VIETNAM INTERNATIONAL UNIVERSITY - HO CHI MINH CITY

INTERNATIONAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

DATA MINING COURSE:

Programming Assignment

Student’s name	Student’s ID
Phạm Lê Đức Thịnh	ITDSIU20085
Nguyễn Thanh Bình	ITDSIU20056
Ung Thị Hoài Thương	ITDSIU20028
Võ Thị Ngọc Thảo	ITDSIU20083
Trương Quân Bảo	ITDSIU20120
Nguyễn Phương Minh Ngọc	ITDSIU18045

TABLE OF CONTENT

I. Introduction

In recent years, e-commerce platforms have emerged as a significant disruptor, transforming traditional physical transactions into online interactions. This shift is particularly noticeable in the developing world, where entrepreneurs are leveraging these platforms to optimize their businesses, accelerate development, and access new market segments.

Understanding how markets operate is a critical component of planning an effective business model. The supply and demand theory posits that price is the primary factor driving the quantity sold, assuming other variables remain constant. Therefore, examining how markets interact to achieve maximum sales and profit is essential.

Given this context, our group has selected the dataset "Sales of Summer Clothes in E-commerce Wish" for our project. This dataset comprises 43 columns, containing product listings, product ratings, and sales performance metrics.

title	title_orig	price	retail_price	currency_buyer	units_sold	uses_ad_boosts	rating	rating_count	rating_five_count	rating_four_count
2020 Summer Vintaç	2020 Summer Vintaç	16	14	EUR	100	0	3.76	54	26	8
SSHOUSE Summer	Women's Casual Sun	8	22	EUR	20000	1	3.45	6135	2269	1027
2020 Nouvelle Arrivé	2020 New Arrival Wo	8	43	EUR	100	0	3.57	14	5	4
Hot Summer Cool T-	Hot Summer Cool T	8	8	EUR	5000	1	4.03	579	295	119
Femmes Shorts d'été	Women Summer Shc	2.72	3	EUR	100	1	3.1	20	6	4
Plus la taille d'été fe	Plus Size Summer W	3.92	9	EUR	10	0	5	1	1	0
Women Fashion Loo	Women Fashion Loo	7	6	EUR	50000	0	3.84	6742	3172	1352
Robe tunique ample	Women's Baggy Tun	12	11	EUR	1000	0	3.76	286	120	56
Robe d'été décontra	Women's Summer Ca	11	84	EUR	100	1	3.47	15	6	2
Femmes d'été, plus l	Summer Women Plu	5.78	22	EUR	5000	0	3.6	687	287	128
Femmes Mode Été L	Women's Fashion Su	5.79	5	EUR	1000	0	3.46	613	245	101
Été Sexy Femmes M	Summer Sexy Wome	6	8	EUR	100	1	3.31	13	3	4
Shorts de causalité c	New Women's Summ	1.91	6	EUR	1000	1	3.45	141	49	29
Mode féminine sans	Women Fashion Slee	5.79	42	EUR	1000	0	3.32	121	36	24
2019 Summer Wome	2019 Summer Wome	2	2	EUR	20000	1	3.65	2457	984	481
Mode d'été Flare Sle	Summer Fashion Fla	11	81	EUR	1000	0	3.92	426	204	94
Nouvelle mode d'été	New Summer Womei	11	10	EUR	10000	0	3.72	2058	840	435

Figure 1.1: The overview of our dataset.

II. Data Pre – Processing

a. *Data Cleaning*

- Step 1: Dropping Unnecessary Columns from the Dataset

  In this step, we will drop unnecessary columns from a dataset. This involves reading the CSV file, identifying, and removing the columns that are

```java
// Define columns to drop
Set<String> columnsToDrop = new HashSet<>(Arrays.asList(
    "title", "title_orig", "currency_buyer", "shipping_option_name", "urgency_text",
    "merchant_title", "merchant_name", "merchant_info_subtitle", "merchant_id",
    "merchant_profile_picture", "product_url", "product_picture", "product_id",
    "tags", "has_urgency_banner", "theme", "crawl_month", "origin_country"
));
```

Figure 2.1: the code for first step.

- Step 2: Removing Duplicate Columns from the Dataset

  In the second step of our data preprocessing pipeline, we aim to further refine the dataset by eliminating any duplicate columns. Duplicate columns can occur due to data entry errors, merging datasets, or other inconsistencies and can lead to redundant information that complicates data analysis.

```java
        // Read and collect data for each relevant column
        String[] row;
        while ((row = reader.readNext()) != null) {
            List<String> filteredRow = new ArrayList<>();
            for (int index : indicesToKeep) {
                filteredRow.add(row[index]);
                columnData.computeIfAbsent(index, k -> new ArrayList<>()).add(row[index]);
            }
            uniqueRows.add(filteredRow);  // Add row to set to prevent duplicates
        }
```

Figure 2.2: The code for the second step.

- Step 3: Handling Missing Values in the Dataset

  In the third step of our data preprocessing pipeline, we address missing values within the dataset. Missing values can negatively impact the performance of data analysis and modeling tasks. Therefore, it is crucial to handle them appropriately. Our approach involves:

  - Filling missing numerical values with the mean of their respective columns.
  - Filling missing categorical values with the most frequent value or, if a suitable mode cannot be determined, with the placeholder value 'Unknown'.

```java
    private static Map<Integer, String> calculateReplacements(Map<Integer, List<String>> columnData) {
        Map<Integer, String> replacements = new HashMap<>();
        columnData.forEach((index, data) -> {
            if (NumberUtils.isCreatable(data.get(0))) {
                double mean = data.stream().filter(StringUtils::isNotBlank)
                                    .mapToDouble(Double::parseDouble).average().orElse(0);
                replacements.put(index, String.format("%.2f", mean));
            } else {
                String mostCommon = data.stream().filter(StringUtils::isNotBlank)
                                    .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()))
                                    .entrySet().stream().max(Map.Entry.comparingByValue())
                                    .map(Map.Entry::getKey).orElse("Unknown");
                replacements.put(index, mostCommon);
            }
        });
        return replacements;
    }
```

```java
            // Fill missing values
            if (StringUtils.isBlank(value)) {
                value = replacements.get(index);
            }
```

Figure 2.3: The code for the third step.

- Step 4: Normalizing

In the fourth step of our data preprocessing pipeline, we will normalize the product_variation_size_id column. Normalization is essential to ensure consistency in data representation, especially for categorical variables that may have varied formats. For this specific column, normalization involves standardizing the values to a common format.

```java
private static String normalizeSize(String size) {
    if (size == null) return "Unknown"; // Handle null values
    size = size.trim().toUpperCase(); // Normalize case and remove whitespace

    Map<String, String> sizeMap = new HashMap<>();
    sizeMap.put("S", "S");
    sizeMap.put("SMALL", "S");
    sizeMap.put("SML", "S");
    sizeMap.put("XS", "XS");
    sizeMap.put("EXTRA SMALL", "XS");
    sizeMap.put("M", "M");
    sizeMap.put("MEDIUM", "M");
    sizeMap.put("MED", "M");
    sizeMap.put("L", "L");
    sizeMap.put("LARGE", "L");
    sizeMap.put("LRG", "L");
    sizeMap.put("XL", "XL");
    sizeMap.put("EXTRA LARGE", "XL");
    sizeMap.put("XXL", "XXL");
    sizeMap.put("2XL", "XXL");
    sizeMap.put("DOUBLE XL", "XXL");

    return sizeMap.getOrDefault(size, "Other");
}
```

Figure 2.4: The code for normalizing the product_variation_size_id column.

In the next section, we will encode the sizes into numerical values, facilitating easier manipulation and interpretation of the data

```java
//Encode normalized sizes
private static Map<String, Integer> encodeSizes() {
    Map<String, Integer> encodingMap = new HashMap<>();
    encodingMap.put("XS", 1);
    encodingMap.put("S", 2);
    encodingMap.put("M", 3);
    encodingMap.put("L", 4);
    encodingMap.put("XL", 5);
    encodingMap.put("XXL", 6);
    encodingMap.put("Unknown", 7);
    encodingMap.put("Other", 8); // Use for unexpected or new sizes

    return encodingMap;
}
```

Figure 2.5: The code for encoding the sizes into numerical values.

In the last section of this part, we will convert the currency values in a specific column to ensure consistency across the dataset.

```java
    private static double convertCurrency(double amount, String fromCurrency, String toCurrency) {
        return amount * 1.1;
    }
```

```java
            // Convert currency if needed
            if (headers[index].equals("price")) {
                value = String.valueOf(convertCurrency(Double.parseDouble(value), fromCurrency:"USD", toCurrency:"EUR"));
            }
```

Figure 2.6: The code for converting the currency values.

- Step 5: Normalizing color variations

  In this step of our data preprocessing pipeline, we will normalize the color variations in the dataset to ensure consistency. This involves several tasks:

  - Removing all non-alphanumeric characters in the product_color column.
  - Converting all values in the product_color column to lowercase.
  - Mapping compound color names to their base colors (e.g., ‘light blue’ to ‘blue’, ‘army green’ to ‘green’).
  - Use a hashmap to map specific color names to a unified standard color.
  - Divide product_color column into multiple columns with binary values.

```java
    private static String normalizeColor(String color) {
        // Remove all non-alphanumeric characters (keep only letters)
        String normalized = color.replaceAll("[^a-zA-Z0-9]", "").toLowerCase();

        if (colorMap.containsKey(normalized)) {
            return colorMap.get(normalized);
        }

        // Check for base color inclusion and map accordingly
        for (String baseColor : baseColors) {
            if (normalized.contains(baseColor)) {
                return baseColor;
            }
        }
        return normalized; // return as is if no base color is found
    }

    private static Set<String> initializeBaseColors() {
        return new HashSet<>(Arrays.asList(
            "blue", "green", "red", "yellow", "orange", "purple", "pink", "white", "black", "grey", "beige", "brown"
        ));
    }
```

Figure 2.7: The code for converting all values into lowercase.

```java
private static Map<String, String> initializeColorMap() {
    Map<String, String> map = new HashMap<>();
    // Detailed normalization of color variations
    map.put("navy", "blue");
    map.put("khaki", "beige");
    map.put("lightkhaki", "beige");
    map.put("nude", "beige");
    map.put("tan", "beige");
    map.put("apricot", "orange");
    map.put("cream", "white");
    map.put("ivory", "white");
    map.put("camouflage", "green");
    map.put("army", "green");
    map.put("gold", "yellow");
    map.put("star", "yellow");
    map.put("claret", "red");
    map.put("wine", "red");
    map.put("burgundy", "red");
    map.put("lightgray", "grey");
    map.put("silver", "grey");
    map.put("violet", "purple");
    map.put("rosegold", "pink");
    map.put("rose", "pink");
    map.put("leopardprint", "multicolor");
    map.put("leopard", "multicolor");
    map.put("jasper", "multicolor");
    map.put("rainbow", "multicolor");
    map.put("floral", "multicolor");
    map.put("camel", "brown");
    map.put("coffee", "brown");

    return map;
}
```

Figure 2.8: The code for hashmap.

```java
// Encode data
for (int i = 1; i < data.size(); i++) {
    List<String> newRow = new ArrayList<>(Arrays.asList(data.get(i)));
    String value = newRow.remove(columnIndex);
    uniqueValues.forEach(uniqueValue -> {
        newRow.add(value.equals(uniqueValue) ? "1" : "0");
    });
    result.add(newRow.toArray(new String[0]));
}
```

Figure 2.9 The code to convert into Binary value.

After completing all the steps in our data preprocessing pipeline, we obtain a new, clean dataset.

price	retail_price	units_sold	uses_ad_b	rating	rating_cou	rating_five_	rating_four	rating_thre	rating_two_	rating_one_	badges_co	badge_loc	badge_pro	badge_fast	product_va	product_va	shipping_o	shipping_is	countries_	inventory_t	merchant_	merchant
8.8	75	50	0	4.33	3	1	2	0	0	0	0	0	0	0	2	50	3	0	41	50	4773	4.002933
12.1	59	100	1	4.33	3	2	0	1	0	0	0	0	0	0	2	50	3	0	41	50	38258	3.955434
8.8	7	100	0	3.44	63	23	14	8	4	14	0	0	0	0	2	50	2	0	40	50	5595	3.772654
9.9	8	5000	1	3.61	1926	740	405	345	164	272	0	0	0	0	6	50	3	0	41	50	73271	3.954443
1.87	9	1000	1	3.86	86	41	17	13	5	10	0	0	0	0	8	50	1	0	57	50	484	3.991736
14.3	11	20000	1	2.97	2191	603	321	343	263	661	0	0	0	0	3	34	3	0	35	50	14482	4.001588
6.6	56	1000	1	3.5	938	349	171	173	88	157	0	0	0	0	8	1	2	0	28	50	13309	3.8767
6.6	6	100	0	3.47	19	6	5	4	0	4	0	0	0	0	2	10	2	0	45	50	10600	3.867547
15.4	12	100	0	2.7	20	6	1	3	1	9	0	0	0	0	8	14	3	0	27	50	1643	3.922702
6.226	5	20000	0	4.14	2927	1646	564	389	145	183	0	0	0	0	2	14	2	0	29	50	8205	4.195856
15.4	93	1000	1	4.24	254	147	57	26	11	13	0	0	0	0	3	3	3	0	40	50	90105	4.201298
6.325	5	10	1	5	0	442.26	179.6	134.55	63.71	95.74	0	0	0	0	1	5	2	0	139	50	3	2.333333
7.7	6	100	0	4	6	3	1	1	1	0	0	0	0	0	1	7	2	0	24	50	1559	4.039128
8.8	7	1000	0	3.81	443	215	76	65	29	58	0	0	0	0	2	4	2	0	46	50	3460	3.870809
8.8	8	10000	1	3.61	2187	843	484	350	182	328	0	0	0	0	8	20	2	0	38	50	70773	4.03832
7.7	6	5000	0	3.23	1212	393	201	198	133	287	0	0	0	0	2	9	2	0	13	50	1686	3.687426
6.6	21	5000	0	3.18	573	174	96	97	69	137	0	0	0	0	2	13	2	0	34	50	9827	4.11387
1.826	2	1000	1	4.21	33	21	6	2	0	4	1	0	1	0	8	50	1	0	34	50	15221	4.393207
8.8	48	1000	0	4.34	794	500	152	86	26	30	1	0	1	0	1	50	2	0	9	50	3403	4.337349
6.6	22	1000	0	3.74	518	240	89	77	37	75	0	0	0	0	2	50	2	0	38	50	4233	4.016773
6.215	25	10000	0	3.65	418	180	74	65	36	63	0	0	0	0	8	50	1	0	43	50	23465	4.017601
4.4	4	5000	0	3.39	783	268	129	155	102	129	0	0	0	0	2	7	1	0	48	50	32168	3.884544
6.6	17	6	0	5	0	442.26	179.6	134.55	63.71	95.74	0	0	0	0	1	10	2	0	43	50	65189	4.04964

Figure 2.10: The new clean dataset.

b. *Data Analysis*

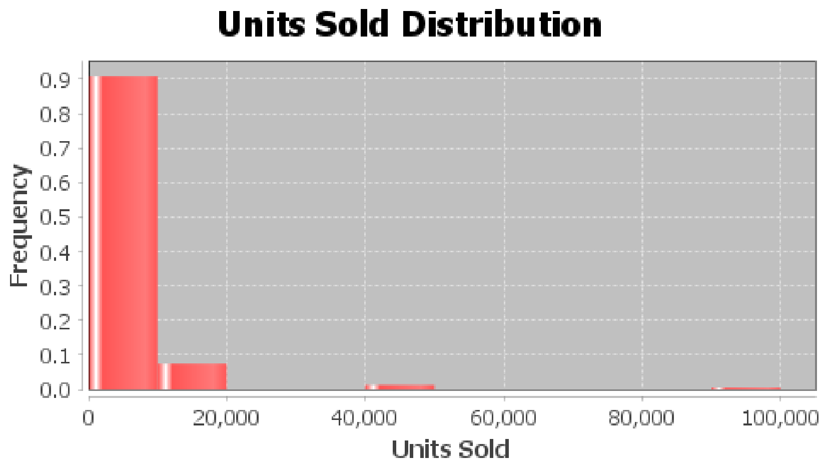


Figure 2.11: The bar chart of units sold distribution.

The "Units Sold Distribution" chart shows the frequency distribution of units sold across different products. Most products have a low number of units sold, with a significant concentration below 20,000 units. There are a few outliers with higher sales, but these are rare. This distribution indicates that while some products are highly popular, the majority of products have modest sales figures.

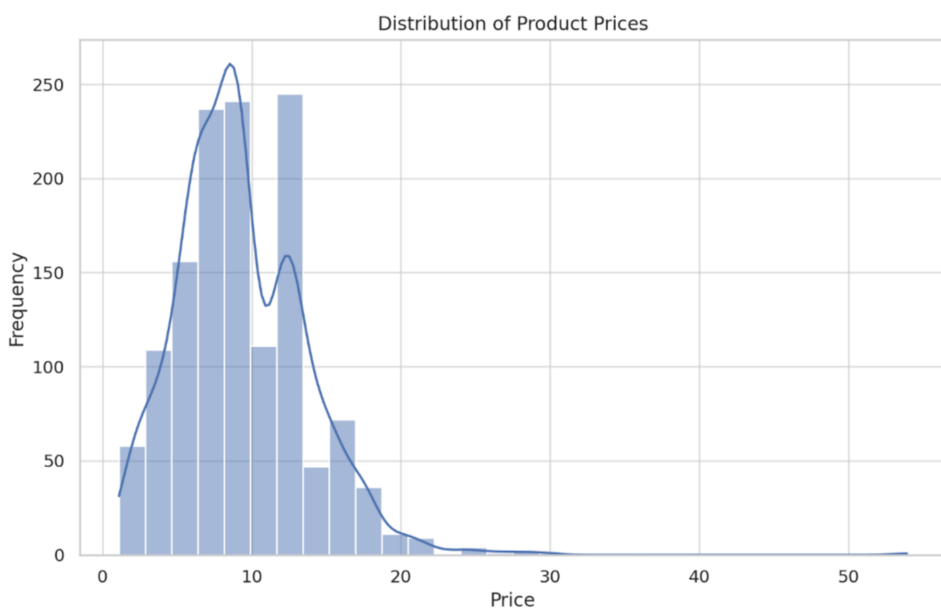


Figure 2.12: The bar chart rating distribution.

The histogram and kernel density estimate (KDE) plot illustrate the distribution of product prices, with a concentration of prices between 0 and 20 units and a noticeable peak around 10 units, indicating a common price point. The highest frequency exceeds 250 for prices slightly below 10 units, with additional significant frequencies around 5 and 15 units. The distribution is right-skewed, with fewer products priced above 20 units, and exhibits a bimodal characteristic with two peaks. The skewness and outliers suggest the presence of higher-priced or premium products, reflecting a varied pricing strategy within the dataset.

Figure 2.13: The boxplot of rating distribution across sales categories

The box plot illustrates the distribution of ratings across three sales categories: Slow, Stable, and Hot. Each category shows a different spread of ratings, with the median rating slightly decreasing from Slow to Hot categories. The Slow category exhibits the widest interquartile range, indicating a higher variability in ratings, and includes several outliers on the lower end. The Stable category has a more compact distribution with fewer outliers, suggesting consistent ratings around the median. The Hot category shows a narrow interquartile range similar to the Stable category but has a slightly lower median rating and more lower-end outliers. Overall, this analysis suggests that while highly-rated products exist across all categories, those in the Slow category display greater variability in ratings, whereas those in the Stable and Hot categories tend to have more consistent ratings with fewer extreme values.

III. Prediction algorithms.

The following section provides an in-depth examination of the implementation of the prediction of 'Unit_Sold` Java class, which leverages the Weka library to classify product ratings. The primary objective is to categorize products into three groups: "slow," "stable," and "hot," based on their attributes. To achieve this classification, various algorithms were evaluated, including Decision Tree, SMO, and Linear Regression. However, the two machine learning algorithms that demonstrated the highest accuracy were Naive Bayes and Random Forest. This class not only constructs and trains these models but also evaluates their performance to determine which model exhibits superior accuracy.

The initial step in the implementation of two algorithms involves loading the dataset. The primary purpose of this step is to import the dataset from the specified ARFF file into an Instances object. The Instances object serves as the main data structure in Weka for storing datasets, enabling subsequent data processing and analysis.

```
// Load dataset
String datasetPath = "C:/Users/Admin/Desktop/DATA MINING/Clothing-Sales-Prediction-on-E-commerce-main/data/cleaned_data.arff";
DataSource source = new DataSource(datasetPath);
Instances data = source.getDataSet();
```

Figure 3.1: The code for first step.

The subsequent step in the implementation involves calculating the quartiles to determine thresholds for categorizing sales performance. This process begins with sorting the dataset based on the units_sold attribute. By arranging the data in ascending order according to this attribute, the quartile values can be accurately determined.

Quartiles Q1 and Q3 are then calculated, representing the 25th and 75th percentiles, respectively. These quartile values serve as critical thresholds for categorizing the sales performance of the products. Specifically:

- Q1 (First Quartile): Represents the 25th percentile of the units_sold data, indicating the value below which 25% of the data falls.
- Q3 (Third Quartile): Represents the 75th percentile of the units_sold data, indicating the value below which 75% of the data falls.

```
// Calculate quartiles
data.sort(data.attribute(name:"units_sold"));
double Q1 = data.instance((int) (data.numInstances() * 0.25)).value(data.attribute(name:"units_sold"));
double Q3 = data.instance((int) (data.numInstances() * 0.75)).value(data.attribute(name:"units_sold"));
```

Figure 3.2: The code for second step.

In the third step, a new nominal attribute named units_sold_categories is created. This attribute is designed to classify the products into three distinct categories based on their sales performance: "Slow," "Stable," and "Hot." The creation of this attribute involves assigning each product to one of these categories according to the thresholds determined in the previous quartile calculation step.

```
// Create new nominal attribute with specified categories
List<String> labels = new ArrayList<>();
labels.add(e:"Slow");
labels.add(e:"Stable");
labels.add(e:"Hot");
Attribute attribute = new Attribute(attributeName:"units_sold_categories", labels);
```

Figure 3.3: The code for the third step.

In the fourth step, the newly created units_sold categories attribute is added to the dataset. Additionally, the class index of the dataset is updated to reference this new attribute. This ensures that the classification algorithms will use units_sold categories as the target variable for training and prediction purposes.

```
int unitsSoldIndex = data.attribute(name:"units_sold").index();
data.insertAttributeAt(attribute, unitsSoldIndex + 1);
data.setClassIndex(unitsSoldIndex + 1); // Update class index to new attribute
```

Figure 3.4: The code for fourth step.

In the fifth step, each instance in the dataset is categorized based on the value of the units_sold attribute, using the quartile thresholds previously calculated. The categorization criteria are as follows:

- "Slow": Instances with units_sold values less than or equal to Q1 (the 25th percentile).
- "Stable": Instances with units_sold values greater than Q1 but less than or equal to Q3 (the 75th percentile).
- "Hot": Instances with units_sold values greater than Q3.

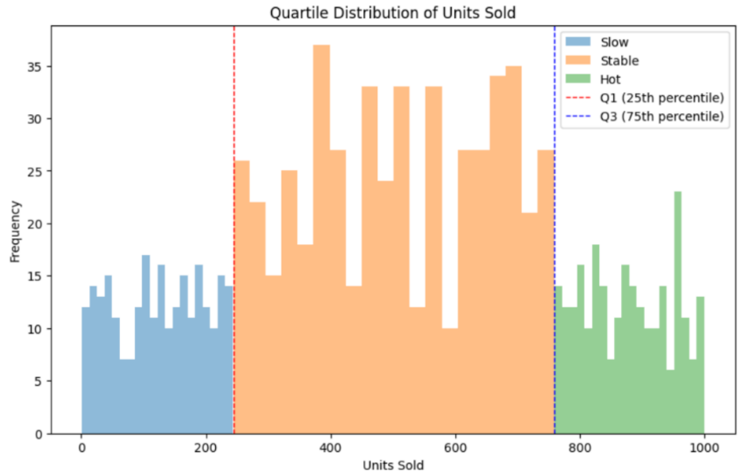


Figure 3.5: The quartile distribution.

```
// Assign values based on quartile data
for (int i = 0; i < data.numInstances(); i++) {
    double value = data.instance(i).value(unitsSoldIndex);
    String category = (value <= Q1) ? "Slow" : (value <= Q3) ? "Stable" : "Hot";
    data.instance(i).setValue(unitsSoldIndex + 1, category);
}
```

Figure 3.6: The code for the fifth step.

In the sixth step, the original units_sold attribute is removed from the dataset. This step ensures that the dataset is streamlined for analysis, containing only the relevant categorical attribute units_sold categories. By eliminating the original numeric attribute, the focus is placed on the newly created categorical classifications, which are essential for the subsequent predictive modeling and analysis.

```
data.deleteAttributeAt(unitsSoldIndex); // Remove the original 'units_sold' attribute
```

Figure 3.7: The code for sixth step.

In the seventh step, the dataset is randomized and subsequently divided into three subsets to facilitate model training, evaluation, and validation:

- Training Set (80%): This subset, comprising 80% of the dataset, is utilized to train the Naive Bayes classifier.
- Test Set (20%): This subset, comprising 20% of the dataset, is used to evaluate the performance of the trained classifier.
- Validation Set (10 instances): This subset, consisting of 10 instances, is employed to provide predictions and assess the model's generalization

```
// Split data into training and testing
data.randomize(new Random(seed:1));
Instances train = new Instances(data, first:0, (int) (data.numInstances() * 0.8));
Instances test = new Instances(data, (int) (data.numInstances() * 0.8),
        (data.numInstances() - (int) (data.numInstances() * 0.8) - 10));
Instances validation = new Instances(data, data.numInstances() - 10, toCopy:10); // Last 10 records for validation
```

Figure 3.8: The code for seventh step.

In the final step, we implement the prediction algorithms for both the Random Forest and Naive Bayes classifiers. This involves the following processes:

- Naïve Bayes: the Naive Bayes classifier is trained on the training set. Post-training, the model is employed to predict the categories of the instances in the test and validation sets. The performance of the Naive Bayes model is assessed to determine its accuracy and effectiveness.

```
// Configure and build the Naive Bayes classifier
NaiveBayes nb = new NaiveBayes();
nb.buildClassifier(train);
```

Figure 3.8: The code for Navie Bayes algorithms.

- Random Forest: To enhance accuracy, The Random Forest classifier is trained on the training set. Once trained, the model is used to predict the categories of the instances in the test and validation sets. The performance of the Random Forest model is then evaluated based on its accuracy and other relevant metrics.

```
// Configure and build the Random Forest
RandomForest forest = new RandomForest()
String[] options = new String[2];
options[0] = "-I"; // number of trees
options[1] = "100"; // trees count
forest.setOptions(options);
forest.buildClassifier(train);
```

Figure 3.9: The code .to construct Random Forest.

IV. Model Evaluation.

In this section, we assess the performance of our classification models, specifically Random Forest and Naive Bayes, employing 10-fold cross-validation. The evaluation criteria encompass accuracy, Kappa statistic, mean absolute error, root mean squared error, and runtime for both model construction and prediction and the evaluation metrics are:

- Accuracy: This metric represents the proportion of correctly classified instances relative to the total number of instances. It provides an overall effectiveness of the model in terms of classification performance.

- Kappa Statistic: The Kappa statistic measures the agreement between predicted and actual classifications, taking into account the possibility of agreement occurring by chance. It is a more robust measure than simple accuracy, particularly for imbalanced datasets.
- Mean Absolute Error (MAE): MAE is the average of the absolute differences between predicted and actual values. It gives an indication of the average magnitude of errors in the predictions, without considering their direction.
- Root Mean Squared Error (RMSE): RMSE is the square root of the average of the squared differences between predicted and actual values. It penalizes larger errors more significantly than MAE and provides a measure of the model's prediction accuracy.
- Runtime: This metric denotes the time taken to build the model and make predictions. It is a critical factor in evaluating the efficiency and practicality of the model in real-time applications.

a. *Naïve Bayes evaluation results.*

The Naïve Bayes algorithm trained and evaluated using 10-fold cross-validation, achieved a runtime of 15 seconds for both training and prediction. The results are:

```
Results
======

Correctly Classified Instances        211              81.4672 %
Incorrectly Classified Instances       48              18.5328 %
Kappa statistic                         0.7106
Mean absolute error                     0.1219
Root mean squared error                 0.3347
Relative absolute error                28.4354 %
Root relative squared error            72.3489 %
Total Number of Instances             259

Confusion Matrix:
88.0 8.0 0.0
29.0 78.0 2.0
0.0 9.0 45.0
```

```
Validation Set Predictions:
Actual: Slow, Predicted: Stable
Actual: Stable, Predicted: Stable
Actual: Stable, Predicted: Slow
Actual: Slow, Predicted: Slow
Actual: Stable, Predicted: Stable
Actual: Stable, Predicted: Stable
Actual: Hot, Predicted: Hot
Actual: Slow, Predicted: Slow
Actual: Slow, Predicted: Slow
Actual: Stable, Predicted: Stable
```

Figure 4.1: The Naïve Bayes result.

b. *Random Forest evaluation results.*

The Random Forest algorithm trained and evaluated using 10-fold cross-validation, achieved a runtime of 15 seconds for both training and prediction. The results are :

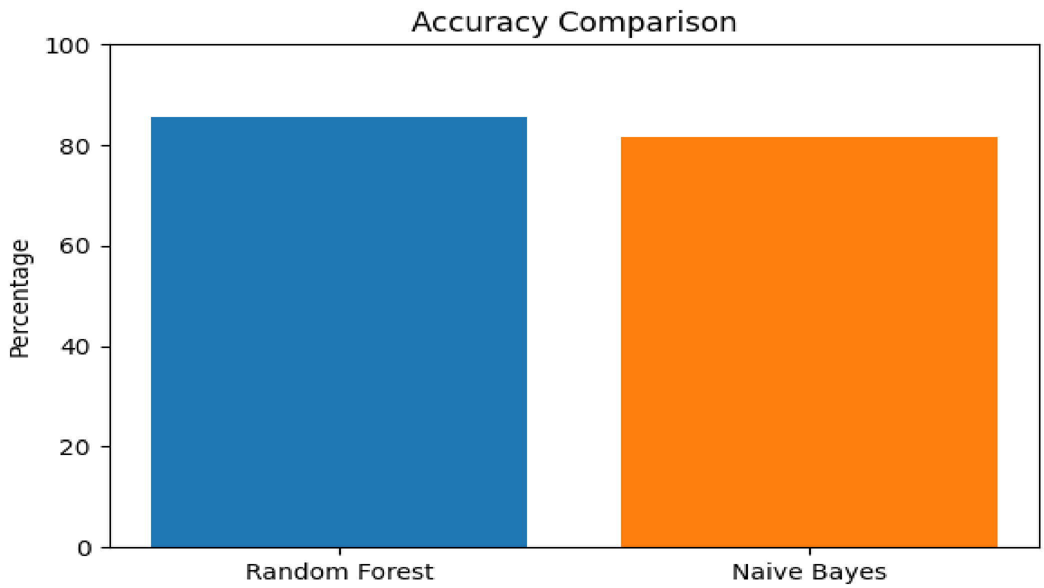
```
Results
======

Correctly Classified Instances         222               85.7143 %
Incorrectly Classified Instances        37               14.2857 %
Kappa statistic                          0.7777
Mean absolute error                      0.1194
Root mean squared error                  0.2444
Relative absolute error                 27.8519 %
Root relative squared error             52.8213 %
Total Number of Instances              259

Confusion Matrix:
=== Confusion Matrix ===

  a  b  c   <-- classified as
 89  7  0 |  a = Slow
 15 87  7 |  b = Stable
  0  8 46 |  c = Hot

Validation Set Predictions:
Actual: Slow, Predicted: Slow
Actual: Stable, Predicted: Stable
Actual: Stable, Predicted: Stable
Actual: Slow, Predicted: Slow
Actual: Stable, Predicted: Stable
Actual: Stable, Predicted: Stable
Actual: Hot, Predicted: Hot
Actual: Slow, Predicted: Slow
Actual: Slow, Predicted: Slow
Actual: Stable, Predicted: Hot
```

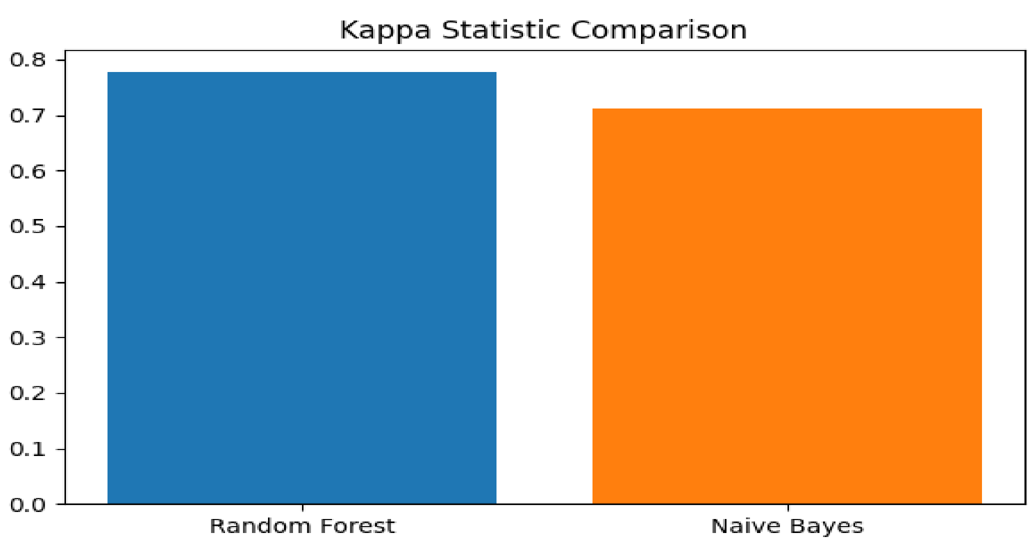
Figure 4.2: The Random Forest result.

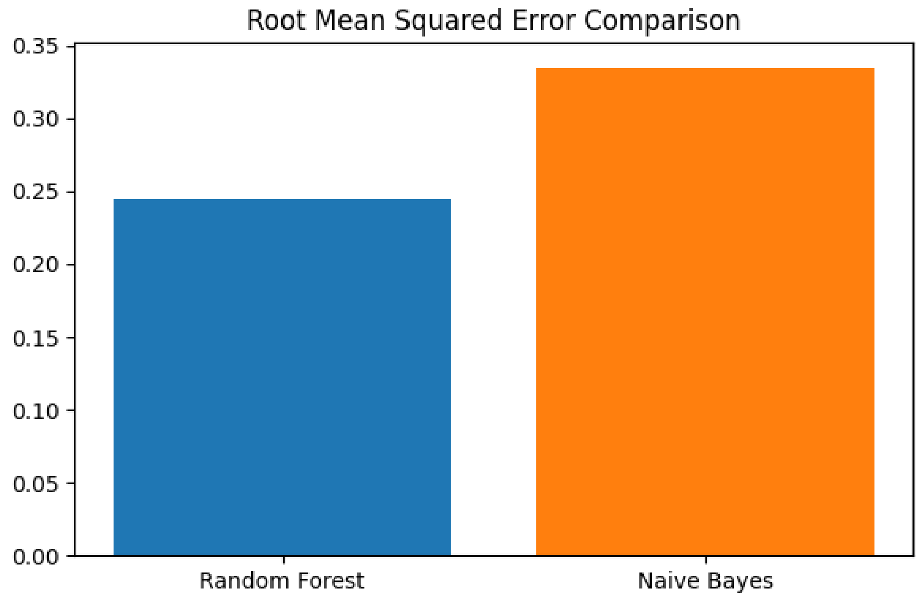
c. <u>Comparison</u>

The Random Forest model outperformed the Naive Bayes model in all the key metrics. The accuracy of the Random Forest was higher, and it showed better agreement between predicted and actual classifications as indicated by the Kappa statistic. Additionally, the Random Forest model had lower error rates (MAE and RMSE). However, the Naive Bayes model had a significantly shorter runtime for both training and prediction.

The following charts provide a visual representation of the performance metrics for both models:







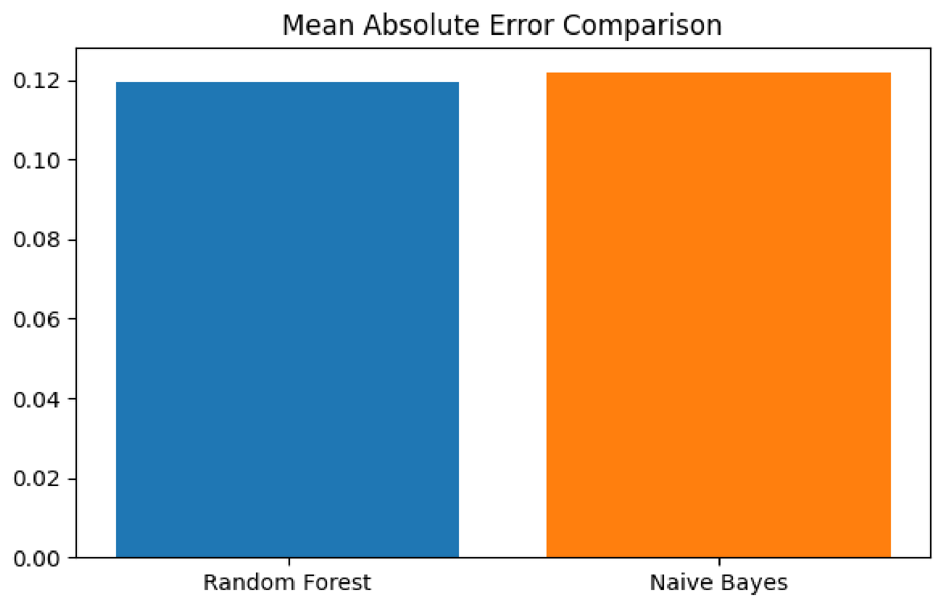


Figure 4.3: The comparison between two algorithms.

V. Conclusion.

Based on the evaluation, the Random Forest algorithm outperforms the Naive Bayes algorithm in several aspects:

- Accuracy: Random Forest achieved an accuracy of 85.7143%, higher than the 81.4672% achieved by Naive Bayes.
- Kappa Statistic: The Kappa statistic for Random Forest (0.7777) is higher than that for Naive Bayes (0.7106), indicating better agreement between predicted

and actual classifications.

- Error Rates: Random Forest has lower Mean Absolute Error (0.1194 vs. 0.1219) and Root Mean Squared Error (0.2444 vs. 0.3347), reflecting more accurate predictions.
- Confusion Matrix: Random Forest shows fewer misclassifications across the classes compared to Naive Bayes.

Overall, the Random Forest algorithm is the better model for this prediction task, offering higher accuracy and better overall performance metrics.

VI. References.

[1] : Chittawar, P. (n.d.). Summer clothing sales prediction [Kaggle notebook]. Retrieved May 25, 2024, from https://www.kaggle.com/code/parthchittawar/summer-clothing-sales-prediction

[2]: Ziegler, A., & König, I. R. (2014). Mining data with random forests: current options for real-world applications. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 4(1), 55-63.

[3]: Singh, S. N., & Sarraf, T. (2020, January). Sentiment analysis of a product based on user reviews using random forests algorithm. In 2020 10th International conference on cloud computing, data science & engineering (Confluence) (pp. 112-116). IEEE.

[4]: Huang, J., Lu, J., & Ling, C. X. (2003, November). Comparing naive Bayes, decision trees, and SVM with AUC and accuracy. In Third IEEE International Conference on Data Mining (pp. 553-556). IEEE.

[5]: Youn, E., & Jeong, M. K. (2009). Class dependent feature scaling method using naive Bayes classifier for text datamining. Pattern Recognition Letters, 30(5), 477-485.

[6]: Jiang, L., Wang, D., Cai, Z., & Yan, X. (2007). Survey of improving naive bayes for classification. In Advanced Data Mining and Applications: Third International Conference, ADMA 2007 Harbin, China, August 6-8, 2007. Proceedings 3 (pp. 134-145). Springer Berlin Heidelberg.