization technique, you will have to use all of the one million samples for completing one iteration while performing the Gradient Descent, and it has to be done for every iteration until the minima are reached. Hence, it becomes computationally very expensive to perform.
This problem is solved by Stochastic Gradient Descent. In SGD, it uses only a single sample, i.e., a batch size of one, to perform each iteration. The sample is randomly shuffled and selected for performing the iteration.

One thing to be noted is that, as SGD is generally noisier than typical Gradient

Descent, it usually took a higher number of iterations to reach the minima, because of its randomness in its descent. Even though it requires a higher number of iterations to reach the minima than typical Gradient Descent, it is still computationally much less expensive than typical Gradient Descent. Hence, in most scenarios, SGD is preferred over Batch Gradient Descent for optimizing a learning algorithm.

```
def SGD(f, theta0, alpha, num_iters):
    """
       Arguments:
       f -- the function to optimize, it takes a single argument
            and yield two outputs, a cost and the gradient
            with respect to the arguments
       theta0 -- the initial point to start SGD from
       num_iters -- total iterations to run SGD for
       Return:
       theta -- the parameter value after SGD finishes
    """
    theta = theta0
    for iter in range(num_iters):
      # For python 2.x - use xrange
        grad = f(theta)[1]

        # there is NO dot product ! return theta
        theta = theta - (alpha * grad)
```

This cycle of taking the values and adjusting them based on different parameters in order to reduce the loss function is called back-propagation.

Module 4.4: hyper parameter tuning	Duration 4	Type code	Category: optimization

Hyperparameter tuning is the process of finding the optimal values for the hyperparameters of a machine learning model. Grid search is a common approach where a predefined set of hyperparameter values is searched exhaustively. The code example demonstrates using GridSearchCV to find the best parameters for an SVM model.

Hyperparameter tuning is the process of finding the optimal values for the hyperparameters of a machine learning model. Hyperparameters are parameters that are not learned from the data but set prior to the model training. Tuning these hyperparameters can significantly impact the performance and generalization of the model.

One common approach for hyperparameter tuning is grid search, where a predefined

set of hyperparameter values is exhaustively searched to find the best combination. Here's an example using scikit-learn's GridSearchCV:

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
from sklearn.datasets import load_iris

# Load the dataset
iris = load_iris()
X = iris.data
y = iris.target

# Define the parameter grid
param_grid = {'C': [0.1, 1, 10], 'gamma': [0.1, 0.01, 0.001],
'kernel': ['linear', 'rbf']}

# Create the model
model = SVC()

# Perform grid search
grid_search = GridSearchCV(estimator=model, param_grid=param_grid,
cv=5)
grid_search.fit(X, y)

# Print the best parameters and best score
print("Best Parameters: ", grid_search.best_params_)
print("Best Score: ", grid_search.best_score_)
```

In this example, we use the Iris dataset and the Support Vector Machine (SVM) algorithm. We define a grid of possible hyperparameter values and create an SVM model. The GridSearchCV performs cross-validation with different hyperparameter combinations and returns the best parameters and best score.

Module 4.5: learning rate	Duration 6	Type concept	Category: optimization

Optimization techniques play a vital role in machine learning by fine-tuning the model's performance. Learning rate, weight decay, warm-up, learning rate scheduling, and other methods influence the convergence and generalization of models. Choosing the appropriate techniques is crucial for achieving efficient and effective optimization.

Learning Rate: Learning rate determines the step size at which the optimization algorithm updates the model parameters. A high learning rate can lead to overshooting the optimal solution, while a low learning rate may cause slow convergence. Finding an appropriate learning rate is crucial for efficient optimization.

Weight Decay: Weight decay, also known as L2 regularization, is a technique used to prevent overfitting by adding a penalty term to the loss function. It encourages the model to have smaller weights, reducing the impact of individual features and promoting generalization.

Warm-Up: Warm-up is an approach where the learning rate starts at a lower value and gradually increases during the initial iterations. It allows the model to stabilize and explore the parameter space before adjusting the learning rate.

Learning Rate Scheduling: Learning rate scheduling involves changing the learning rate over time during training. Common scheduling techniques include step decay (reducing the learning rate at specific steps), exponential decay (decaying the learning rate exponentially), and cosine annealing (using a cosine function to vary the learning rate). These techniques help fine-tune the learning process for improved convergence.

Other Techniques: Additional optimization techniques include momentum, which introduces inertia to accelerate convergence, and adaptive methods like AdaGrad, AdaDelta, and RMSprop, which adjust the learning rate based on the history of gradients or parameter updates.

Module 5.1: Logistic Regression	Duration 8	Type code	Category: Algorithm

logistic regression is a powerful algorithm for binary classification tasks. It models the relationship between input features and the probability of the positive class using the logistic function. By estimating the optimal coefficients through maximum likelihood estimation, logistic regression provides a reliable way to predict the probability of class membership.

Logistic Regression is a popular and widely used algorithm for binary classification tasks. It is a linear model that is commonly employed when the target variable is categorical and has only two possible outcomes. The goal of logistic regression is to

estimate the probability of the positive class based on the input features.

The underlying principle of logistic regression is to model the relationship between the input features and the probability of the positive class using the logistic function, also known as the sigmoid function. The logistic function maps any real-valued number to a value between 0 and 1, which can be interpreted as the probability of belonging to the positive class.

The logistic regression model assumes that the log-odds of the positive class is a linear combination of the input features. To train a logistic regression model, we need a labeled dataset where each instance is associated with the corresponding class label. The model is then trained using maximum likelihood estimation, aiming to find the optimal values for the coefficients that maximize the likelihood of observing the given dataset.

Here's a code snippet in Python using scikit-learn library to demonstrate how logistic regression can be implemented:

```
from sklearn.linear_model import LogisticRegression
from sklearn.mod
el_selection import train_test_split
from sklearn.metrics import accuracy_score

# Assuming X contains the input features and y contains the class labels
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create a logistic regression model
model = LogisticRegression()

# Train the model on the training data
model.fit(X_train, y_train)

# Make predictions on the test data
predictions = model.predict(X_test)

# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, predictions)
print("Accuracy:", accuracy)
```

In this code snippet, we split the dataset into training and test sets using the `train_test_split` function from scikit-learn. Then, we create an instance of the logistic regression model and train it on the training data using the `fit` method.

Afterward, we use the trained model to make predictions on the test data using the `predict` method. Finally, we evaluate the accuracy of the model by comparing the predicted labels with the true labels using the `accuracy_score` function.

Logistic regression is a simple yet powerful algorithm that can provide valuable insights and predictions for binary classification problems. It is widely used in various domains, including healthcare, finance, and marketing, due to its interpretability and ease of implementation.

Module 5.2: Decision tree	Duration 8	Type code	Category: Algorithm

Decision trees are classification algorithms that create a tree-like model based on feature splits. They provide interpretable results and can handle both categorical and numerical features. The Python snippet demonstrates training a decision tree classifier using scikit-learn on the Iris dataset.

A decision tree is a popular and interpretable classification technique in machine learning. It builds a tree-like model of decisions and their possible consequences. Each internal node in the tree represents a feature or attribute, and each leaf node represents a class label or a decision.

Here's a high-level overview of the decision tree algorithm:

> **Splitting: The decision tree algorithm starts by selecting the best feature from the dataset to split the data into subsets. The goal is to find the feature that best separates the classes or reduces the impurity within each subset. The most common splitting criteria are Gini Index and Information Gain.**
> **Recursive Partitioning: The algorithm recursively partitions the dataset based on the selected feature. This process continues until a stopping condition is met, such as reaching a maximum depth or having a minimum number of samples in each leaf.**
> **Leaf Node Assignment: Once the splitting process is complete, the algorithm assigns a class label to each leaf node. This label is typically determined by the majority class in the subset of data at that leaf node.**
> **Prediction: To make predictions, the algorithm traverses the decision tree by evaluating the feature values of a new input instance. The instance moves down the tree based on the feature conditions until it reaches a leaf node, which then provides the predicted class label.**

Here's an example Python code snippet using the scikit-learn library to train a

decision tree classifier:

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

# Load the dataset
dataset = datasets.load_iris()
X = dataset.data
y = dataset.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create a decision tree classifier
clf = DecisionTreeClassifier()

# Train the classifier
clf.fit(X_train, y_train)

# Predict on the test set
y_pred = clf.predict(X_test)
```

In summary, decision trees are versatile and interpretable classification algorithms that partition the data based on feature conditions to make predictions. They are widely used in various domains due to their simplicity and ability to handle both categorical and numerical features.

Module 5.3: Support Vector Classifier	Duration 5	Type code	Category: Algorithm

SVC is a versatile classification algorithm that finds the optimal hyperplane to separate different classes by maximizing the margin. It handles both linearly separable and non-linearly separable data using kernel functions, providing flexibility and accurate classification results.

Support Vector Classifier (SVC) is a machine learning algorithm that is widely used for binary and multi-class classification tasks. It builds a decision boundary or hyperplane to separate different classes in a high-dimensional space. SVC aims to

find the optimal hyperplane that maximizes the margin between the classes, allowing for better generalization and improved classification performance.

The key concept behind SVC is the use of support vectors. These are the data points that lie closest to the decision boundary or hyperplane. They are crucial in defining the hyperplane and influencing the classification decision. SVC works by finding the support vectors and determining the best hyperplane that separates the classes while maximizing the margin, which is the distance between the hyperplane and the support vectors.

SVC can handle both linearly separable and non-linearly separable data by utilizing different kernel functions. The kernel functions transform the input features into a higher-dimensional space, where the classes may become linearly separable. Commonly used kernel functions include the linear kernel for linearly separable data, polynomial kernel for polynomial decision boundaries, and the radial basis function (RBF) kernel for non-linear decision boundaries.

When using SVC, it is important to select the appropriate regularization parameter, C, which controls the trade-off between achieving a wider margin and correctly classifying training examples. A smaller C value allows for a wider margin but may lead to more misclassifications, while a larger C value focuses on correctly classifying training examples but may result in a narrower margin.

Here's an example Python code snippet using scikit-learn library to implement SVC:

```
from sklearn.datasets import load_iris
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load the Iris dataset
data = load_iris()
X = data.data
y = data.target

# Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create an SVC classifier
clf = SVC()

# Train the classifier
clf.fit(X_train, y_train)
```

```
# Make predictions on the test set
predictions = clf.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, predictions)
```

Module 5.4: K-Nearest Neighbors	Duration 6	Type code	Category: Algorithm

KNN is a simple and intuitive classification algorithm that classifies new samples based on the classes of its nearest neighbors. It is easy to understand and implement, making it a popular choice for classification tasks.

K-Nearest Neighbors (KNN) is a non-parametric classification algorithm that predicts the class of a sample based on the classes of its nearest neighbors in the feature space. KNN operates on the assumption that similar data points tend to belong to the same class. It is a simple yet effective algorithm for classification tasks.

In KNN, the number of nearest neighbors, denoted by 'K', is a hyperparameter that needs to be specified. When a new sample is presented for classification, the algorithm calculates the distances between the new sample and all other samples in the training set. It then selects the 'K' nearest neighbors based on the distance metric (e.g., Euclidean distance) and assigns the class label that is most prevalent among the neighbors to the new sample.

KNN is a lazy learning algorithm as it does not explicitly build a model during the training phase. Instead, it stores the entire training dataset and uses it during the prediction phase. This makes KNN computationally expensive for large datasets.

Here's an example Python code snippet using scikit-learn library to implement KNN:

```
from sklearn.datasets import load_iris
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load the Iris dataset
data = load_iris()
X = data.data
```

```
y = data.target

# Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create a KNN classifier
k = 3
clf = KNeighborsClassifier(n_neighbors=k)

# Train the classifier
clf.fit(X_train, y_train)

# Make predictions on the test set
predictions = clf.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, predictions)
```

Module 5.5: Random Forest	Duration 5	Type code	Category: Algorithm

Random Forest is a powerful ensemble learning algorithm that combines multiple decision trees to improve classification accuracy. It is robust, versatile, and effective in handling various types of datasets.

Random Forest is an ensemble learning algorithm that combines multiple decision trees to make predictions. It is a powerful and versatile classification technique that improves the accuracy and robustness of individual decision trees by reducing overfitting. Random Forest creates a collection of decision trees and aggregates their predictions to make the final classification.

Each decision tree in the Random Forest is built using a random subset of the training data and a random subset of features. This randomness introduces diversity among the trees, making them less correlated and more independent. During prediction, the class that is most frequently predicted by the individual trees is chosen as the final prediction.

Random Forest has several advantages, including handling high-dimensional data, handling both numerical and categorical features, and providing estimates of feature

importance. It is less prone to overfitting compared to a single decision tree, and it can handle imbalanced datasets effectively.

Here's a Python code snippet using scikit-learn library to implement Random Forest Classifier:

```
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load the Iris dataset
data = load_iris()
X = data.data
y = data.target

# Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create a Random Forest classifier
n_estimators = 100  # Number of trees in the forest
clf = RandomForestClassifier(n_estimators=n_estimators)

# Train the classifier
clf.fit(X_train, y_train)

# Make predictions on the test set
predictions = clf.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, predictions)
```

Module 5.6: Linear Discriminant Analysis	Duration 4	Type code	Category: Algorithm

Linear Discriminant Analysis is a dimensionality reduction technique and classification algorithm that finds a linear projection maximizing class separability. It is commonly used for feature extraction and improving classification performance.

Linear Discriminant Analysis (LDA) is a dimensionality reduction technique and a

supervised classification algorithm. It aims to find a linear combination of features that maximizes the separation between different classes. LDA calculates the optimal projection by maximizing the ratio of between-class scatter to within-class scatter.

LDA assumes that the data follows a Gaussian distribution and that the classes have equal covariance matrices. It projects the data onto a lower-dimensional space while preserving the class-discriminatory information. LDA is often used for feature extraction or as a preprocessing step before applying other classification algorithms.

Here's a Python code snippet using scikit-learn library to implement Linear Discriminant Analysis:

```
from sklearn.datasets import load_iris
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load the Iris dataset
data = load_iris()
X = data.data
y = data.target

# Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create a Linear Discriminant Analysis classifier
lda = LinearDiscriminantAnalysis()

# Fit the classifier to the training data
lda.fit(X_train, y_train)

# Make predictions on the test set
predictions = lda.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, predictions)
```

Module 5.7: Naive Bayes	Duration 7	Type code	Category: Algorithm
Gaussian Naive Bayes is a probabilistic classification algorithm that assumes feature			

independence and Gaussian distribution. It is suitable for both binary and multiclass classification tasks and performs well on datasets with continuous features.

Naive Bayes is a family of simple but powerful probabilistic classifiers based on Bayes' theorem with an assumption of independence between features. Despite its simplicity, it often performs well in various classification tasks and is particularly effective with large datasets.

The "naive" assumption in Naive Bayes refers to the assumption of feature independence, which means that the presence or absence of a particular feature in a class is unrelated to the presence or absence of other features. Although this assumption may not hold true in many real-world scenarios, Naive Bayes can still produce good results in practice.

Naive Bayes calculates the posterior probability of each class given an input using Bayes' theorem and then assigns the class with the highest probability. It estimates the prior probability of each class based on the training data and the likelihood of observing the features given each class. The classifier assumes that the features are conditionally independent given the class, which allows it to calculate the likelihood by multiplying the probabilities of individual feature occurrences.

There are different variants of Naive Bayes classifiers, such as Gaussian Naive Bayes, Multinomial Naive Bayes, and Bernoulli Naive Bayes, which are suitable for different types of data. Gaussian Naive Bayes assumes that continuous features follow a Gaussian distribution, while Multinomial Naive Bayes and Bernoulli Naive Bayes are commonly used for discrete features.

Despite its simplicity and the assumption of feature independence, Naive Bayes classifiers have shown remarkable performance in various real-world applications such as text classification, spam filtering, sentiment analysis, and document categorization. They are computationally efficient, easy to implement, and can handle high-dimensional data effectively.

Overall, Naive Bayes is a popular and widely used classification algorithm that leverages the principles of probability and independence assumptions to make predictions. It is particularly useful in scenarios where the data has a large number

of features and when interpretability and computational efficiency are important.

Here's a Python code snippet using scikit-learn library to implement Naive bayes:

```
from sklearn.datasets import load_iris
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load the Iris dataset
data = load_iris()
X = data.data
y = data.target

# Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create a Gaussian Naive Bayes classifier
nb = GaussianNB()

# Fit the classifier to the training data
nb.fit(X_train, y_train)

# Make predictions on the test set
predictions = nb.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, predictions)
```

Module 5.8: Classification and Regression Trees	Duration 4	Type code	Category: Algorithm

CART is a powerful and interpretable algorithm that can handle both categorical and numerical features. It can capture complex relationships and interactions between features and is robust to outliers. However, CART is prone to overfitting if the tree is allowed to grow too deep, so regularization techniques like pruning or limiting the tree depth are often applied.

CART (Classification and Regression Trees) is a machine learning algorithm that builds binary trees for classification and regression tasks. It follows a top-down,

recursive approach to partition the feature space based on the values of input features.

The CART algorithm aims to create the most optimal splits by evaluating the impurity of each potential split. In classification tasks, impurity measures like Gini impurity or entropy are used to assess the quality of splits. The goal is to minimize the impurity within each resulting partition and maximize the separation between different classes.

For regression tasks, CART uses a similar approach but instead of impurity measures, it assesses the error or variance reduction achieved by each potential split. The goal is to find splits that minimize the overall sum of squared errors or variance in the resulting partitions.

CART constructs the tree by iteratively choosing the best split at each node based on the impurity or error reduction. The process continues until a stopping criterion is met, such as reaching a maximum tree depth or having a minimum number of samples in a leaf node. The resulting tree can be interpreted as a set of if-else conditions that guide the decision-making process.

Once the CART tree is built, it can be used for making predictions on unseen data by traversing the tree based on the feature values of the input. In classification tasks, the predicted class is determined by the majority class in the corresponding leaf node. In regression tasks, the predicted value is typically the mean or median of the target values in the leaf node.

CART has several advantages, including its ability to handle both categorical and numerical features, capture complex interactions, and provide interpretable results. However, it can be prone to overfitting, especially if the tree is allowed to grow too deep. Regularization techniques such as pruning or limiting the tree depth can help mitigate this issue.

Overall, CART is a versatile and widely used algorithm for decision tree-based classification and regression tasks.

Here's an example of using CART for classification in Python with scikit-learn:

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

# Load the dataset
data = load_iris()
X = data.data
y = data.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create a CART classifier
classifier = DecisionTreeClassifier()

# Train the classifier
classifier.fit(X_train, y_train)

# Make predictions on the test set
y_pred = classifier.predict(X_test)

# Evaluate the accuracy of the classifier
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```