Department of Computer Engineering
University of Peradeniya
CO544: Machine Learning and Data Mining

Lab 03: Decision Trees and k-Nearest Neighbors Classification

June 10, 2024

Objectives

To demonstrate

- the decision tree algorithm, attribute selection measures, optimize decision tree performance, evaluate performance, visualize decision trees and provide students hands-on experience in building a classifier using `scikit-learn`.
- the k-Nearest Neighbors (kNN) algorithm, distance metrics, parameter tuning, and comparative evaluation against decision trees.

Preliminaries

In this lab, you will use a set of predefined Python functions to build and manipulate decision trees. You need the following Python packages installed:

- `scikit-learn` and `pandas`
- `graphviz` and `pydotplus` for plotting

1 Decision Tree Algorithm

A decision tree is a flowchart-like structure where:

- The topmost node is the **Root Node**: this represents the entire population or a sample, and it is further divided into two or more further nodes.
- An internal **Decision Node** represents a feature: when a subnode splits into further subnodes, it is called decision node.
- A branch represents a decision rule - **Splitting**: a process of dividing a node into two or more subnodes based on a certain rule.
- A **Leaf/Terminal Node** represents an outcome: the final node in a decision.

The decision tree is a white-box type of machine learning algorithm. It shares the logic of internal decision-making, which is not available in black-box algorithms, such as neural networks. Its training time is relatively fast compared to the neural network algorithm. The time complexity of decision trees is a function of the number of records and the number of attributes in a given data. The decision tree is a distribution-free or non-parametric method, which does not rely on probability distribution assumptions. Decision trees can handle higher dimensional data with better accuracy.

The intuition behind any decision tree algorithm is as follows:

1. Select the best attribute using attribute selection measures (ASM) to split records.
2. Turn that attribute into a decision node and split the dataset into smaller subsets.

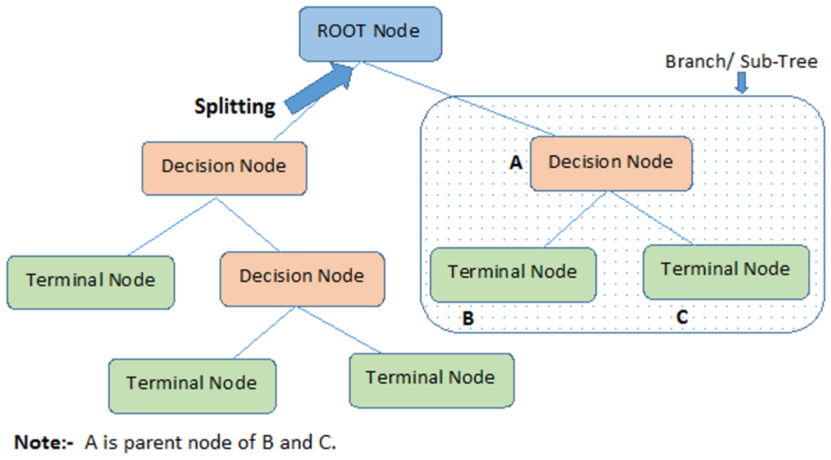

Figure 1: A typical decision tree

3. Start building the tree by repeating this process recursively to each child until one of the conditions is met:
   - All tuples belong to the same attribute value.
   - No more remaining attributes.
   - No more instances.

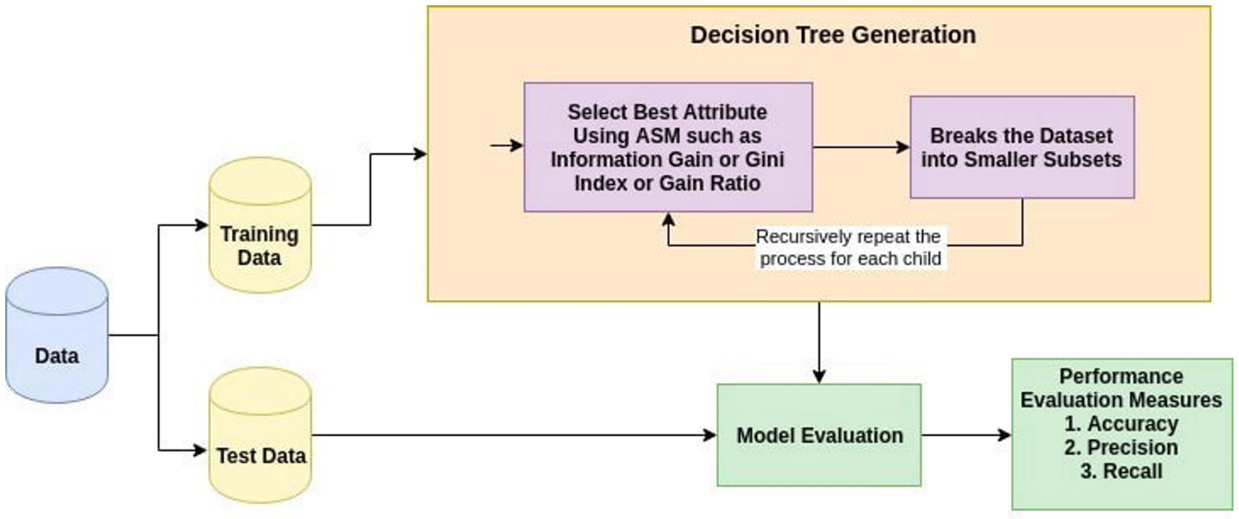
Decision Tree Generation
Select Best Attribute Using ASM such as Information Gain or Gini Index or Gain Ratio
Breaks the Dataset into Smaller Subsets
Recursively repeat the process for each child
Training Data
Data
Test Data
Model Evaluation
Performance Evaluation Measures
1. Accuracy
2. Precision
3. Recall

Figure 2: A complete pipeline of a typical decision tree algorithm-based ML model

1.1 Attribute Selection Measures (ASM)

ASM is a heuristic for selecting data splitting criteria in the best possible way. It is also known as splitting rules because it helps us to determine breakpoints for tuples on a given node. ASM provides a ranking for each feature (or attribute) by explaining the given dataset. The best score attribute is selected as a splitting attribute (Devi and Nirmala, 2013). In the case of a continuous-valued attribute, split points for branches should also be defined. The most popular selection measures are information gain, gain ratio, and Gini index.

1.1.1 Information Gain

Shannon (1948) invented the concept of entropy, which measures the impurity of the input set. In general, entropy is referred to as the randomness or the impurity in a system. In information theory, it refers to the impurity in a group of examples. Information gain means decreasing entropy. It computes the difference between the entropy before splitting and the average entropy after splitting the dataset based on given attribute values. The iterative dichotomiser (ID3) decision tree algorithm uses entropy to calculate information gain.

Shannon entropy is defined as:

$$\text{Info}(D) \;=\; -\sum_{i=1}^{m} p_i \log_2 p_i$$

The expected entropy after splitting on attribute A is:

$$\text{Info}_A(D) \;=\; \sum_{j=1}^{v} \frac{|D_j|}{|D|}\,\text{Info}(D_j)$$

Information gain is:

$$\text{Gain}(A) \;=\; \text{Info}(D) - \text{Info}_A(D)$$

The attribute A with the highest information gain [i.e., Gain(A)] is selected as the splitting attribute at node N().

1.1.2 Gain Ratio

Information gain is biased for an attribute with many outcomes. This means that it prefers an attribute with a large number of distinct values. Information gain maximizes the information retrieval and creates useless partitions.

$$\text{SplitInfo}_A(D) \;=\; -\sum_{j=1}^{v} \frac{|D_j|}{|D|}\,\log_2\!\left(\tfrac{|D_j|}{|D|}\right)$$

where v is the number of discrete values in attribute A.

The gain ratio can be defined as:

$$\text{GainRatio}(A) \;=\; \frac{\text{Gain}(A)}{\text{SplitInfo}_A(D)}$$

The attribute with the highest gain ratio is selected as the splitting attribute (Nair et al., 2010).

1.1.3 Gini Index (CART)

Another attribute selection measure used by the categorical and regression trees (CART) to create split points is the Gini index.

$$\text{Gini}(D) \;=\; 1 - \sum_{i=1}^{m} p_i^2$$

The Gini Index considers a binary split for each attribute. Here a weighted sum of the impurity of each partition can be calculated. If a binary split on attribute A partitions data D into D_1 and D_2, the Gini index of D can be written as:

$$\text{Gini}_A(D) \;=\; \frac{|D_1|}{|D|}\,\text{Gini}(D_1) + \frac{|D_2|}{|D|}\,\text{Gini}(D_2)$$

In the case of a discrete-valued attribute, the subset that gives the minimum Gini index is selected as a splitting attribute, and 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 a smaller Gini index as the splitting point.

1.2 Dataset Description

In this lab work, the Pima Indians Diabetes dataset is used to build the decision tree. This dataset is originally from the National Institute of Diabetes and Digestive and Kidney Diseases. The objective of the dataset is to diagnostically predict whether or not a patient has diabetes, based on certain diagnostic measurements included in the dataset. Several constraints were placed on the selection of these instances from a larger database. In particular, all patients here are females at least 21 years old of Pima Indian heritage. The dataset can be downloaded from Kaggle using the following url: https://www.kaggle.com/datasets/uciml/pima-indians-diabetes-database

1.3 Building Decision Trees

i. **Importing libraries**

```
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn import metrics
import warnings
warnings.filterwarnings('ignore')
```

ii. **Dataset loading and exploratory data analysis**
Loading Dataset

```
diabetes_df = pd.read_csv('diabetes.csv')
diabetes_df.head()  # Preview the dataset
diabetes_df.shape  # Number of instances and variables
```

Renaming columns

```
col_names = ['pregnant','glucose','bp','skin','insulin','bmi','pedigree','age
    ','label']
diabetes_df.columns = col_names  # Rename column names
```

Summary of dataset

```
diabetes_df.info()
```

Frequency distributions of values in variables

```
for col in col_names:
    print(diabetes_df[col].value_counts())
```

Exploring target variable

```
diabetes_df[   label   ].value_counts()
```

Checking missing values in variables

```
diabetes_df.isnull().sum()
```

iii. **Defining feature vector and target variable**

```
X = diabetes_df.drop(['label'], axis=1)
y = diabetes_df['label']
```

iv. **Splitting data**

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
    random_state=1)  # 75% training and 25% test

X_train.shape, X_test.shape  # Shapes of X_train and X_test
```

v. **Feature engineering: encoding categorical variables**

This is the process of transforming raw data into useful features that help us better understand our model better and increase its predictive power

```
X_train.dtypes # Check data types in X_train

import category_encoders as ce

encoder = ce.OrdinalEncoder(cols=X.columns.tolist())
X_train = encoder.fit_transform(X_train)
X_test  = encoder.transform(X_test)
```

vi. **Building decision tree classifier with the Gini index criterion**

```
clf_gini = DecisionTreeClassifier(criterion='gini', max_depth=4, random_state
    =0)

clf_gini.fit(X_train, y_train)  # Train the classifier
```

vii. **Predicting results for the test set**

```
y_pred = clf_gini.predict(X_test)
```

viii. Evaluating model

```
print('Accuracy:', metrics.accuracy_score(y_test, y_pred))
```

ix. **Confusion matrix**

A confusion matrix is a matrix that can be used to measure the performance of an machine learning algorithm, usually a supervised learning one. In general, each row of the confusion matrix represents the instances of an actual class and each column represents the instances of a predicted class, but it can be the other way around as well.

Four types of outcomes are possible while evaluating a classification model performance:

- **True Positives (TP)**– True Positives occur when we predict an observation belongs to a certain class and the observation actually belongs to that class.
- **True Negatives (TN)**– True Negatives occur when we predict an observation does not belong to a certain class and the observation actually does not belong to that class.
- **False Positives (FP)**– False Positives occur when we predict an observation belongs to a certain class but the observation actually does not belong to that class. This type of error is called **Type I error**.

- **False Negatives (FN)**– False Negatives occur when we predict an observation does not belong to a certain class but the observation actually belongs to that class. This is a very serious error and it is called **Type II error**.

```
from sklearn.metrics import confusion_matrix
conf_mat = confusion_matrix(y_test, y_pred)
```

1.4 Optimizing Decision Tree Performance

In scikit-learn, optimization of decision tree classifier performed by only pre-pruning. Maximum depth of the tree can be used as a control variable for pre-pruning. In addition to the pre-pruning parameters, other attribute selection measures such as entropy can be used.

- **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.

(source: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html)

1.5 Visualizing Decision Trees

```
from six import StringIO
from IPython.display import Image
from sklearn.tree import export_graphviz
import pydotplus

dot_data = StringIO()
export_graphviz(clf_gini,
                out_file=dot_data,
                filled=True,
                rounded=True,
                special_characters=True,
                feature_names=X.columns,
                class_names=['0','1'])

graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
graph.write_png('diabetes.png')

Image(graph.create_png())
```

1.6 Classification Report

Classification report is another way to evaluate the classification model performance. It displays the **precision**, **recall**, **f1**, and other support scores for the model.

```
from sklearn.metrics import classification_report

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

2 k-Nearest Neighbors (kNN)

kNN is a *lazy*, instance-based learning method: there is no explicit model building (training) step beyond storing the labeled examples. Instead, all "work" happens at prediction time, by comparing a new instance to the stored ones.

2.1 Key Concepts

- **Instance–based / Lazy learning:** The algorithm simply memorizes the training data. There is no global model or parameter fitting.
- **Distance metric:** Commonly Euclidean,

$$d(\mathbf{x}, \mathbf{x}') = \sqrt{\sum_{i=1}^{p} (x_i - x_i')^2},$$

  but you can also use Manhattan or Minkowski distances.
- **Number of neighbors (k):** Controls bias–variance trade-off.
  - Small k low bias, high variance (sensitive to noise).
  - Large k high bias, low variance (smoother decision boundary).
- **Feature scaling:** Essential so that no feature "dominates" the distance calculation.
- **Majority vote:** The class label is chosen by majority among the k nearest points.

2.2 Algorithmic Steps

1. **Preprocess / Store:**
   - Scale each feature (e.g. standardize to zero mean, unit variance).
   - Store the entire training set $(\mathbf{X}_{\text{train}}, \mathbf{y}_{\text{train}})$.
2. **Predict for a new point x:**
   (a) Compute the distance from $\mathbf{x}$ to each training point.
   (b) Identify the k smallest distances (the k "nearest neighbors").
   (c) Collect their labels and take a majority vote.
   (d) (Tie-breaking: either pick the class of the nearest neighbor or random.)
3. **Parameter Selection:**
   - Choose k (and distance metric) via cross-validation.
   - Use grid search to compare performance for different (k, metric) combinations.

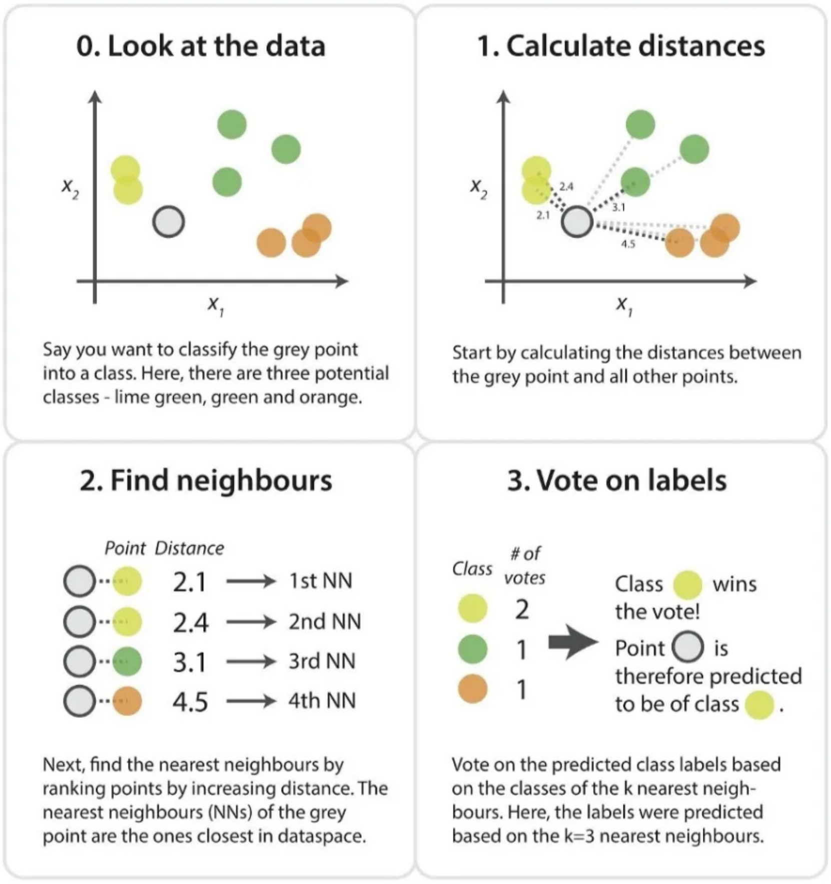
0. Look at the data
x_2
x_1
Say you want to classify the grey point into a class. Here, there are three potential classes - lime green, green and orange.
1. Calculate distances
x_2
2.4
2.1
3.1
4.5
x_1
Start by calculating the distances between the grey point and all other points.
2. Find neighbours
Point Distance
2.1 → 1st NN
2.4 → 2nd NN
3.1 → 3rd NN
4.5 → 4th NN
Next, find the nearest neighbours by ranking points by increasing distance. The nearest neighbours (NNs) of the grey point are the ones closest in dataspace.
3. Vote on labels
Class # of votes
2
1
1
Class wins the vote!
Point is therefore predicted to be of class .
Vote on the predicted class labels based on the classes of the k nearest neighbours. Here, the labels were predicted based on the k=3 nearest neighbours.

Figure 3: Illustration of the kNN decision process. (source: https://www.kdnuggets.com/2016/01/implementing-your-own-knn-using-python.html)

2.3 Advantages and Limitations

- **Pros:** Simple to implement; no training time; naturally handles multi-class.
- **Cons:** Prediction can be very slow for large datasets ($O(n)$ per query); sensitive to irrelevant or unscaled features; memory-intensive.

2.4 Implementation in Python

```
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV
from sklearn import metrics
from sklearn.metrics import classification_report

# 1. Scale features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled  = scaler.transform(X_test)

# 2. Hyperparameter tuning for k and distance metric
param_grid = {
    'n_neighbors': [3,5,7,9],
    'metric':      ['euclidean','manhattan']
}
grid = GridSearchCV(
    KNeighborsClassifier(),
    param_grid,
    cv=5,
    scoring='accuracy'
)
grid.fit(X_train_scaled, y_train)

best_knn   = grid.best_estimator_
y_pred_knn = best_knn.predict(X_test_scaled)

print('Best params:',  grid.best_params_)
print('kNN accuracy:', metrics.accuracy_score(y_test, y_pred_knn))
print(classification_report(y_test, y_pred_knn))
```

2.5 Decision Boundary Visualization (2D projection)

If you pick any two features (here we use the first two), you can plot kNN's decision regions:

```
import numpy as np
import matplotlib.pyplot as plt

# Use only first two features
X2_train = X_train_scaled[:, :2]
X2_test  = X_test_scaled[:,  :2]

knn2 = KNeighborsClassifier(
    n_neighbors=grid.best_params_['n_neighbors'],
    metric      =grid.best_params_['metric']
)
knn2.fit(X2_train, y_train)

# create meshgrid
h = 0.02
x_min, x_max = X2_train[:,0].min()-1, X2_train[:,0].max()+1
y_min, y_max = X2_train[:,1].min()-1, X2_train[:,1].max()+1
xx, yy = np.meshgrid(
    np.arange(x_min, x_max, h),
    np.arange(y_min, y_max, h)
)

# predict on grid
Z = knn2.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

plt.figure()
plt.contourf(xx, yy, Z, alpha=0.4)
plt.scatter(
    X2_train[:,0], X2_train[:,1],
    c=y_train, edgecolor='k', marker='.'
)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title(
  f'kNN Decision Boundary (k={grid.best_params_[\'n_neighbors\']})'
)
```

```
plt.show()
```

Submission

Write a short report based on the following tasks. The report should be submitted as a single file (in `.md`, `.doc`, or `.docx` format) and should not longer than four pages. Rename it as `exxyyy_lab03_report.md/.doc/.docx`, where xx indicates your batch and yyy indicates your registration number.

1. **Task 1:** Build two decision tree classifiers with **Gini index** and **entropy** criteria for the given **Wine.csv** dataset. More information on the dataset is available on `UCI Machine Learning Repository` (source: `https://archive.ics.uci.edu/ml/datasets/Wine`).
   (a) Demonstrate how decision trees deal with missing values.
   (b) Evaluate the classifiers with suitable performance metrics.
   (c) Demonstrate how pruning can be applied to overcome overfitting of decision tree classifiers.
   (d) Visualize decision trees.
2. **Task 2:** Apply k-Nearest Neighbors to the same **Wine.csv** dataset.
   (a) Preprocess with feature scaling.
   (b) Tune k (and distance metric) via cross-validation.
   (c) Compare kNN's accuracy, precision/recall, and runtime to your decision-tree results.