



**IQRA University**

**CSC 471 Artificial Intelligence**

**Lab# 10**

**Data Preprocessing and Decision Tree Classifier**

**Objective:**

This lab discusses the data loading and pre-processing techniques frequently used within Machine Learning. The lab also provides the implementation steps for a basic Decision Tree classifier.

**Name of Student:** Faraz Alam

**Roll No:** 13948 **Sec.** Saturday (12:00pm-14:00pm)

**Date of Experiment:** 4/01/24

## Data Pre-processing

Data scientists come across many datasets and not all of them may be well formatted or noise free. While doing any kind of analysis with data it is important to clean it, as raw data can be highly unstructured with noise or missing data or data that is varying in scales which makes it hard to extract useful information. Pre-processing refers to the transformations applied to our data before feeding it to any machine learning algorithm. It 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

There are different techniques of data preprocessing some of which will be covered in this lab:

## Data Loading

In order to implement data pre-processing, first we need to load the raw dataset. In this lab, we will be using a sample dataset of CPU job scheduling as shown below:

Job Id	Burst time	Arrival Time	Preemptive	Resources
334	179	0.6875	1	4
234	340	0.78	0	4
138	143	0.915	1	4
463	264	nan	0	5
283	216	0.555	0	6
88	36	0.6625	0	5
396	128	0.1975	1	nan
470	203	0.9875	1	4
335	271	0.0275	0	3
272	399	0.215	nan	3
237	nan	0.4825	1	4
318	311	0.5675	1	1
84	111	0.2725	1	2
311	87	nan	0	7
163	103	0.46	0	5
453	213	0.0775	1	1
176	251	0.705	0	6
449	49	0.255	1	4
11	168	0.3175	1	7

To load this dataset into our application, we need to write a simple set of commands as:

```
import pandas as pd, scipy, numpy as np
from sklearn.preprocessing import MinMaxScaler

ds = pd.read_excel("Job_Scheduling.xlsx")

x=ds.iloc[:,0:4].values #for input values

y=ds.iloc[:,4].values #for output value
```

## 1. Dealing with Missing Data

We have some missing fields in the data denoted by “nan” which is an acronym for “not a number”. Machine learning models cannot accommodate missing fields in the data they are provided with. So, the missing fields must be filled with values that will not affect the variance of the data and make it less noisy.

The sklearn.impute python library contains SimpleImputer class that provides basic strategies for imputing missing values. Missing values can be imputed with a provided constant value, or using the statistics (mean, median or most frequent) of each column in which the missing values are located. This class also allows for different missing values encodings.

The following snippet demonstrates how to replace missing values, encoded as np.nan, using the mean value of the columns (axis 0) that contain the missing values:

```
#Filling missing values
from sklearn.preprocessing import Imputer
imp = Imputer(missing_values=np.nan, strategy="mean")
X = imp.fit_transform(x)
Y = y.reshape(-1,1)
Y = imp.fit_transform(Y)
Y = Y.reshape(-1)
print(Y)
```

```
[4.  4.  4.  5.  6.  5.  4.167 4.  3.  3.  4.  1.
 2.  7.  5.  1.  6.  4.  7. ]
```

The SimpleImputer class also supports categorical data represented as string values or pandas categoricals when using the 'most\_frequent' or 'constant' strategy:

```
import pandas as pd df =
pd.DataFrame([["a", "x"],
[ np.nan, "y"]],
```

```

        ["a", np.nan],
        ["b", "y"]], dtype="category")

imp = SimpleImputer(strategy="most_frequent")
print(imp.fit_transform(df))

```

The following snippet demonstrates how to replace missing values, encoded as np.nan, using the mean feature value of the two nearest neighbors of samples with missing values:

```

import numpy as np
from sklearn.impute import KNNImputer
nan = np.nan
X = [[1, 2, nan], [3, 4, 3], [nan, 6, 5], [8, 8, 7]]
imputer = KNNImputer(n_neighbors=2, weights="uniform")
imputer.fit_transform(X)

```

NaN is usually used as the placeholder for missing values. However, it enforces the data type to be float. The parameter missing\_values allows to specify other placeholder such as integer. In the following example, we will use -1 as missing values:

```

from sklearn.impute import MissingIndicator
X = np.array([[ -1, -1, 1, 3],
              [4, -1, 0, -1],
              [8, -1, 1, 0]])
indicator = MissingIndicator(missing_values=-1)
mask_missing_values_only = indicator.fit_transform(X)
mask_missing_values_only

```

## 2. Rescale Data

When the data is comprised of attributes with varying scales, many machine learning algorithms can benefit from rescaling the attributes to all have the same scale. It is also useful for algorithms that weight inputs like regression and neural networks and algorithms that use distance measures like K-Nearest Neighbors. Data can be rescaled using scikit-learn using the MinMaxScaler class.

$$v'_i = \frac{(v_i - \min_i(v_i))}{(\max_i(v_i) - \min_i(v_i))} \times (high - low) + low$$

Let us rescale the job burst time into the range of 0 and 1:

```
#Rescaling Burst Time
from sklearn.preprocessing import MinMaxScaler
scaler=MinMaxScaler(feature_range=(0,1))
rescaledX=scaler.fit_transform(X[:,1].reshape(-1,1))
numpy.set_printoptions(precision=3) #Setting precision for the output
X[:,1]= rescaledX.reshape(1,-1)
print(X[:,1])
```

---

```
[0.394 0.837 0.295 0.628 0.496 0.    0.253 0.46  0.647 1.    0.432 0.758
 0.207 0.14  0.185 0.488 0.592 0.036 0.364]
```

### 3. Normalization

Normalization involves adjusting the values in the feature vector so as to measure them on a common scale. Here, the values of a feature vector are adjusted so that they sum up to 1. Normalization is used to ensure that data points do not get boosted due to the nature of their features.

```
#Normalizing data  
from sklearn.preprocessing import Normalizer  
scaler=Normalizer().fit(X)  
normalizedX=scaler.transform(X)  
  
array([[1.000e+00, 1.179e-03, 2.058e-03, 2.994e-03],  
       [1.000e+00, 3.579e-03, 3.333e-03, 0.000e+00],  
       [9.999e-01, 2.136e-03, 6.630e-03, 7.246e-03],  
       [1.000e+00, 1.357e-03, 1.037e-03, 0.000e+00],  
       [1.000e+00, 1.752e-03, 1.961e-03, 0.000e+00],  
       [1.000e+00, 0.000e+00, 7.528e-03, 0.000e+00],  
       [1.000e+00, 6.400e-04, 4.987e-04, 2.525e-03],  
       [1.000e+00, 9.788e-04, 2.101e-03, 2.128e-03],  
       [1.000e+00, 1.932e-03, 8.209e-05, 0.000e+00],  
       [1.000e+00, 3.676e-03, 7.904e-04, 2.042e-03],  
       [1.000e+00, 1.824e-03, 2.036e-03, 4.219e-03],  
       [1.000e+00, 2.382e-03, 1.785e-03, 3.145e-03],  
       [9.999e-01, 2.459e-03, 3.244e-03, 1.190e-02],
```

The preprocessing module provides the `StandardScaler` utility class, which is a quick and easy way to perform the following operation on an array-like dataset:

The standard score of a sample  $v$  is calculated as:

$$v_i' = (v_i - \mu_v) / \sigma_v$$

where  $\mu_v$  is the mean of the training samples or zero if `with_mean=False`, and  $\sigma_v$  is the standard deviation of the training samples or one if `with_std=False`.

```
>>> from sklearn import preprocessing
>>> import numpy as np
>>> X_train = np.array([[ 1., -1.,  2.],
...                    [ 2.,  0.,  0.],
...                    [ 0.,  1., -1.]])
>>> scaler = preprocessing.StandardScaler().fit(X_train)
>>> scaler
StandardScaler()

>>> scaler.mean_
array([1. ..., 0. ..., 0.33...])

>>> scaler.scale_
array([0.81..., 0.81..., 1.24...])

>>> X_scaled = scaler.transform(X_train)
>>> X_scaled
array([[ 0. ..., -1.22...,  1.33...],
       [ 1.22...,  0. ..., -0.26...],
       [-1.22...,  1.22..., -1.06...]])
```

Scaled data has zero mean and unit variance:

```
>>> X_scaled.mean(axis=0) array([0.,
0., 0.])

>>> X_scaled.std(axis=0)
array([1., 1., 1.])
```

## Data Splitting

In machine learning we usually split our data into two subsets: training data and testing data (and sometimes to three: train, validate and test), and fit our model on the train data, in order to make predictions on the test data. Training data and test data are two important concepts in machine learning.

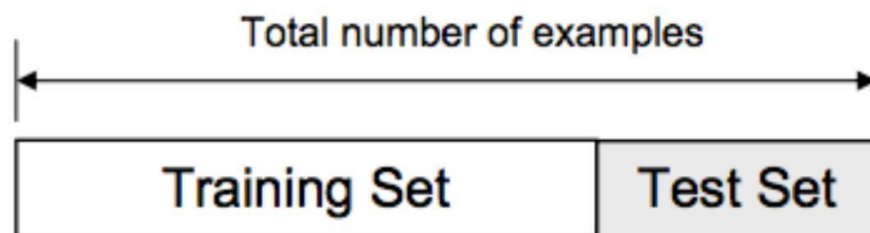
## Training Data

The training set contains a known output and the model learns on this data in order to be generalized to other data later on. The observations in the training set form the experience that the algorithm uses to learn. In supervised learning problems, each observation consists of an observed output variable and one or more observed input variables.

## Test Data

The test dataset (or subset) in order to test our model's prediction on this subset. The test set is a set of observations used to evaluate the performance of the model using some performance metric. It is important that no observations from the training set are included in the test set. If the test set does contain examples from the training set, it will be difficult to assess whether the algorithm has learned to generalize from the training set or has simply memorized it.

A program that generalizes well will be able to effectively perform a task with new data. In contrast, a program that memorizes the training data by learning an overly complex model could predict the values of the response variable for the training set accurately, but will fail to predict the value of the response variable for new examples. A program that memorizes its observations may not perform its task well, as it could memorize relations and structures that are noise or coincidence.



When the data is splitted, one of two thing might happen: we overfit the model or we underfit the model. We don't want any of these things to happen, because they affect the predictability of our model.

## Decision Tree

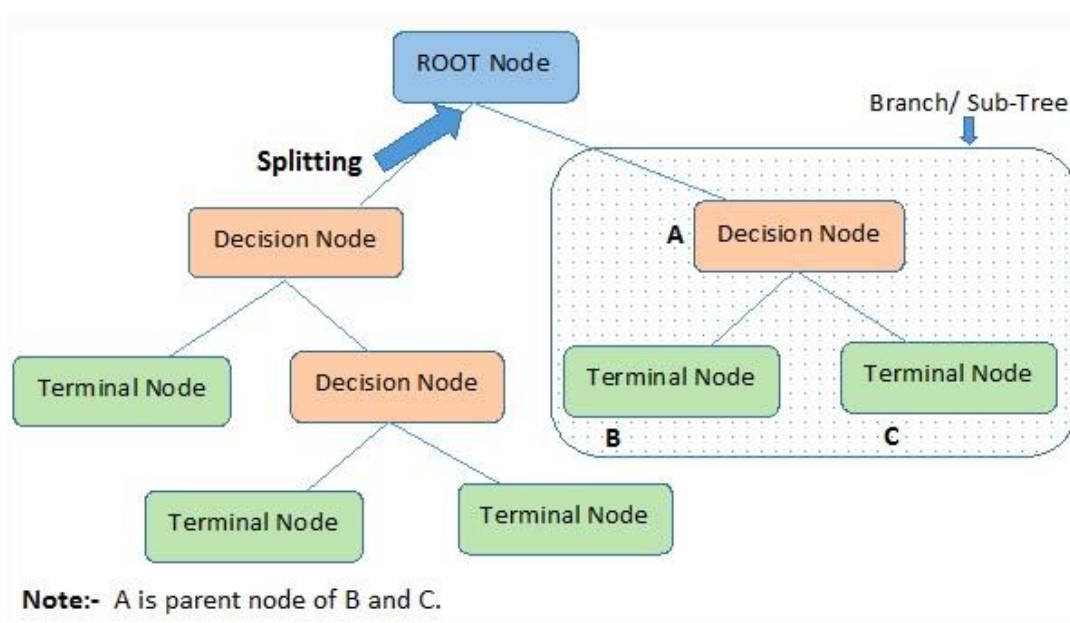
A decision tree is a structure that can be used to divide up a large collection of records into successively smaller sets of records by applying a sequence of simple decision rules. With each



successive division, the members of the resulting sets become more and more similar to one another.

A decision tree model consists of a set of rules for dividing a large heterogeneous population into smaller, more homogeneous groups with respect to a particular target variable. The target variable is usually categorical and the decision tree model is used either to calculate the probability that a given record belongs to each of the categories, or to classify the record by assigning it to the most likely class. Decision trees can also be used to estimate the value of a continuous variable, although there are other techniques more suitable to that task.

### Important Decision Tree Terms



A record enters the tree at the root node. The root node applies a test to determine which child node the record will encounter next. There are different algorithms for choosing the initial test, but the goal is always the same: To choose the test that best discriminates among the target classes. This process is repeated until the record arrives at a terminal node. All the records that end up at a given terminal of the tree are classified the same way. There is a unique path from the root to each leaf. That path is an expression of the rule used to classify the records.

### How a Decision Tree Is Grown

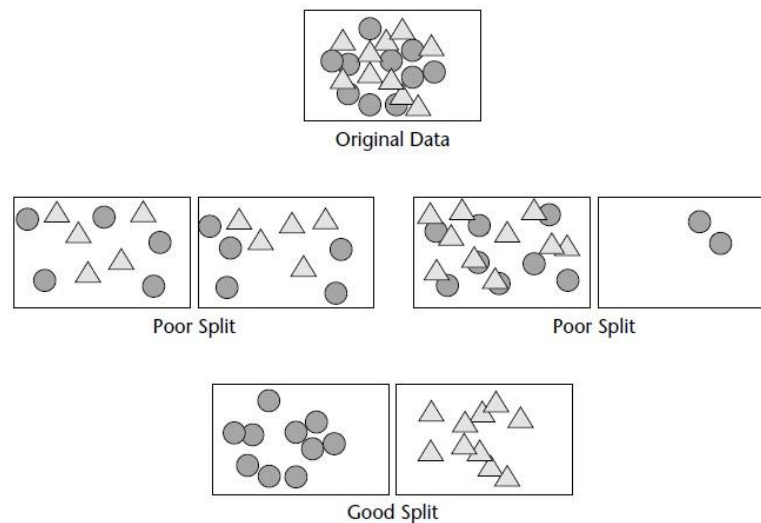
Although there are many variations on the core decision tree algorithm, all of them share the same basic procedure: Repeatedly split the data into smaller and smaller groups in such a way that each new generation of nodes has greater purity than its ancestors with respect to the target variable. For most of this discussion, we assume a binary, categorical target variable. At the start of the process, there is a training set consisting of preclassified records—that is, the value of the target



variable is known for all cases. The goal is to build a tree that assigns a class (or a likelihood of membership in each class) to the target field of a new record based on the values of the input variables.

The tree is built by splitting the records at each node according to a function of a single input field. The first task, therefore, is to decide which of the input fields makes the best split. The best split is defined as one that does the best job of separating the records into groups where a single class predominates in each group.

The measure used to evaluate a potential split is purity.



## Measures of Calculating the Purity

A number of different measures are available to evaluate potential splits. Algorithms developed in the machine learning community focus on the increase in purity resulting from a split, while those developed in the statistics community focus on the statistical significance of the difference between the distributions of the child nodes. Alternate splitting criteria often lead to trees that look quite different from one another, but have similar performance. That is because there are usually many candidate splits with very similar performance. Different purity measures lead to different candidates being selected, but since all of the measures are trying to capture the same idea, the resulting models tend to behave similarly

Purity measures for evaluating splits for categorical target variables include:

- Gini (also called population diversity)
- Entropy (also called information gain)
- Information gain ratio
- Chi-square test

## Growing the Full Tree

The initial split produces two or more child nodes, each of which is then split in the same manner as the root node. Once again, all input fields are considered as candidate splitters, even fields already used for splits. However, fields that take on only one value are eliminated from consideration since there is no way that they can be used to create a split. A categorical field that has been used as a splitter higher up in the tree is likely to become single-valued fairly quickly. The best split for each of the remaining fields is determined. When no split can be found that significantly increases the purity of a given node, or when the number of records in the node reaches some preset lower bound, or when the depth of the tree reaches some preset limit, the split search for that branch is abandoned and the node is labeled as a leaf node.

### Student Exercise

#### Task 1

**Thoroughly attempt and execute the notebooks provided with this lab.**

#### Task 2

**For the heart disease dataset (heart.csv), use the categorical variable to learn a decision tree to classify the records. Your model should be learnt against a 70-30 testing training split. Report the accuracy of the trained model and plot the decision tree learnt.**

#### Task 3

**For the diabetes dataset (diabetes.csv), use the categorical variable to learn a decision tree to classify the records. Your model should be learnt against a 70-30 testing training split. Explore how the decision tree handles the continuous variables. Also explore the options available for the handling of continuous variables. Report the accuracy of the trained model and plot the decision tree learnt.**