**Modelling with Decision Trees Algorithms**

In this tutorial, we learn Decision Tree Classification, attribute selection measures, and how to build and optimise Decision Tree Classifiers using the Python Scikit-learn package.

As a bank manager (finance provider), you must identify risky loan applications to achieve a lower loan default rate. This process of classifying customers into a group of approved and declined customers or safe or risky loan applications is known as a classification problem.

Classification is a two-step process: the learning step and the prediction step. In the learning step, the model is developed based on the training data. In the prediction step, the model predicts the response to a given test data. Decision Tree is one of the easiest and most popular classification algorithms to understand and interpret. It can be utilised for both classification and regression problems.

In this tutorial, you are going to cover the following topics:

* Decision Tree Algorithm
* How does the Decision Tree algorithm work?
* Attribute Selection Measures: Information Gain, Gain Ratio, Gini index, etc.
* Visualising Decision Trees
* Optimising Decision Tree Performance
* Classifier Building in Scikit-learn.
* Pros and Cons
* Conclusion

**Decision Tree Algorithm**

A decision tree is a flowchart-like tree structure where an internal node represents a feature(or attribute), the branch represents a decision rule, and each leaf node represents the outcome.

The topmost node in a decision tree is known as the root node. It learns to partition on the basis of the attribute value. It partitions the tree in a recursive manner called recursive partitioning. This flowchart-like structure helps you make decisions. Its visualisation is like a flowchart diagram, which easily mimics human-level thinking. That is why decision trees are easy to understand and interpret, see figure 1.

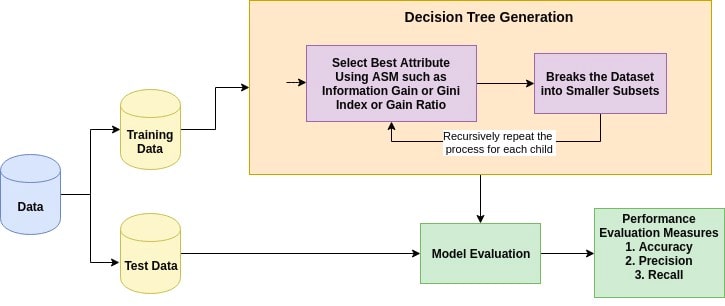


Decision Tree is a white box type of ML algorithm. It shares internal decision-making logic, which is unavailable in black-box algorithms such as Artificial Neural Networks (discussed in later tutorials). Its training time is faster compared to the neural network algorithm. The time complexity of decision trees is a function of the number of records and number of attributes in the given data. Decision trees can handle high-dimensional data with good accuracy.

**How does the Decision Tree algorithm work?**

The basic idea behind any decision tree algorithm is as follows:

1. Select the best attribute to split the instances using Attribute Selection Measures (ASM).
2. Make that attribute (feature) a decision node and break the dataset into smaller subsets.
3. Starts tree building by repeating this process recursively for each child until one of the conditions will match:
   * + All the tuples belong to the same attribute value.
     + There are no more remaining attributes.
     + There are no more instances.



**Attribute Selection Measures**

Attribute selection measure is a heuristic for selecting the splitting criterion that partitions data into the best possible manner. It is also known as splitting rules because it helps us to determine breakpoints for tuples on a given node. ASM provides a rank to each feature(or attribute) by explaining the given dataset. Best score attribute will be selected as a splitting attribute. In the case of a continuous-valued attribute, split points for branches also need to define. Most popular selection measures are Information Gain, Gain Ratio, and Gini Index.

**Information Gain**

Shannon invented the concept of entropy, which measures the impurity of the input set. In physics and mathematics, entropy referred as the randomness or the impurity in the system. In information theory, it refers to the impurity in a group of examples. Information gain is the decrease in entropy. Information gain computes the difference between entropy before split and average entropy after split of the dataset based on given attribute values. The attribute A with the highest information gain, Gain(A), is chosen as the splitting attribute at node N().

**Gain Ratio**

Information gain is biased for the attribute with many outcomes. It means it prefers the attribute with a large number of distinct values. For instance, consider an attribute with a unique identifier such as **client\_ID** has zero info(D) because of pure partition. This maximizes the information gain and creates useless partitioning.

C4.5, an improvement of ID3, uses an extension to information gain known as the gain ratio. Gain ratio handles the issue of bias by normalizing the information gain using Split Info.

**Gini index**

Another decision tree algorithm CART (Classification and Regression Tree) uses the Gini method to create split points.

The Gini Index considers a binary split for each attribute. You can compute a weighted sum of the impurity of each partition. If a binary split on attribute A partitions data D into D1 and D2, the Gini index of D is:

In case of a discrete-valued attribute, the subset that gives the minimum gini index for that chosen is selected as a splitting attribute. In the case of continuous-valued attributes, the strategy is to select each pair of adjacent values as a possible split-point and point with smaller gini index chosen as the splitting point. The attribute with minimum Gini index is chosen as the splitting attribute.

## Optimising Decision Tree Performance

Decision trees can be optimised with hyperparameter turning function, GridSeatrch, some of the hyperparameters which you could search are:

* **criterion : optional (default=”gini”) or Choose attribute selection measure**: This parameter allows us to use the different-different attribute selection measure. Supported criteria are “gini” for the Gini index and “entropy” for the information gain.
* **splitter : string, optional (default=”best”) or Split Strategy**: This parameter allows us to choose the split strategy. Supported strategies are “best” to choose the best split and “random” to choose the best random split.
* **max\_depth : int or None, optional (default=None) or Maximum Depth of a Tree**: The maximum depth of the tree. If None, then nodes are expanded until all the leaves contain less than min\_samples\_split samples. The higher value of maximum depth causes overfitting, and a lower value causes underfitting.
* **min\_samples\_split:** specifies the minimum number of samples required to split an *internal node*
* **min\_samples\_leaf:** specifies the minimum number of samples required to be at a *leaf* node.

For instance, if **min\_samples\_split = 5**, and there are 7 samples at an internal decision node, then the split is allowed. But let's say the split results in two leaves, one with 1 sample, and another with 6 samples. If **min\_samples\_leaf = 2**, then the split won't be allowed (even if the internal node has 7 samples) because one of the leaves resulted will have less than the minimum number of samples required to be at a leaf node.

In Scikit-learn, optimisation of the decision tree classifier is performed by only **pre-pruning**. The maximum depth of the tree can be used as a control variable for **pre-pruning**. In the following example, you can plot a decision tree on the same data with max\_depth=3. Other than pre-pruning parameters, You can also try other attribute selection-measure such as entropy.

Decision Tree models for bank loan approval and maximum allowed load amount

The loan approval dataset is a collection of financial records and associated information used to determine individuals' eligibility to obtain loans from a lending institution. It includes various factors such as income, employment status, loan term, loan amount, etc. This dataset is commonly used in machine learning and data analysis to develop models and algorithms that predict the likelihood of loan approval based on the given features. To maximise the bank’s returns, the bank can also estimate the maximum allowed loan amount for those who are approved for a loan; this practice is to maximise lending and minimise risks. There are ethical concerns about such practices, of course, around several factors like affordability.

Here, we have a dataset, and we are required to model two problems with decision tree algorithms.

**A) Decision Tree Classification (prediction) of Loan Approval Status:** For this, we need to extract a data subset from the original data to build classification models to approve {0} or decline {1} a person for a personal loan, predicting the **loan\_approval\_status**.

**B) Decision Tree Regression (prediction) of Maximum Loan Amount:** For this, we need to extract a second data subset from the original data to build regression models to predict the maximum loan amount, or loan limit, **max\_allowed\_loan** that can provided to clients with approved load status.

A maximum loan amount, or loan limit, describes the total amount of money that an applicant is authorised to borrow. Maximum loan amounts are used for standard loans, credit cards, and line-of-credit accounts.

**A) Data Understanding Phase**

1) In Google Colab, import all Python libraries to load, access, prepare and model the approval dataset.

# To transform your features into new values

from sklearn import preprocessing

# Import train\_test\_split function

from sklearn.model\_selection import train\_test\_split

# To construct and plot decision trees

from sklearn import tree

from matplotlib import pyplot as plt

# Import Decision Tree Classifier to predict loan approval status

from sklearn.tree import DecisionTreeClassifier

# Import Decision Tree Regressor to predict maximum loan amount

from sklearn.tree import DecisionTreeRegressor

#To optimise the Decision Tree model's hyperparameters

from sklearn.model\_selection import GridSearchCV

#Import scikit-learn metrics module for classification and regression performance metrics

from sklearn import metrics

# To create and plot the confusion matrix

from sklearn.metrics import confusion\_matrix

from sklearn.metrics import ConfusionMatrixDisplay

# To produce the classification metrics report

from sklearn.metrics import classification\_report

# To create, calculate and plot the RoC curve

from sklearn.metrics import RocCurveDisplay

2) Change pandas’ option to ensure better visibility of rendered dataset values, it can be a good idea to expand truncation and scientific notation e.

# To expand the truncation of rows and columns

pd.set\_option('display.max\_rows', 3000)

pd.set\_option('display.max\_columns', 3000)

pd.set\_option('display.width', 150)

# To expand e scientific notation

pd.set\_option('display.float\_format', '{:.2f}'.format)

3) Upload your data to google Colab then load your loan approval dataset

data = pd.read\_csv('/content/loan\_approval\_data.csv')

4) Understand your dataset, check its dimensions, data types, and summary stats, and look for any inconsistencies.

data.shape



data.info()

A screenshot of a computer program

Description automatically generated

data.describe(include='all').transpose()

Observe the potential outliers in the dataset from the summary stats below, in the variables: **age**, **employment\_length** and **loan\_interst\_rate**. From the summary stats, what indicates that these variables have outliers?

A screenshot of a graph

Description automatically generated

5) Understand and visualise your dataset. Check the distribution of your target variables. We also observe some categorical (object) variables, which require label encoding.

loan\_approval\_status\_fig = px.histogram(data, x='loan\_approval\_status')

loan\_max\_amount\_fig = px.histogram(data, x='max\_allowed\_loan')

loan\_approval\_status\_fig.show()

loan\_max\_amount\_fig.show()

A blue square with white squares

Description automatically generated

When analysing the above two histograms, observe that the number of people rejected for a loan is much less than those approved in **loan\_approval\_status**; this is a normal data behaviour in the real world. We expect to have more individuals approved than declined. When analysing the **max\_allowed\_amount**, we see two distributions. The first has about 8000 clients with zero max\_allowed\_amount, which seems plausible to those who rejected the loans, n= 8350. Also, observe from the second histogram that there are some negatives in the **max\_allowed\_amount**; this also indicates outliers, which require mitigation.The rest of the data follows a normal distribution with a very long tail to the right. This long right tail is normal behaviour since very few clients will be eligible for loans of very large amounts above £240K.

6) Discover outliers in your dataset. Find the variables which have outliers that require attention and treatment. It is best to use box plots for the numeric variables.

A graph with blue dots

Description automatically generated

A graph showing a number of dots

Description automatically generated

A graph of employment length

Description automatically generated

A graph with a line

Description automatically generated

A graph with a blue square

Description automatically generated

A graph with a line

Description automatically generated

A graph with blue lines

Description automatically generated

A graph with a line and a square

Description automatically generated with medium confidence

Observe in the above box plots some variables have outliers that are extreme and are regarded as invalid, while others are high, but they are within a valid range. The invalid outliers are in the following variables:

Age = 123

emplyment\_length = 123 and 150

loan\_interest\_rate = -11.14

max\_allowed\_loan = -£89k, -£111.739K and -£2.4269M

7) Explore the portion of missing data in your dataset to ensure that an appropriate method is used for missingness mitigation.

data.isna().sum()/len(data)\*100

A screenshot of a phone

Description automatically generated

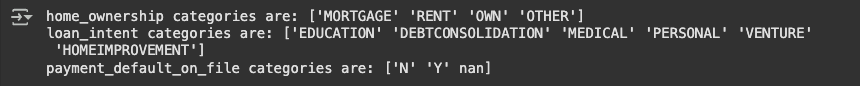
Observe that there is a relatively small portion of missing values in three different variables; the missingness is below 5%.

8) Check for inconsistent labelling in the categorical (object) variables by checking their unique values.

print('home\_ownership categories are:', data['home\_ownership'].unique())

print('loan\_intent categories are:',data['loan\_intent'].unique())

print('payment\_default\_on\_file categories are:',data['payment\_default\_on\_file'].unique())



Observe there is no incorrect labelling or duplications, we can conclude no inconsistencies can be found in labelling the object variables.

**B) Data Preparation Phase**

1) Find the outlier data points whose values were deemed invalid in the data understanding phase in four different variables. Build a function for finding outliers, and then drop them from the dataset.

#Create a function to find the outlier data points

def find\_outliers\_IQR(data):

q1=data.quantile(0.25)

q3=data.quantile(0.75)

IQR=q3-q1

outliers = data[((data<(q1-1.5\*IQR)) | (data>(q3+1.5\*IQR)))]

return outliers

# Find the outlier data points in the age variable

age\_outliers = find\_outliers\_IQR(data['age'])

print("number of outliers: "+ str(len(age\_outliers)))

print("max outlier value: "+ str(age\_outliers.max()))

print("min outlier value: "+ str(age\_outliers.min()))

age\_outliers.sort\_values(ascending=False)

A yellow sign with black text

Description automatically generated

# Find the outlier data points in emplyment\_length variable

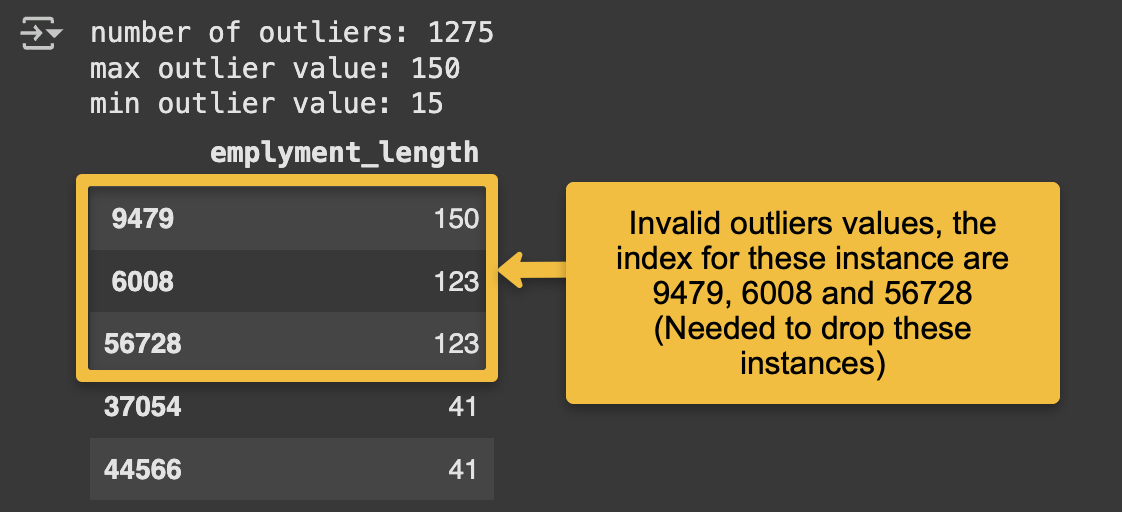
emplyment\_length\_outliers = find\_outliers\_IQR(data['emplyment\_length'])

print("number of outliers: "+ str(len(emplyment\_length\_outliers)))

print("max outlier value: "+ str(emplyment\_length\_outliers.max()))

print("min outlier value: "+ str(emplyment\_length\_outliers.min()))

emplyment\_length\_outliers.sort\_values(ascending=False)



# Find the outlier data points in loan\_interest\_rate variable

loan\_interest\_rate\_outliers = find\_outliers\_IQR(data['loan\_interest\_rate'])

print("number of outliers: "+ str(len(loan\_interest\_rate\_outliers)))

print("max outlier value: "+ str(loan\_interest\_rate\_outliers.max()))

print("min outlier value: "+ str(loan\_interest\_rate\_outliers.min()))

loan\_interest\_rate\_outliers.sort\_values(ascending=True)

A screenshot of a computer

Description automatically generated

# Find the outlier data points in max\_allowed\_loan variable

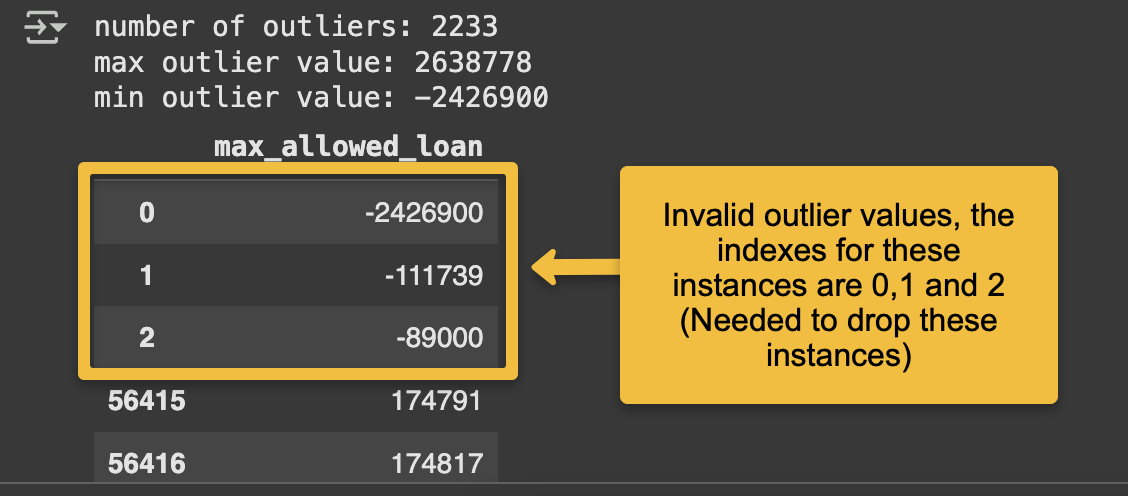
max\_allowed\_loan\_outliers = find\_outliers\_IQR(data['max\_allowed\_loan'])

print("number of outliers: "+ str(len(max\_allowed\_loan\_outliers)))

print("max outlier value: "+ str(max\_allowed\_loan\_outliers.max()))

print("min outlier value: "+ str(max\_allowed\_loan\_outliers.min()))

max\_allowed\_loan\_outliers.sort\_values(ascending=True)



By running the **find\_outlier\_IQR** function, we identified the indexes for the outlier instances with invalid values in the four variables of interest, **age, emplyment\_length, loan\_interest\_rate** and **max\_allowed\_loan.**

Now use the drop() method to remove all of the invalid outlier instances from the dataset. In total, there are 8 outlier data points to be removed from the dataset.

data.drop(data.index[[39792, 9479, 6008, 56728, 8998, 0, 1, 2]], inplace=True)

To verify the drop of these eight outliers, plot the box plots again for their four variable of interest **age, emplyment\_length, loan\_interest\_rate** and **max\_allowed\_loan.**

age\_fig = px.box(data, x='age')

emplyment\_length\_fig = px.box(data, x='emplyment\_length')

loan\_interest\_rate\_fig = px.box(data, x='loan\_interest\_rate')

max\_allowed\_loan\_fig = px.box(data, x='max\_allowed\_loan')

age\_fig.show()

emplyment\_length\_fig.show()

loan\_interest\_rate\_fig.show()

max\_allowed\_loan\_fig.show()

A graph with blue dots

Description automatically generated

A graph with blue dots

Description automatically generated

A graph with a purple square

Description automatically generated

A graph with blue dots

Description automatically generated

Observe the difference between the box plots in step 6 in the data understanding phase and the above box plots after dropping invalid outliers in data preparation.

2) Lets perform Label encoding for the categorical variables, use the label encoder method to convert all categories in object-type variables into a numeric format.

# Import label encoder

from sklearn import preprocessing

label\_encoder = preprocessing.LabelEncoder()

# Encode the categories in each object variable to a numeric form

data['home\_ownership']= label\_encoder.fit\_transform(data['home\_ownership'])

data['loan\_intent']= label\_encoder.fit\_transform(data['loan\_intent'])

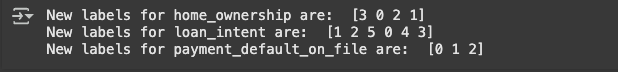
data['payment\_default\_on\_file']= label\_encoder.fit\_transform(data['payment\_default\_on\_file'])

# Check the categories' labels after the label encoding transformation is completed

print("New labels for home\_ownership are: ", data['home\_ownership'].unique())

print("New labels for loan\_intent are: ", data['loan\_intent'].unique())

print("New labels for payment\_default\_on\_file are: ", data['payment\_default\_on\_file'].unique())



3) Let’s use simple imputation to impute missing data values in the dataset. We will substitute the missing values with the mean of values in each variable using the **fillna()** method. From our data understanding phase, we know that only two numeric variables require imputation in the dataset, **age** and **loan\_interest\_rate**

# Calculate the mean of each variable

Mean\_age = data['age'].mean()

Mean\_loan\_interest\_rate = data['loan\_interest\_rate'].mean()

# use the mean to impute the missing values in each variable

data['age'].fillna(Mean\_age, inplace=True)

data['loan\_interest\_rate'].fillna(Mean\_loan\_interest\_rate, inplace=True)

Check for missing values after imputation, just to find there are no longer any missing values.

data.isna().sum()/len(data)\*100

A screenshot of a computer

Description automatically generated

4) Now all the issues in your data are mitigated or resolved, let’s save this clean (prepared) dataset as a csv file with the name **prepared\_loan\_approval\_data**

data.to\_csv(r'/content/prepared\_loan\_approval\_data.csv', index=False)

A screenshot of a computer

Description automatically generated

5) Time to generate two datasets from the **prepared\_loan\_approval\_data.** The first dataset will be used for classification to predict the first target variable **loan\_approval\_status.**  Therefore, we need to drop the other target **max\_allowed\_loan**  which will be used for regression modelling, using the drop function and save the emerged classification dataset as csv, lets call it **loan\_approval\_status\_data.**

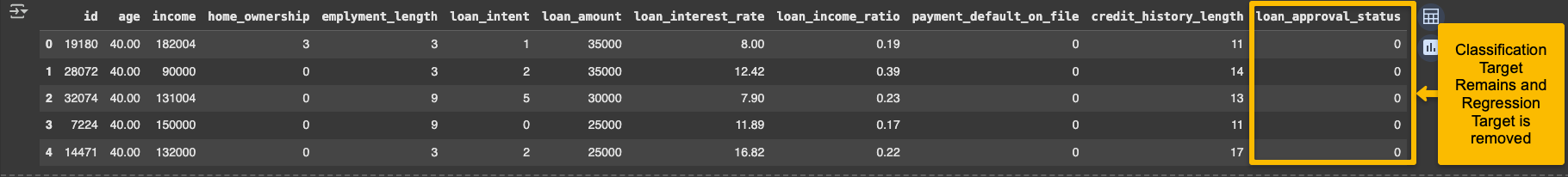
#Load your prepared dataset as df

df = pd.read\_csv('/content/prepared\_loan\_approval\_data.csv')

# Keep the target for classification and drop only the regression target max\_allowed\_loan

df.drop('max\_allowed\_loan',axis=1, inplace=True)

df.head()



#Save your classification dataset under the name loan\_approval\_status\_data

df.to\_csv(r'/content/loan\_approval\_status\_data.csv', index=False)

A screenshot of a computer

Description automatically generated

Your classification dataset is now saved. Make sure you download it from the Colab environment.

6) Time to generate the second dataset, the regression set. Load the full prepared dataset (58637 ✕ 13) **prepared\_loan\_approval\_data.** Remember, this dataset contains two target variables. To model the **max\_allowed\_loan** for the approved loan applicant {0}, we need to drop all the declined applicants {1} from the prepared original dataset. Therefore, we apply a filter to the classification target variable **loan\_approval\_status** : {0, 1} to only keep the status 0.

#load your fully prepared dataset as df2

df2=pd.read\_csv('/content/prepared\_loan\_approval\_data.csv')

# Filter approved applicants (**loan\_approval\_status** for status = 0) in a new data frame called approved\_loan\_applicants

approved\_loan\_applicants = df2[(df2.loan\_approval\_status < 1)]

# Check the basic stats to observe only approved applicants remain

approved\_loan\_applicants.describe().transpose()

A screenshot of a graph

Description automatically generated

7) Now all the instances required for regression modelling of **max\_allowed\_loan**  are filtered, we drop the useless variable **loan\_approval\_status**  from the data\_frame **approved\_loan\_applicants.**

# Drop the loan\_approval\_status variable from the data frame

approved\_loan\_applicants.drop('loan\_approval\_status',axis=1, inplace=True)

approved\_loan\_applicants.head()



Save your regression dataset for predicting **max\_allowed\_loan**  as a csv file under the name **loan\_max\_amount\_data** and ensure you download it from google Colab.

approved\_loan\_applicants.to\_csv(r'/content/loan\_max\_amount\_data.csv', index=False)

A screenshot of a computer

Description automatically generated

Finally ensure you downloaded the four dataset files uploaded into your Google Colab environment:

**loan\_approval\_data.csv** this is the original dataset

**prepared loan\_approval\_data.csv** this is the clean full dataset

**loan\_approval\_status\_data.csv** for classification modelling

**loan\_max\_amount\_data.csv** for regression modelling

A screenshot of a computer

Description automatically generated A screenshot of a computer

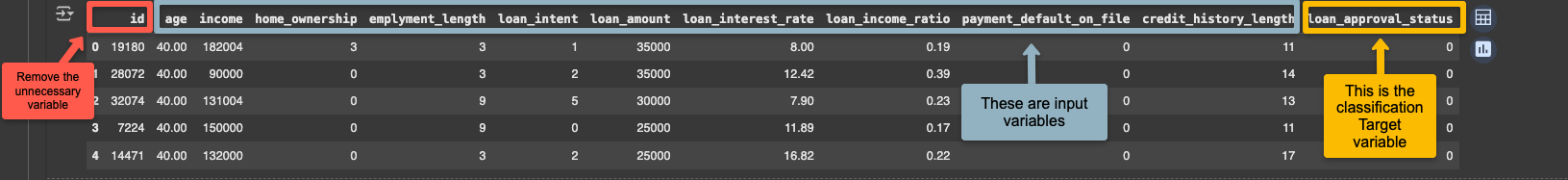
Description automatically generated

**C) Classification Modelling of loan\_approval\_status**

1) load your classification dataset file **loan\_approval\_status\_data.csv** for classification modelling into you Google Colab environment.

df\_loan\_status = pd.read\_csv('/content/loan\_approval\_status\_data.csv')

df\_loan\_status.head()



2) Define your **X** inputs and **y** target output in the modelling environment. Then split your data into training and test subsets for model building and evaluation.

# The inputs are all the variables in the data assigned to X except the target and id, hence dropped

X = df\_loan\_status.drop(['loan\_approval\_status', 'id'], axis=1)

# The target output is assigned to y

y = df\_loan\_status['loan\_approval\_status']

# Split the dataset in 60% Training and 40% Test with class stratification

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.4, random\_state=43, stratify=y)

3) using the **fit()** method, train a fully grown decision tree classifier, **clf**, on the training data subset then use to perform predictions on the test set. The construct and plot the fully grown decision tree.

# Build a fully grown decision tree clf

clf = DecisionTreeClassifier()

clf.fit(X\_train, y\_train)

#To make predictions on the test set, ues the predict method:

y\_pred\_clf = clf.predict(X\_test)

#You can size your plot then plot the fully grown classification tree

Unpruned\_Tree\_figure = plt.figure(figsize=(200,200))

DT\_Graph = tree.plot\_tree(clf, feature\_names=list(X\_train.columns), class\_names = ['0','1'], filled=True)

A white lines on a white background

Description automatically generated

As you can see, the fully grown tree is very complex to follow its rules and decision to conclude the final outcome, this is known as **model interpretability problem**. To find the depth of the fully grown tree:

# To find the full depth of the decision tree

print(clf.tree\_.max\_depth)



You can save your tree graph as **.png** image or for high resolution, you can save the tree images as vector image **.svg.** Vector images can be scaled up or down without losing any of its resolution. And they may require sophisticated software to access them such as adobe packages. Let’s save the file as **fully\_grown\_decistion\_tree.svg.** You can then download it from your google Colab environment side file menu.

#To save the DT graph

Unpruned\_Tree\_figure.savefig("fully\_grown\_decistion\_tree.svg")

A screenshot of a computer

Description automatically generated

4) using the **fit()** method, train a pruned decision tree classifier, **clf\_pruned**, on the training data subset then use to perform predictions on the test set. You can indicate the **max\_depth** of the DT, then construct and plot the pruned decision tree. Then save your **pruned\_decision\_tree.png**

# Create Decision Tree classifer object

clf\_pruned = DecisionTreeClassifier(max\_depth = 4)

# Train Decision Tree Classifer

clf\_pruned = clf\_pruned.fit(X\_train,y\_train)

#Predict the response for test dataset

y\_pred\_clf\_pruned = clf\_pruned.predict(X\_test)

#You can size your plot then plot the pruned classification tree

pruned\_Tree\_figure = plt.figure(figsize=(200,200))

DT\_Graph = tree.plot\_tree(clf\_pruned, feature\_names=list(X\_train.columns), class\_names = ['0','1'], filled=True)

#To save the pruned DT graph

pruned\_Tree\_figure.savefig("pruned\_decistion\_tree.png")

A screenshot of a computer

Description automatically generated

A diagram of a triangle

Description automatically generated

**D) Performance Evaluation of loan\_approval\_status prediction**

1) Let’s start by plotting the confusion matrix, the classification report and the AUC-Roc before and after pruning to understand the effect of pruning the decision tree.

# Construct and plot the confusion matrix for the fuly-grown tree (clf model)

clf\_cm\_test = confusion\_matrix(y\_test, y\_pred\_clf, labels=clf.classes\_ )

disp = ConfusionMatrixDisplay(clf\_cm\_test,display\_labels=clf.classes\_ )

disp.plot(values\_format = '.0f') # value\_format = .0f expands the scientific notation e inside the confusion matrix

# Produce the classification report for the fuly-grown tree (clf model)

print("clf fully grown tree report \n", classification\_report(y\_test, y\_pred\_clf))

# Plot the ROC curve for the fully-grown treen

clf\_Roc = RocCurveDisplay.from\_estimator(clf, X\_test, y\_test)

A screen shot of numbers and symbols

Description automatically generated

A chart with yellow and purple labels

Description automatically generated A graph of a positive label

Description automatically generated

# Construct and plot the confusion matrix for the pruned tree (clf\_pruned model)

clf\_pruned\_cm\_test = confusion\_matrix(y\_test, y\_pred\_clf\_pruned, labels = clf\_pruned.classes\_)

disp = ConfusionMatrixDisplay(clf\_pruned\_cm\_test,display\_labels = clf\_pruned.classes\_)

disp.plot(values\_format = '.0f')

# Produce the classification report for the prunded tree (clf\_pruned model)

print("clf\_pruned tree report \n", classification\_report(y\_test, y\_pred\_clf\_pruned))

# Plot the ROC curve for the pruned treen

clf\_pruned\_Roc = RocCurveDisplay.from\_estimator(clf\_pruned, X\_test, y\_test)

A screenshot of a graph

Description automatically generated

A chart with text and numbers

Description automatically generated with medium confidence A graph of a positive rate

Description automatically generated

The decision tree has a higher overall accuracy after pruning; however this higher accuracy is due to largely identifying more Approved Loan applications correctly from 94% to 99% (Recall of the 0 class before and after). A larger portion of the Declined Loan applications were incorrectly approved (from 31% to 42%) due to a 58% recall of the declined class 1 after pruning, this puts the bank at a higher risk of non-repayments.

**Homework 1 | Homework 1 | Homework 1 | Homework 1**

You can perform hyperparameter tuning on the fully grown tree with GridSearchCV to find out the best combinations of hyperparameters value for the tree, including the splitting criterion, the maximum depth and the splitter. Run the code below to find the best hyperparameters for the DT and document your observations, did the tuning process improve the model’s performance?

#create new a Decision Tree model

clf\_optimise = DecisionTreeClassifier()

#create a dictionary of all values we want to test for n\_neighbors and distances

param\_grid = {'max\_depth': np.arange(1, 35), 'criterion': ['gini', 'entropy', 'log\_loss'], 'splitter':['best', 'random'], }

#use gridsearch to test all values for n\_neighbors

clf\_gscv = GridSearchCV(clf\_optimise, param\_grid, cv=5, scoring = 'recall')

#fit model to data

clf\_gscv.fit(X, y)

clf\_gscv.best\_params\_

# Create Decision Tree classifer object

clf\_optimal = DecisionTreeClassifier(max\_depth = 19, criterion = "log\_loss", splitter = 'random')

# Train Decision Tree Classifer

clf\_optimal = clf\_optimal.fit(X\_train,y\_train)

#Predict the response for test dataset

y\_pred\_clf\_optimal = clf\_pruned.predict(X\_test)

#print the new classification report

print(classification\_report(y\_test, y\_pred\_clf\_optimal))

**Homework 2 | Homework 2 | Homework 2 | Homework 2**

**E) Regression Modelling of max\_allowed\_loan**

1) load your classification dataset file **loan\_approval\_status\_data.csv** for classification modelling into you Google Colab environment.

df\_loan\_max=pd.read\_csv('/content/loan\_max\_amount\_data.csv')

df\_loan\_max.head()

**A screenshot of a computer

Description automatically generated**

2) Define your **X** inputs and **y** target output in the modelling environment. Then split your data into training and test subsets for model building and evaluation.

X = df\_loan\_max.drop(['max\_allowed\_loan', 'id'], axis=1)

y = df\_loan\_max['max\_allowed\_loan']

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.4, random\_state=10)

3) using the **fit()** method, train a fully grown decision tree **regressor**, on the training data subset then use to perform predictions on the test set.

# To train the algorithm use fit method

regressor = DecisionTreeRegressor()

regressor.fit(X\_train, y\_train)

# To make predictions on the test set, ues the predict method:

y\_pred = regressor.predict(X\_test)

# Display the full tree depth

print("The full Regression Decision Tree Levels: ", regressor.tree\_.max\_depth)



4) using the **fit()** method, train a pruned decision tree classifier, **regressor\_pruned**, on the training data subset then use it to perform predictions on the test set. You can indicate the **max\_depth** of the DT, then construct and plot the pruned decision tree. Then save your **pruned\_decision\_tree.png**

# Limit the tree growth to 4 levels

pruned\_regressor = DecisionTreeRegressor(max\_depth=4)

pruned\_regressor.fit(X\_train, y\_train)

# To make predictions on the test set, ues the predict method:

y\_pred\_pruned = pruned\_regressor.predict(X\_test)

# Plot the regression DT

pruned\_Tree\_model = plt.figure(figsize=(50,50))

pruned\_Tree\_model\_Graph = tree.plot\_tree(pruned\_regressor, feature\_names=list(X\_train.columns), filled=True)

#To save the DT graph as a png image

pruned\_Tree\_model.savefig("pruned\_reg\_decision\_tree.png")

**A screenshot of a computer

Description automatically generated**

A group of objects connected to a wire

Description automatically generated with medium confidence

**F) Performance Evaluation of max\_allowed\_loan prediction**

1) Let’s start by obtaining the regression metrics, Mean Absolute Error (MAE), Mean Square Error (MSE) and Coefficient of Determination R2 , before and after pruning to understand the effect of pruning the decision tree.

# Calculating the regression metrics for the fully grown regression decision Tree

print('Mean Absolute Error:', metrics.mean\_absolute\_error(y\_test, y\_pred))

print('Mean Squared Error:', metrics.mean\_squared\_error(y\_test, y\_pred))

print('Root Mean Squared Error:', np.sqrt(metrics.mean\_squared\_error(y\_test, y\_pred)))

print('R2:', metrics.r2\_score(y\_test, y\_pred))

A computer code with numbers and a black background

Description automatically generated

# Calculating the regression metrics for the pruned regression decision Tree

print('Mean Absolute Error:', metrics.mean\_absolute\_error(y\_test, y\_pred\_pruned))

print('Mean Squared Error:', metrics.mean\_squared\_error(y\_test, y\_pred\_pruned))

print('Root Mean Squared Error:', np.sqrt(metrics.mean\_squared\_error(y\_test, y\_pred\_pruned)))

print('R2:', metrics.r2\_score(y\_test, y\_pred\_pruned))

**A computer code with numbers and a black background

Description automatically generated**

By looking at the regression metrics after pruning, we observe a large increase in the error compared to the fully grown regression tree. This indicates loss of important information as a result of excessive tree pruning. Using hyperparameter tuning with GridSearchCV will help estimate the best pruning hyperparameters.

#create new a Decision Tree model

regressor\_optimise = DecisionTreeRegressor()

#create a dictionary of all values we want to test for n\_neighbors and distances

param\_grid = {'max\_depth': np.arange(1, 22), 'criterion': ['squared\_error'], 'splitter':['best', 'random']}

#use gridsearch to test all values for n\_neighbors

regressor\_gscv = GridSearchCV(regressor\_optimise, param\_grid, cv=5, scoring = 'r2')

#fit model to data

regressor\_gscv.fit(X, y)

#get the best parameters

regressor\_gscv.best\_params\_



Now retrain the decision tree regressor with the new hyperparameters. Observe any improvement in the regression metrics calculations.

# Create Decision Tree classifer object

regressor\_optimal = DecisionTreeRegressor(max\_depth = 16, criterion = "squared\_error", splitter = 'best')

# Train Decision Tree Classifer

regressor\_optimal = regressor\_optimal.fit(X\_train,y\_train)

#Predict the response for test dataset

y\_pred\_regressor\_optimal = regressor\_optimal.predict(X\_test)

#print the new performance metrics

print('Mean Absolute Error:', metrics.mean\_absolute\_error(y\_test, y\_pred\_regressor\_optimal))

print('Mean Squared Error:', metrics.mean\_squared\_error(y\_test, y\_pred\_regressor\_optimal))

print('Root Mean Squared Error:', np.sqrt(metrics.mean\_squared\_error(y\_test, y\_pred\_regressor\_optimal)))

print('R2:', metrics.r2\_score(y\_test, y\_pred\_regressor\_optimal))

A computer screen shot of a error

Description automatically generated