University of Dhaka

Department of Computer Science and Engineering

CSE-4255 : Introduction to Data Mining and Warehousing Lab

Lab Report: Comparative Analysis of Classification Algorithms (Decision Tree and Naïve Bayes)

Submitted By:

Name : Zisan Mahmud

Roll No : 23

Submitted On:

JUNE 25, 2025

Submitted To:

Dr. Chowdhury Farhan Ahmed

Md. Mahmudur Rahman

Contents

1 ABSTRACT

This report presents a comparative analysis of Decision Tree and Naïve Bayes algorithm. Total ten datasets are used and both algorithms are implemented on the datasets to determine the most efficient method. The differet accuracy measures (accuracy, precision, recall, F-measure, AUC) of the two algorithms is also analyzed.

keywords: Decision Tree, Naïve Bayes.

2 INTRODUCTION

Data classification is a fundamental task in data mining that categorizes data points into predefined classes based on their attributes. The objective is to construct a model from a labeled training dataset which can then be used to accurately predict the class of new, unclassified data. This is also called supervised learning. The efficacy of this predictive task is heavily dependent on the underlying algorithm used to build the classification model.

In this assignment. I presents a comparative analysis of two widely recognized classification algorithms: the Decision Tree and the Naïve Bayes classifier. While both are used for predictive modeling, they originate from different theoretical paradigms. Decision Trees employ a rule-based, hierarchical structure, whereas Naïve Bayes classifiers utilize principles of probability. Analyzing the process of this two algorithm provides with a deep knowledge on how they work on different types of data in real world.

The Decision Tree Classifier

A Decision Tree is a non-parametric supervised learning method that predicts the value of a target variable by learning simple decision rules inferred from the data features. It employs a tree-like structure, where each internal node denotes a test on an attribute, each branch represents an outcome of the test, and each leaf node holds a class label.

The construction of the tree, known as induction, is a recursive process of partitioning the data. At each node, an attribute selection measure is used to identify the attribute that most effectively splits the data into purer subsets.

A prominent measure for this purpose is **Information Gain**, which is based on the concept of entropy. The information gain, $Gain(A)$, of an attribute A is calculated as:

$$Gain(A) = Info(D) - Info_A(D)$$

Where:

- $Info(D)$ is the entropy of the original dataset D, calculated as:

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

  Here, m is the number of classes and p_i is the probability of a tuple in D belonging to class C_i.

- $Info_A(D)$ is the expected information (entropy) after the dataset D has been partitioned by attribute A:

$$Info_A(D) = \sum_{j \in Values(A)} \frac{|D_j|}{|D|} x Info(D_j)$$

  Here, $Values(A)$ is the set of all possible values for attribute A, and D_j is the subset of D for which attribute A has value j. The attribute with the highest information gain is chosen as the splitting criterion. While interpretable, Decision Trees can be susceptible to overfitting, where the model captures noise in the training data, potentially leading to lower accuracy on unseen data.

The Naïve Bayes Classifier

In contrast, the Naïve Bayes classifier is a probabilistic model grounded in **Bayes' Theorem**. It calculates the probability of a data instance belonging to a specific class given a set of attributes. For a data instance $\mathbf{X}$ with attributes $(x_1, x_2, \ldots, x_n)$ and a set of classes $(C_1, C_2, \ldots, C_m)$, the classifier predicts that $\mathbf{X}$ belongs to the class C_i if and only if:

$$P(C_i|\mathbf{X}) > P(C_j|\mathbf{X}) \quad \text{for } 1 \leq j \leq m, j \neq i$$

The posterior probability, $P(C_i|\mathbf{X})$, is calculated using Bayes' Theorem:

$$P(C_i|\mathbf{X}) = \frac{P(\mathbf{X}|C_i)P(C_i)}{P(\mathbf{X})}$$

Where:

- $P(C_i|\mathbf{X})$ is the posterior probability of class C_i given the attribute vector $\mathbf{X}$.
- $P(\mathbf{X}|C_i)$ is the likelihood of observing $\mathbf{X}$ given that it belongs to class C_i.
- $P(C_i)$ is the prior probability of class C_i.
- $P(\mathbf{X})$ is the prior probability of the predictor vector $\mathbf{X}$.

The algorithm's "naïve" assumption is the class-conditional independence of attributes. This presumes that the effect of an attribute's value on a given class is independent of the values of other attributes. This simplifies the computation of the likelihood to:

$$P(\mathbf{X}|C_i) = \prod_{k=1}^{n} P(x_k|C_i)$$

Consequently, this simplifies the classification process, making the algorithm highly efficient and particularly effective for high-dimensional data, such as in text classification.

3 IMPLEMENTATION DETAIL

This section outlines the implementation specifics for both the Decision Tree and Naïve Bayes classifier.

3.1 Decision Tree Classifier

3.1.1 Data Structure

The tree is represented using a custom `TreeNode` class with the following attributes:

```
@dataclass
class TreeNode:
    feature: Optional[str] = None
    value: Optional[float] = None
    children: Dict[Any, 'TreeNode'] = field(default_factory=dict)
    label: Optional[Any] = None
```

Where:

- `feature`: The feature name on which to split (for internal nodes)
- `value`: The threshold value for continuous features
- `children`: A dictionary mapping feature values to child nodes
- `label`: The class prediction (for leaf nodes)

3.1.2 Core Algorithm

The decision tree is built recursively using the following algorithm:

Algorithm 1 Build Decision Tree

```
function BuildTree($X, y, depth$)
    if StoppingCriteriaMet($X, y, depth$) then
        return leaf node with majority class
    end if
    $feature \leftarrow$ FindBestFeature($X, y$)
    if $feature$ is None then
        return leaf node with majority class
    end if
    $node \leftarrow$ new TreeNode($feature.name, feature.value$)
    $subsets \leftarrow$ SplitData($X, y, feature$)
    if $feature$ is continuous then
        $node.children["left"] \leftarrow$ BuildTree($leftSubset.X, leftSubset.y, depth+1$)
        $node.children["right"] \leftarrow$ BuildTree($rightSubset.X, rightSubset.y, depth+1$)
    else
        for each unique value $v$ in $feature$ do
            $node.children[v] \leftarrow$ BuildTree($subsetForValue(v).X, subsetForValue(v).y, depth+1$)
        end for
    end if
    return $node$
end function
```

3.1.3 Information Theory Foundation

The algorithm uses information gain to determine the best feature for splitting at each node. The mathematical foundation relies on entropy as a measure of impurity:

1. **Entropy**

   The entropy of a set S with c different classes is defined as:

$$\text{Entropy}(S) = -\sum_{i=1}^{c} p_i \log_2(p_i) \tag{1}$$

Where p_i is the proportion of examples in class i. This is implemented in the _entropy method:

```
def _entropy(self, y: Union[pd.Series, np.ndarray]) -> float:
    if isinstance(y, pd.Series):
        y = y.value_counts(normalize=True)
    else:
        y = pd.Series(y).value_counts(normalize=True)

    return -np.sum(y * np.log2(y + 1e-9))
```

The small constant $1e{-9}$ is added to avoid numerical issues with $\log(0)$.

2. **Information Gain**

   Information gain measures the reduction in entropy achieved by splitting on a particular feature:

$$\text{Gain}(S, A) = \text{Entropy}(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} \text{Entropy}(S_v) \tag{2}$$

   Where S is the parent set, A is the feature, and S_v is the subset of S where feature A has value v. This is implemented in the _information_gain method:

```
def _information_gain(self, parent_y: Union[pd.Series, np.ndarray],
                      *child_y: List[Union[pd.Series, np.ndarray]]) -> float:
    parent_entropy = self._entropy(parent_y)
    child_entropy = sum((len(child) / len(parent_y)) * self._entropy(child)
                        for child in child_y)

    return parent_entropy - child_entropy
```

3.1.4 Feature Type Handling

The implementation distinguishes between categorical and continuous features:

1. **Continuous Features**

   For continuous features, the algorithm finds the optimal threshold by:

   (a) Sorting feature values
   (b) Calculating midpoints between adjacent unique values
   (c) Evaluating information gain for each potential split point
   (d) Selecting the split point that maximizes information gain

   This is implemented in the `_best_split_index_with_gain` method:

```
def _best_split_index_with_gain(self, X: pd.Series, y: Union[pd.Series, np.nd
                              -> Tuple[float, float]:
    # Sort values and handle missing data
    sorted_indices = np.argsort(X_clean)
    X_sorted = X_clean.iloc[sorted_indices].values
    y_sorted = y_clean.iloc[sorted_indices].values

    # Calculate midpoints between adjacent unique values
    midpoints = []
    for i in range(len(X_sorted) - 1):
        if X_sorted[i] != X_sorted[i + 1]:
            midpoint = (X_sorted[i] + X_sorted[i + 1]) / 2
            midpoints.append(midpoint)

    # Evaluate information gain for each midpoint
    gains = []
    for midpoint in midpoints:
        left_mask = X_sorted <= midpoint
        right_mask = X_sorted > midpoint

        if np.any(left_mask) and np.any(right_mask):
            gain = self._information_gain(y_sorted,
                                           y_sorted[left_mask],
                                        y_sorted[right_mask])
            gains.append(gain)
        else:
            gains.append(0.0)

    # Return the best midpoint and corresponding gain
    best_gain_index = np.argmax(gains)
    return midpoints[best_gain_index], np.max(gains)
```

2. **Categorical Features**

   For categorical features, the algorithm:

   (a) Creates one branch for each unique feature value
   (b) Calculates information gain across all resulting subsets

3.1.5 Feature Detection

The implementation automatically detects continuous features using the `is_continuous` method:

$$\text{is\_continuous}(X) = \begin{cases} \text{True} & \text{if } X \text{ is numeric and } \frac{|\text{unique}(X)|}{|X|} > 0.001 \\ \text{False} & \text{otherwise} \end{cases} \tag{3}$$

This heuristic classifies a feature as continuous if it's numeric and has more than 0.1% unique values relative to its length.

3.1.6 Prediction Process

Predictions are made by traversing the tree from root to leaf:

Algorithm 2 Predict Class

```
1: function PREDICTROW($node, row$)
2:     if $node$ is leaf then
3:         return $node.label$
4:     end if
5:     $featureValue \leftarrow row[node.feature]$
6:     if $featureValue$ is missing then
7:         return None
8:     end if
9:     if $node.feature$ is continuous then
10:        if $featureValue \leq node.value$ then
11:            $childNode \leftarrow node.children["left"]$
12:        else
13:            $childNode \leftarrow node.children["right"]$
14:        end if
15:    else
16:        $childNode \leftarrow node.children[featureValue]$
17:    end if
18:    if $childNode$ is None then
19:        return None
20:    end if
21:    return PREDICTROW($childNode, row$)
22: end function
```

The `predict` method handles missing values and uncertain paths by falling back to the most common class:

```python
def predict(self, X: Union[pd.DataFrame, np.ndarray]) -> np.ndarray:
    predictions = []
    for _, row in X.iterrows():
        prediction = self._predict_row(self.tree, row)
        # If prediction is None, use the default class
        if prediction is None:
            prediction = self.default_class
        predictions.append(prediction)

    return np.array(predictions)
```

3.1.7 Stopping Criteria

The tree building process stops when one of the following conditions is met:

- All samples in the node belong to the same class
- Maximum depth is reached (if specified)
- No feature provides any information gain
- The node has no samples

3.1.8 Complexity Control

Overfitting is controlled through the `max_depth` parameter, which limits the maximum depth of the tree:

```
def __init__(self, max_depth=None):
    self.max_depth = max_depth
    self.tree = None
    self.default_class = None  # Store most common class as fallback
```

3.1.9 Probability Estimation

The `predict_proba` method returns class probabilities:

```
def predict_proba(self, X: Union[pd.DataFrame, np.ndarray]) -> np.ndarray:
    # For each sample, create a one-hot encoded probability array
    # (1.0 for predicted class, 0.0 for others)
    probabilities = []
    for _, row in X.iterrows():
        prediction = self._predict_row(self.tree, row)
        if prediction is None:
            prediction = self.default_class

        # Create probability array
        prob_row = np.zeros(len(classes))
        if prediction in classes:
            class_idx = classes.index(prediction)
            prob_row[class_idx] = 1.0
        probabilities.append(prob_row)

    return np.array(probabilities)
```

This implementation returns "hard" probabilities (0 or 1) rather than calibrated probability estimates, which is typical for non-ensemble decision trees.

3.1.10 Implementation Considerations

The implementation includes several practical features:

- **Missing Value Handling**: Both during training and prediction
- **Input Validation**: Checks for correct data types and shapes
- **Fallback Mechanisms**: Uses the most common class when uncertain
- **Flexible Input Formats**: Handles pandas and numpy data structures

3.2 Naïve Bayes Classifier

3.2.1 Implementation Overview

The Naïve Bayes classifier is implemented from scratch with support for both discrete and continuous features. The implementation uses Laplace smoothing to handle zero probabilities and incorporates Gaussian distribution assumptions for continuous features.

3.2.2 Key Components

Initialization The classifier is initialized with a smoothing parameter α (default=1) to prevent zero probabilities:

```
def __init__(self, alpha: float = 1):
    self.alpha = alpha
```

Training Process The training process follows these steps:

Algorithm 3 Naïve Bayes Training

```
1: procedure Fit($X, y, \alpha$)
2:     Calculate class prior probabilities with Laplace smoothing
3:     for each class $c$ do
4:         for each feature $f$ do
5:             if $f$ is continuous then
6:                 Calculate mean and standard deviation of $f$ in class $c$
7:             else
8:                 Calculate conditional probabilities $P(f = v|c)$ with Laplace smoothing
9:             end if
10:        end for
11:    end for
12: end procedure
```

Prior Probability Calculation Class prior probabilities $P(c)$ are calculated with Laplace smoothing:

$$P(c) = \frac{count(c) + \alpha}{N + \alpha \cdot |C|} \tag{4}$$

where N is the total number of samples and $|C|$ is the number of classes.

Feature Type Detection The implementation automatically detects continuous features:

$$is\_continuous(X) = \begin{cases} \text{True}, & \text{if } X \text{ is numeric and } \frac{|unique(X)|}{|X|} > 0.001 \\ \text{False}, & \text{otherwise} \end{cases} \tag{5}$$

Discrete Feature Handling For categorical features, conditional probabilities are calculated with Laplace smoothing:

$$P(X_i = v|c) = \frac{count(X_i = v, c) + \alpha}{count(c) + \alpha \cdot |V_i|} \tag{6}$$

where $|V_i|$ is the number of unique values for feature i.

Continuous Feature Handling For continuous features, the implementation assumes a Gaussian distribution:

$$P(X_i|c) = \frac{1}{\sigma_{c,i}\sqrt{2\pi}} \exp\left(-\frac{(X_i - \mu_{c,i})^2}{2\sigma_{c,i}^2}\right) \tag{7}$$

where $\mu_{c,i}$ and $\sigma_{c,i}$ are the mean and standard deviation of feature i for class c.

Prediction Process The prediction process applies Bayes' theorem:

Algorithm 4 Naïve Bayes Prediction

```
1: procedure PredictProba($X$)
2:     for each sample $x$ in $X$ do
3:         for each class $c$ do
4:             $P(c|x) \leftarrow P(c)$                          ▷ Start with prior
5:             for each feature $i$ in $x$ do
6:                 if feature $i$ is continuous then
7:                     $P(c|x) \leftarrow P(c|x) \times P(x_i|c)$      ▷ Using Gaussian PDF
8:                 else
9:                     if value $x_i$ seen in training then
10:                        $P(c|x) \leftarrow P(c|x) \times P(x_i|c)$
11:                    else
12:                        $P(c|x) \leftarrow P(c|x) \times 0$          ▷ Unseen value
13:                    end if
14:                end if
15:            end for
16:        end for
17:        Normalize probabilities to sum to 1
18:    end for
19:    return normalized probabilities
20: end procedure
```

Probability Normalization After calculating the unnormalized posterior probabilities, they are normalized to sum to 1:

$$P(c|x) = \frac{P(c)\prod_i P(x_i|c)}{\sum_{c' \in C} P(c')\prod_i P(x_i|c')} \tag{8}$$

3.2.3 Implementation Details

Gaussian Probability Density Function The Gaussian PDF is calculated using:

$$f(x; \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{9}$$

Special handling is included for zero standard deviation:

```
def _gaussian_pdf(self, x: float, mean: float, std: float) -> float:
    if std == 0:
        return 0.0
    coeff = 1 / (std * np.sqrt(2 * np.pi))
    exponent = -((x - mean) ** 2) / (2 * std ** 2)
    return coeff * np.exp(exponent)
```

Data Structure The conditional probabilities are stored in a nested dictionary structure:

- First level: Class labels
- Second level: Feature names
- Third level:
  - For discrete features: Feature values mapped to probabilities
  - For continuous features: Mean and standard deviation

Edge Case Handling The implementation includes mechanisms for:

- **Zero probabilities**: Laplace smoothing prevents zero probabilities for seen feature values
- **Unseen values**: Feature values not seen during training get zero probability
- **Zero standard deviation**: Returns 0 for the Gaussian PDF when standard deviation is 0
- **Numerical stability**: Normalizes probabilities to prevent underflow

Interface The implementation provides a scikit-learn-like interface:

- `fit(X, y)`: Trains the model
- `predict(X)`: Returns predicted class labels
- `predict_proba(X)`: Returns class probabilities

3.3 Comparison Analysis

The comparative analysis of Decision Tree and Naive Bayes classifiers was conducted using a robust evaluation methodology to ensure reliable performance assessment. For each dataset, the following approach was implemented:

1. **Multiple Train-Test Splits**: Rather than relying on a single data partition, we performed 5 independent train-test splits using different random seeds (42, 43, 44, 45, and 46).
2. **Consistent Stratification**: Each split maintained the class distribution through stratified sampling, with 80% of data allocated for training and 20% for testing.
3. **Performance Tracking**: For each split, we recorded accuracy, precision, recall, F1-score, AUC, and training/inference times for both classifiers.
4. **Average Split Selection**: For the final comparison, we selected each algorithm's average-performing split based on accuracy metrics. This approach provides a fair assessment of each classifier's potential performance.
5. **Comprehensive Evaluation**: Beyond accuracy, we analyzed confusion matrices, ROC curves, and class-specific metrics across datasets to provide insights into classifier behavior.

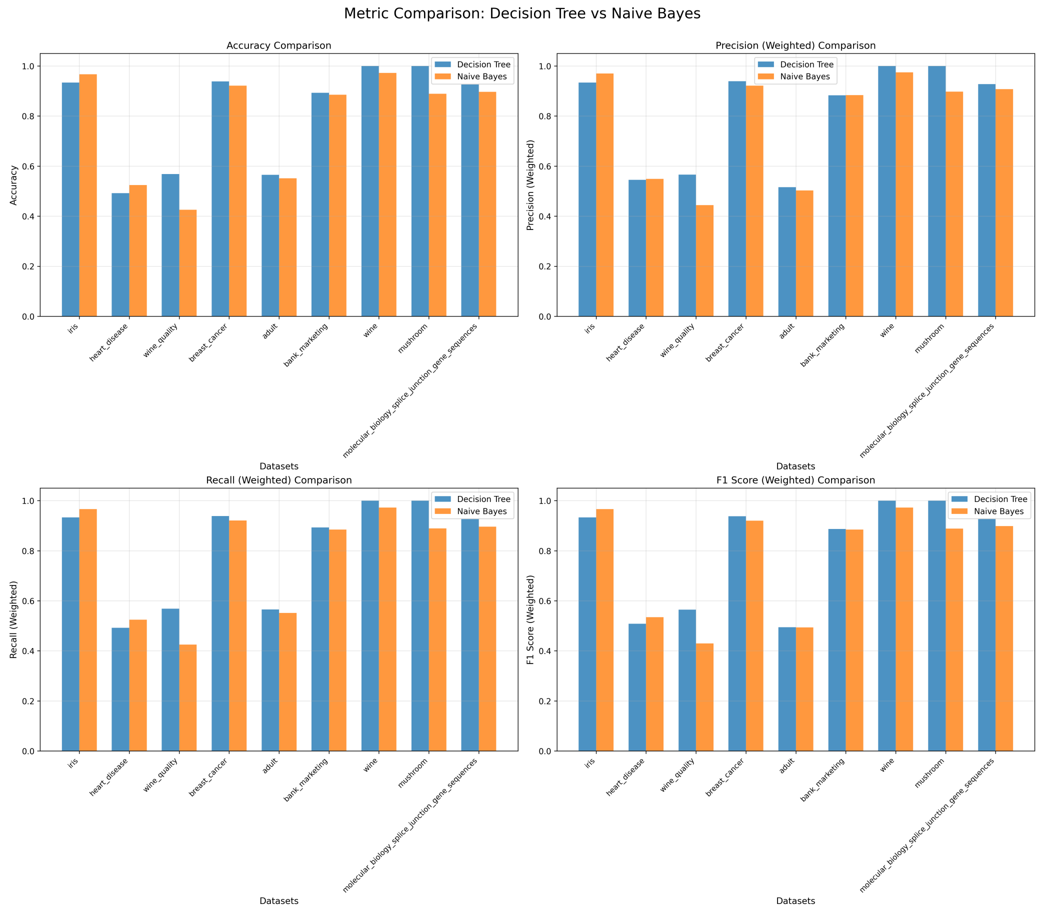
This methodology addresses the variability introduced by random train-test partitioning, offering a more stable and representative performance assessment than single-split evaluations. By testing across multiple data splits while maintaining identical preprocessing steps, we can better isolate and compare the intrinsic characteristics of the Decision Tree and Naive Bayes algorithms.

4 Hardware Specification

This experiment were run on a Laptop with 2.30 GHz Intel(R) Core(TM) i7 - 11800H (11th Gen) processor and 16Gbyte Ram. The operating system was Windows 11.

5 EXPERIMENTAL RESULT

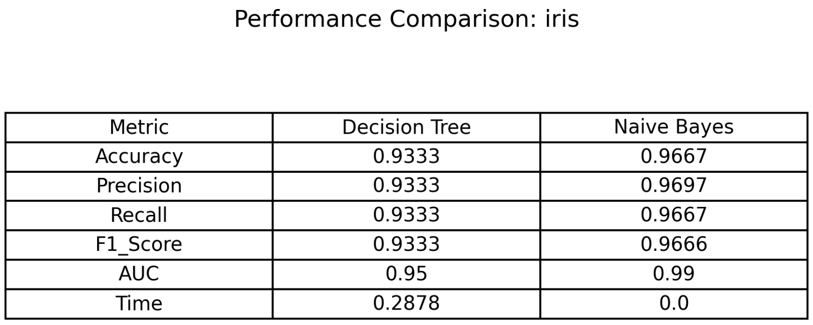
Here I present the experimental result of the implementation of Decision Tree and Naïve Bayes Classifier.



(a) Comparison Graph

Dataset: Iris

Feature Type: Continuous

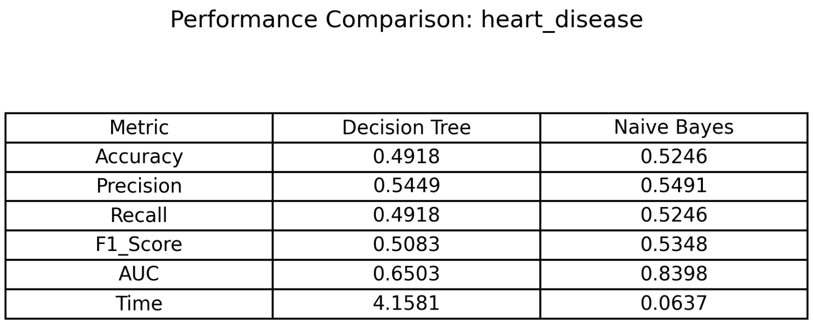


Performance Comparison: iris

Metric	Decision Tree	Naive Bayes
Accuracy	0.9333	0.9667
Precision	0.9333	0.9697
Recall	0.9333	0.9667
F1_Score	0.9333	0.9666
AUC	0.95	0.99
Time	0.2878	0.0

Dataset: Heart Disease

Feature Type: Categorical

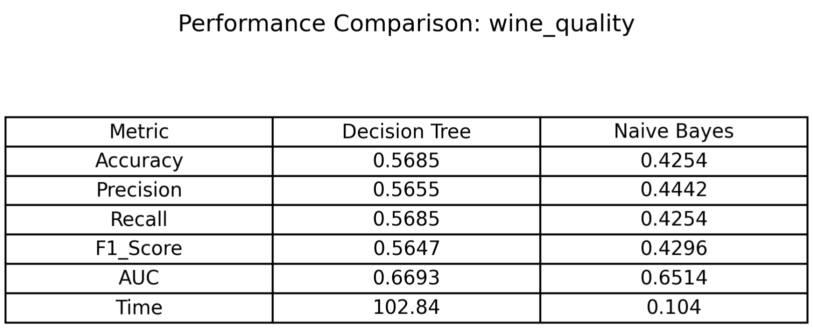


Performance Comparison: heart_disease

Metric	Decision Tree	Naive Bayes
Accuracy	0.4918	0.5246
Precision	0.5449	0.5491
Recall	0.4918	0.5246
F1_Score	0.5083	0.5348
AUC	0.6503	0.8398
Time	4.1581	0.0637

Dataset: Wine Quality

Feature Type: Continuous

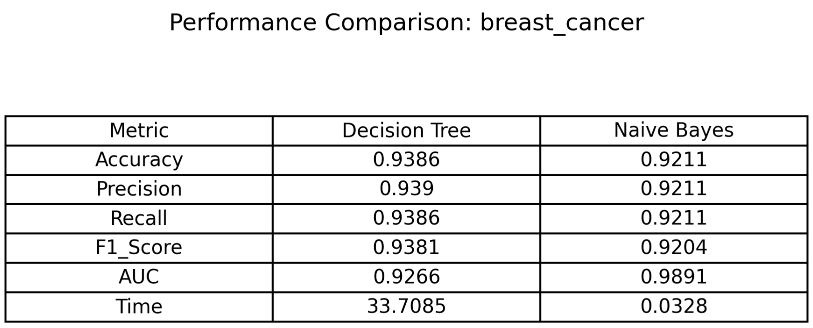


Performance Comparison: wine_quality

Metric	Decision Tree	Naive Bayes
Accuracy	0.5685	0.4254
Precision	0.5655	0.4442
Recall	0.5685	0.4254
F1_Score	0.5647	0.4296
AUC	0.6693	0.6514
Time	102.84	0.104

Dataset: Breast Cancer

Feature Type: Categorical

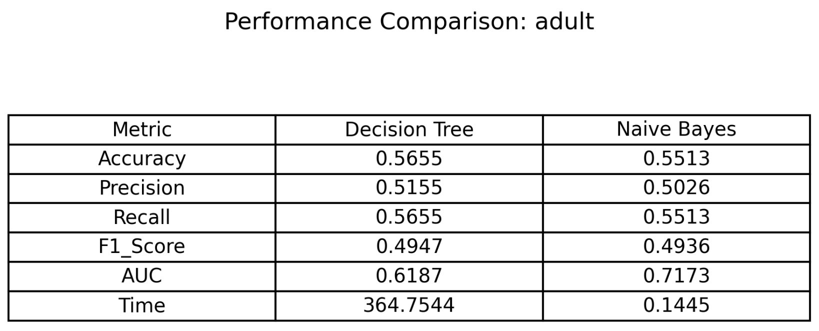


Performance Comparison: breast_cancer

Metric	Decision Tree	Naive Bayes
Accuracy	0.9386	0.9211
Precision	0.939	0.9211
Recall	0.9386	0.9211
F1_Score	0.9381	0.9204
AUC	0.9266	0.9891
Time	33.7085	0.0328

Dataset: Adult

Feature Type: Categorical

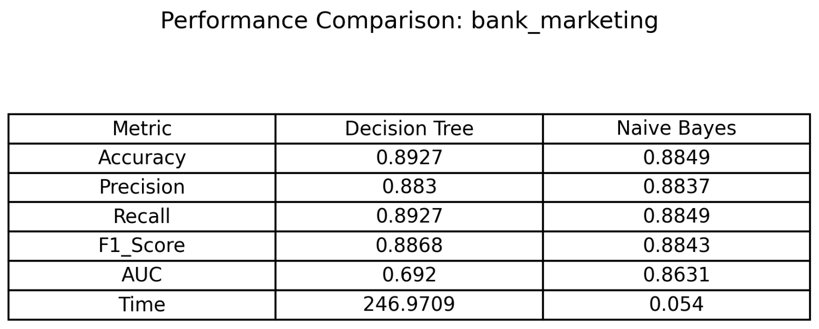


Performance Comparison: adult

Metric	Decision Tree	Naive Bayes
Accuracy	0.5655	0.5513
Precision	0.5155	0.5026
Recall	0.5655	0.5513
F1_Score	0.4947	0.4936
AUC	0.6187	0.7173
Time	364.7544	0.1445

Dataset: Bank Marketing

Feature Type: Categorical



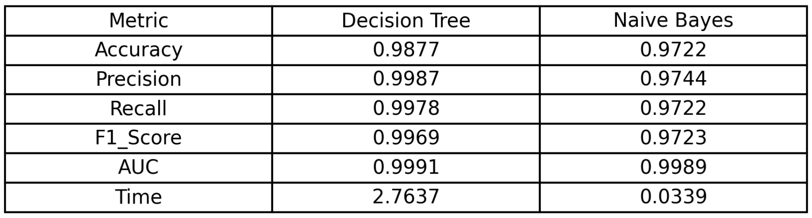
Performance Comparison: bank_marketing

Metric	Decision Tree	Naive Bayes
Accuracy	0.8927	0.8849
Precision	0.883	0.8837
Recall	0.8927	0.8849
F1_Score	0.8868	0.8843
AUC	0.692	0.8631
Time	246.9709	0.054

Dataset: Wine

Feature Type: Continuous

Performance Comparison: wine

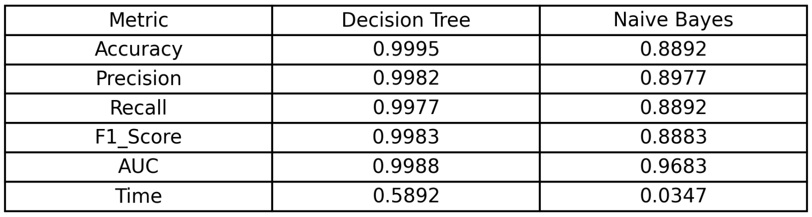


Metric	Decision Tree	Naive Bayes
Accuracy	0.9877	0.9722
Precision	0.9987	0.9744
Recall	0.9978	0.9722
F1_Score	0.9969	0.9723
AUC	0.9991	0.9989
Time	2.7637	0.0339

Dataset: Mushroom

Feature Type: Categorical

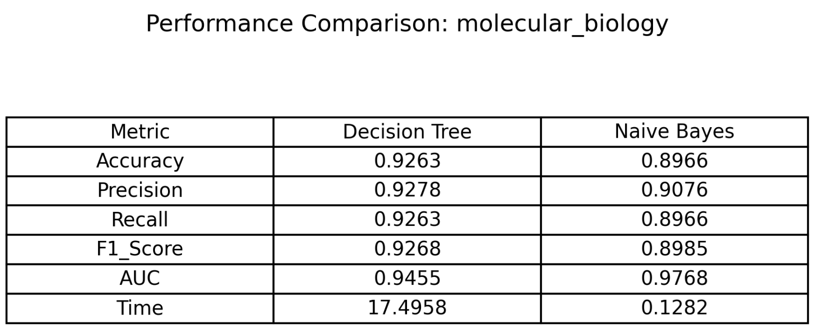
Performance Comparison: mushroom



Metric	Decision Tree	Naive Bayes
Accuracy	0.9995	0.8892
Precision	0.9982	0.8977
Recall	0.9977	0.8892
F1_Score	0.9983	0.8883
AUC	0.9988	0.9683
Time	0.5892	0.0347

Dataset: Molecular Biology Splice Junction Gene Sequences

Feature Type: Categorical


Performance Comparison: molecular_biology

Metric	Decision Tree	Naive Bayes
Accuracy	0.9263	0.8966
Precision	0.9278	0.9076
Recall	0.9263	0.8966
F1_Score	0.9268	0.8985
AUC	0.9455	0.9768
Time	17.4958	0.1282

6 CONCLUSION

In conclusion, the comparative analysis highlights that while both Decision Tree and Naive Bayes have their merits, Decision Tree consistently outperforms in core classification metrics such as accuracy, precision, recall, and F1-score, making it a more reliable and effective choice across diverse datasets. Although Naive Bayes excels in AUC and computational speed, its trade-offs in overall predictive performance position Decision Tree as the preferred algorithm for most practical scenarios.

References

[1] I. Rish,
An Empirical Study of the Naive Bayes Classifier,
In *IJCAI 2001 Workshop on Empirical Methods in AI*, pp. 41–46, 2001.
Available at: `https://www.scirp.org/journal/paperinformation?paperid=125884`

[2] J. R. Quinlan,
Induction of Decision Trees,
In *Machine Learning*, vol. 1, no. 1, pp. 81–106, 1986.
doi: `https://doi.org/10.1007/BF00116251`