

Table of Contents

[1. Problem Statement 2](#_Toc179047228)

[1.1 Initial Data Set 2](#_Toc179047229)

[2. Data cleansing 3](#_Toc179047230)

[2.1 Data cleansing: focus on data structure 3](#_Toc179047231)

[2.2 Data cleansing: Code interventions 5](#_Toc179047232)

[3. exploratory data analysis 6](#_Toc179047233)

[3.1 Customer Base 6](#_Toc179047234)

[3.2 Product Category Distribution 7](#_Toc179047235)

[3.3 Total Revenue per month 7](#_Toc179047236)

[4. Machine learning 8](#_Toc179047237)

[5. conclusion 9](#_Toc179047238)

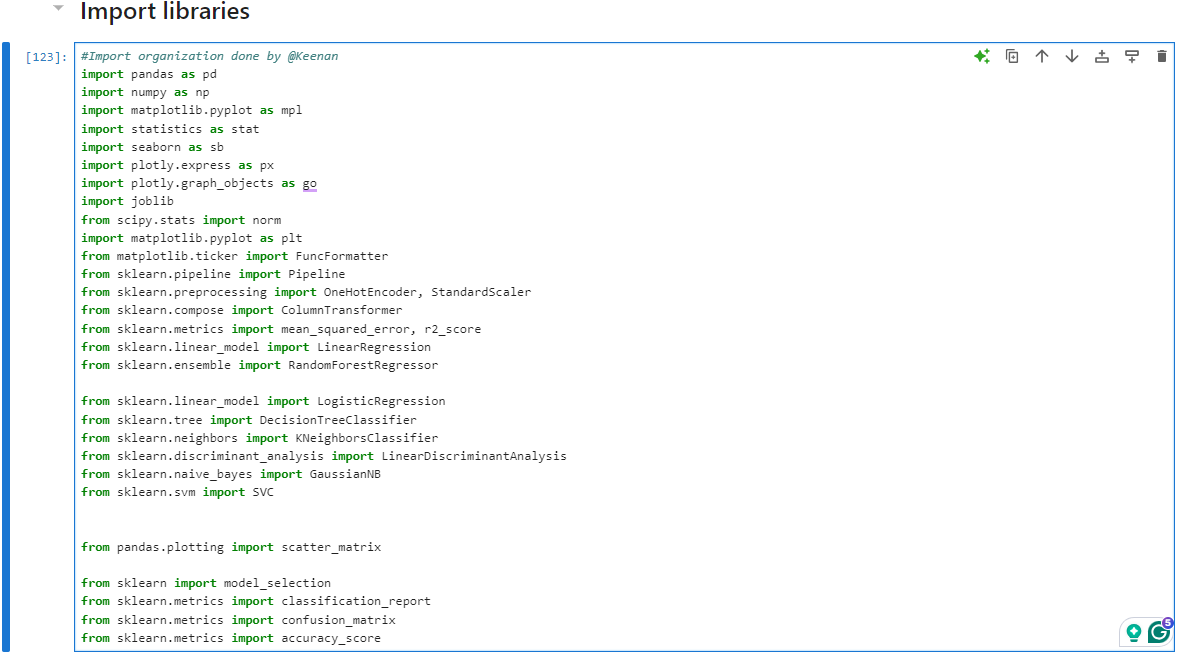
# Problem Statement

The Numpy Supermarket chain, with locations in Windhoek and across Namibia, faces difficulties in leveraging its extensive customer transaction data for informed decision-making. Despite the availability of valuable data, a lack of effective data analysis is preventing the supermarket from optimizing service delivery, product offerings, and customer satisfaction. Utilising Python’s Jupyter Notebook as the main tool, the supermarket applies various data analytics techniques—such as data cleaning, preprocessing, and predictive modeling— to extract actionable insights from customer behaviors. The techniques, implemented in Jupyter Notebook, will drive improvements in operational efficiency, enhance customer loyalty, and contribute to overall business performance.

# Libraries and modules

We imported several Python libraries and modules, setting the stage for a comprehensive data analytics and machine learning workflow. The Screenshot below provides details of the specific libraries and modules used for data manipulation and cleaning, visualization and data exploration, machine learning model building, preprocessing as well as evaluating model performance.

**Screenshot 1. Library Imports Code**



1. **pandas as pd**:

Pandas is a powerful library for data manipulation and analysis. It provides data structures like DataFrames and functions for handling data cleaning, transformation, and analysis. In our Jupyter Notebook, it is stored as **pd** to shorter the name and make it more convenient to work with.

1. **numpy as np**:

NumPy provides support for numerical operations, including working with arrays and performing mathematical computations. It’s often used for efficient numeric data handling, such as creating arrays and performing element-wise operations. In our Jupyter Notebook it is saved as np for convenience.

1. **matplotlib.pyplot as mpl**:

Matplotlib is a library for creating static, animated, and interactive visualizations. The pyplot module provides a MATLAB-like interface for creating plots such as line charts, bar charts, histograms, etc. Which help with data interpretation. In our Jupyter Notebook, it is saved as **mpl** for ease of use.

1. **statistics as stat**:

The Python statistics module provides functions to calculate statistical measures, such as mean, median, and standard deviation, which are useful for summarizing data. In our Notebook it is saved as **stat** for ease of use.

1. **seaborn as sb**:

Seaborn is built on top of Matplotlib and is used for creating more visually appealing and informative statistical plots. It integrates well with Pandas and simplifies the process of generating plots such as heatmaps, pairplots, and violin plots. For our Jupyter Notebook this is stored under the alias **sb**.

1. **plotly.express as px** and **plotly.graph\_objects as go**:

Plotly is a library for creating interactive visualizations. plotly.express provides a high-level interface for generating plots quickly, while plotly.graph\_objects offers more detailed control over the plot elements. These libraries are used to create interactive charts, such as scatter plots, line graphs, and pie charts. They are saved as **px** and **go** in our Notebook, respectively.

1. **joblib**:

Joblib is a library for efficient serialization and parallel processing. It is often used to save and load machine learning models or intermediate results during data processing, which is useful for large datasets and repetitive tasks.

1. **matplotlib.ticker import FuncFormatter**:

FuncFormatter is a Matplotlib utility used for customizing the appearance of ticks on charts. It allows you to format numerical tick values in a more readable or meaningful way.

1. **sklearn.pipeline.Pipeline**:

The Pipeline module from scikit-learn is used to streamline machine learning workflows by combining multiple preprocessing steps and model training into a single, cohesive process. This ensures that the data passes through all necessary transformations before fitting a model.

1. **sklearn.preprocessing.OneHotEncoder** and **StandardScaler**:

OneHotEncoder is used to convert categorical data into a numerical format by creating binary columns for each category (one-hot encoding).

StandardScaler standardizes numeric data by removing the mean and scaling to unit variance, which is essential for certain machine learning algorithms that rely on normalized input data.

1. **sklearn.compose.ColumnTransformer**:

This allows you to apply different preprocessing steps to different columns in the dataset. For example, you might want to apply OneHotEncoder to categorical columns and StandardScaler to numerical columns.

1. **\*\*sklearn.metrics.mean\_squared\_error** and **r2\_score:**

mean\_squared\_error is used to measure the average squared difference between actual and predicted values, often applied in regression tasks.

r2\_score is used to evaluate how well a regression model fits the data, indicating the proportion of variance in the target variable that is explained by the model.

1. **sklearn.linear\_model.LinearRegression**:

Linear Regression is a simple machine learning model that fits a linear relationship between the input features and the target variable. It’s commonly used for predicting continuous variables.

1. **sklearn.ensemble.RandomForestRegressor**:

Random Forest is an ensemble learning method that creates multiple decision trees during training and outputs the average prediction of the individual trees. It’s often used for both regression and classification tasks and handles complex relationships in data well.

1. **scipy.stats.norm**:

This is used to work with the normal distribution, including calculating probabilities, creating normal distribution plots, or fitting data to a normal distribution.

1. **LogisticRegression, DecisionTreeClassifier, KNeighborsClassifier, LinearDiscriminantAnalysis, GaussianNB, SVC**:

These are machine learning classification models from sklearn. They cover different types of algorithms:

* + 1. LogisticRegression: Used for binary or multi-class classification.
    2. DecisionTreeClassifier: A tree-based model for classification.
    3. KNeighborsClassifier: A K-Nearest Neighbors model for classification based on proximity.
    4. LinearDiscriminantAnalysis: A dimensionality reduction and classification technique.
    5. GaussianNB: A Naive Bayes classifier assuming Gaussian distribution for continuous features.
    6. SVC: Support Vector Classification, useful for separating data into categories using hyperplanes.

1. **scatter\_matrix** from pandas.plotting:

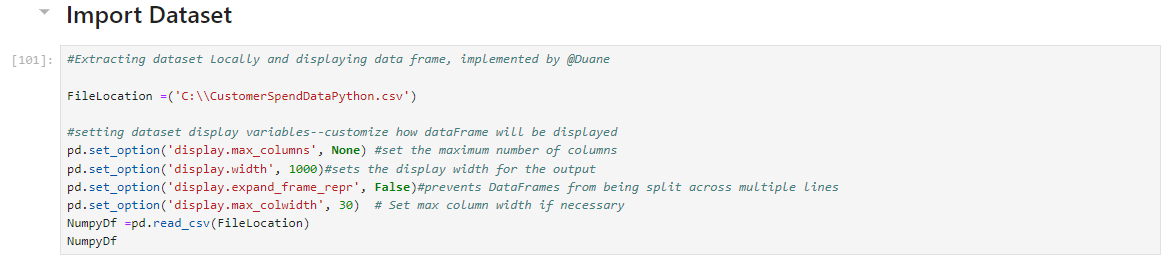
Used to create scatter plot matrices, allowing for visual exploration of pairwise relationships between features in a dataset.

1. **Additional sklearn Imports (model\_selection, classification\_report, confusion\_matrix, accuracy\_score)**:
   1. **model\_selection**: Contains functions for splitting the dataset into training and test sets, as well as cross-validation techniques.
   2. **classification\_report**: Provides a detailed performance summary of a classification model, including precision, recall, and F1 score.
   3. **confusion\_matrix**: Displays the confusion matrix to show the performance of a classification model.
   4. **accuracy\_score**: Calculates the overall accuracy of the model, which is the ratio of correct predictions to total predictions.

# Initial Data Set

The initial data set comprised of 16 columns made up of various data types. The dataset was imported using the code excerpt contained in **Screenshot 2** below. **Table 1** below provides the details of the columns as well as the descriptions of their respective data types. A deeper analysis of the initial dataset found that some of the data types were inappropriate for the purposes of our analysis. Moreover, keeping the initial dataset would have resulted in performance inefficiency and data handing challenges, thus a decision was made to cleanse the data.

**Screenshot 2. Import Dataset Code**



**Table 1. Initial Columns and Data Types**

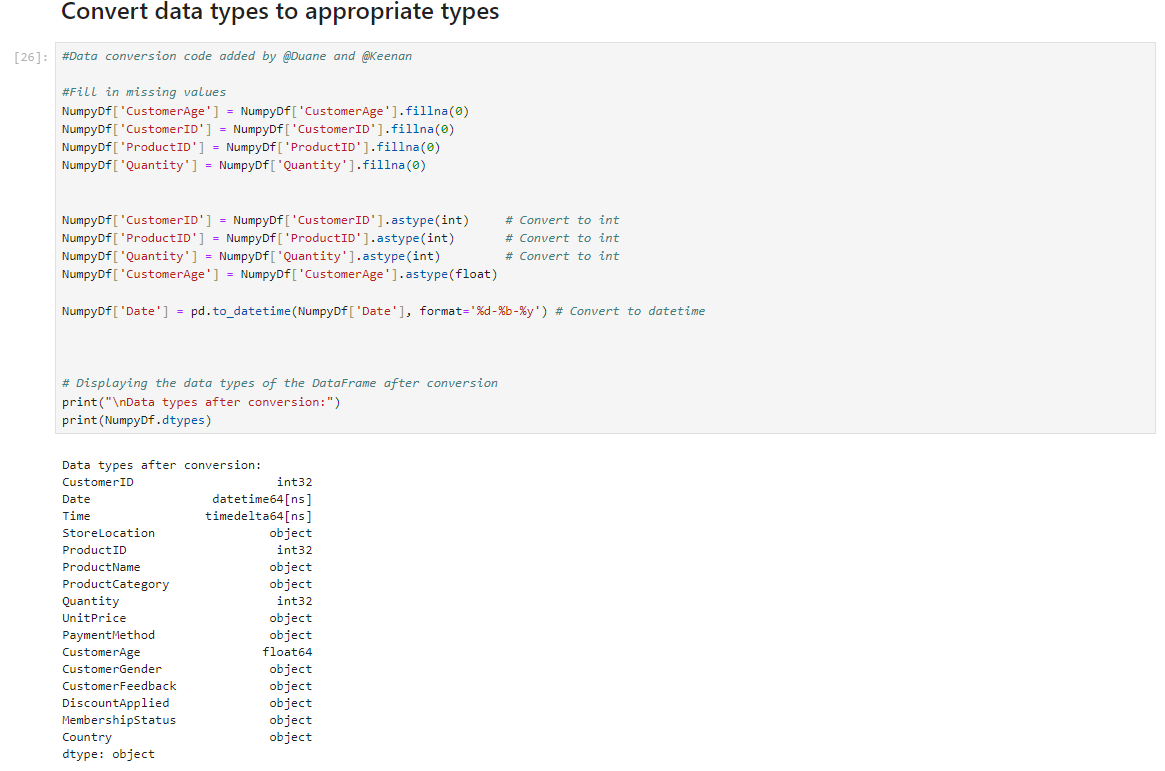


# Data cleansing

To ensure data quality and consistency, a data cleansing exercise was conducted on the initial columns and data types. A top-down approach was applied which focused on data structure first. By examining each column’s data type and its appropriateness for the data it holds we were able to ensure that data consistency was enforced at a higher level across the dataset. Furthermore, by assigning specific data types, we were able to impose constraints on what values are acceptable within each column. For example, converting the data type “object” to datetime64 for the **Date** column. Data type inconsistencies, such as dates stored as strings or numeric data as text were high-level issues and resolving these first simplified later steps because we ensured that all data points in a column followed a single format. Section 2.1 below discusses in detail the steps taken in our top-down approach. **Screenshot 3** below provides an excerpt of the code used to convert data types to their appropriate formats, while **Table 2** provides a summary of our updated columns and data types. Section 2.2 briefly discusses further data cleaning outside of the focus on data structure. These interventions involved coding in Jupyter Notebook to further enhance our data cleansing efforts.

# Data cleansing: focus on data structure

**Screenshot 3. Data Conversion Code**



1. **CustomerID (int32)**

* **Cleansing Applied**: Converted from the initial float64 to int32 to ensure IDs are whole numbers. Any missing or invalid IDs will be addressed, as they are crucial for identifying unique customers. Possible duplicates are also be checked and removed if necessary.

1. **Date (datetime64[ns])**

* **Cleansing Applied**: Converted from the initial object to datetime64 to facilitate date-based operations. Missing or improperly formatted dates are corrected. Any date values that fall outside of expected ranges (e.g., future dates) should be flagged for review.

1. **Time (timedelta64[ns])**

* **Cleansing Applied**: Stored as timedelta64 to allow precise time calculations. Ensure that time data is uniformly formatted and check for any unusual time entries (e.g., negative durations or times that fall outside of business hours).

1. **StoreLocation (object)**

* **Potential Cleansing**: Standardize the text format (e.g., ensure all locations use consistent spelling and case) and check for any erroneous entries. Converting to category can also help identify unexpected values, as they will appear as outliers.

1. **ProductID (int32)**

* **Cleansing Applied**: Changed from the initial float64 to int32. This ensures IDs are whole numbers. Validate for unique IDs and cross-reference with the ProductName column to confirm accurate pairing of IDs and names.

1. **ProductName (object)**

* **Potential Cleansing**: Correct any spelling errors or inconsistencies in product names. We might also normalize the names for uniformity and remove any leading/trailing spaces.

1. **ProductCategory (object)**

* **Potential Cleansing**: Standardize category names to ensure consistency (e.g., "Produce" vs. "produce").

1. **Quantity (int32)**

* **Cleansing Applied**: Converted from the inital float64 to int32, assuming only whole quantities are allowed. Checked for negative or zero values where these are not expected, as they may indicate data entry errors.

1. **UnitPrice (object)**

* **Potential Cleansing**: Since this remained an object, consider converting to a numeric type (float64) for price calculations. Clean up any currency symbols or formatting issues and handle any missing or non-numeric entries.

1. **PaymentMethod (object)**

* **Potential Cleansing**: Standardize entries (e.g., "debit card" vs. "Debit Card").

1. **CustomerAge (float64)**

* **Cleansing Applied**: Ensure there are no negative or unrealistically high age values, as these may indicate errors. Missing values were be handled appropriately, possibly by filling with median or mean age.

1. **CustomerGender (object)**

* **Cleansing**: Standardize the values (e.g., "M" vs. "Male") and ensure only valid entries are present.

1. **CustomerFeedback (object)**

* **Cleansing**: Since this is free-form text, perform basic text cleansing like removing extra spaces and correcting obvious spelling errors.

1. **DiscountApplied (object)**

* **Potential Cleansing**: Standardize entries (e.g., 'Yes' vs. 'yes') and consider converting to a boolean type for easier logical operations. Handle any missing values, possibly by setting to False where appropriate.

1. **MembershipStatus (object)**

* **Cleansing**: Ensure consistent terminology (e.g., "Member" vs. "Yes").

1. **Country (object)**

* **Cleansing**: Column was dropped as all branches currently only exist in Namibia.

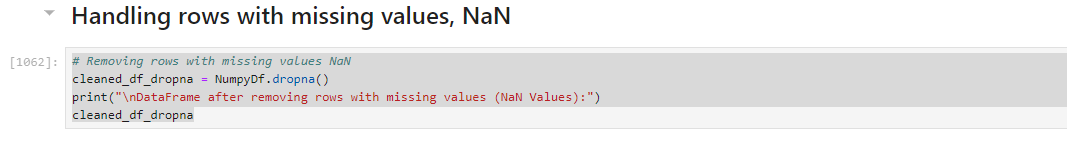
**Table 2. Columns and Data Types post cleansing**



# Data cleansing: Further Code interventions

* + 1. Removal of Incomplete Data

**Screenshot 4. Data Cleaning Missing values Code**



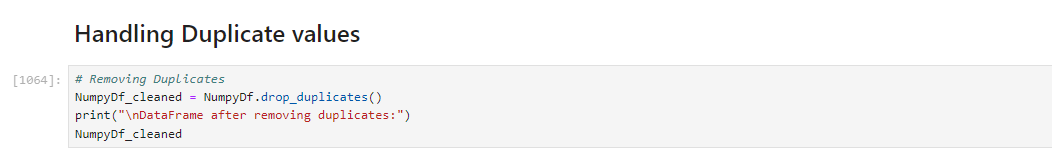
This code snippet was used to remove rows with missing values (NaN) from our DataFrame. The dropna() method is called on the DataFrame NumpyDf. This method is used to **remove rows or columns with missing values (NaN)**.

A new DataFrame called cleaned\_df\_dropna was then created to hold the results of NumpyDF.dropna(). The following print statement outputs a message indicating that the next output will show the DataFrame after NaN values were removed.

Cleaned\_df\_dropna displayes the new DataFRame which no longer contains any rows with missing values.

* + 1. Removing duplicates

**Screenshot 5. Data Cleaning Duplicate Values Code**



This code snippet was used to remove duplicate rows from the **DataFrame**. The **drop\_duplicates()** method is called on the **DataFrame** **NumpyDF** to remove duplicate rows from the **DataFrame**. The results of after the duplicate rows have been removed are stored in **NumpyDf\_cleaned**.

The following print statement then outputs a message indicating that the upcoming output will show the **DataFrame** post duplicate rows having been removed.

**NumpyDf\_cleaned** shows the new **DataFrame** which no longer contains any duplicate rows.

# exploratory data analysis

With the data finally clean we set out to perform exploratory data analysis that would help Numpy Supermarket answer questions about its customer base spread across all regions as well as their gender dynamics. Furthermore, Numpy Supermarket sought to find out the relative popularity across major product categories as well as the balance of distribution across the various regions which could help understand customer preferences and assist with inventory management. Finally, the Supermarket sought to understand its revenue performance relative to the respective month of the year.

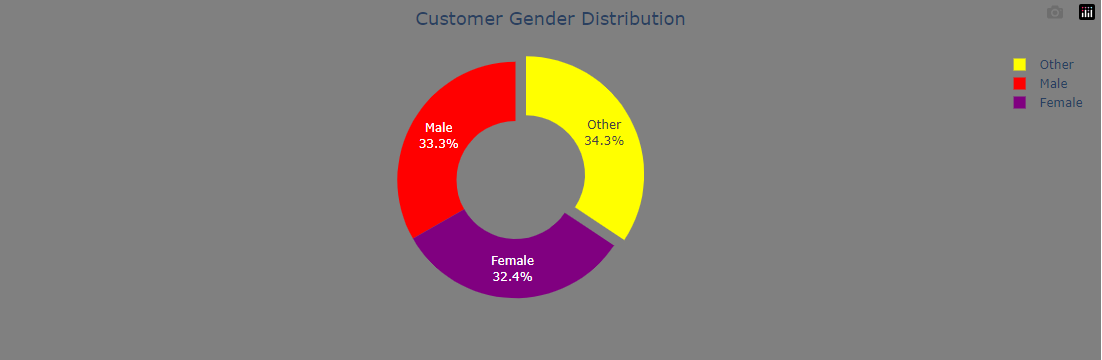
# Customer Base

The Numpy Supermarket customer bases comprise of a total of 2,300 clients that fall into 1 of the following categories: Female, Male or Other. **Screenshot 6** provides the excerpt of the code used to breakdown the customer base according to their respective genders. The data in **Chart 1** below shows that the dominant customer is the one who identifies as “Other” consisting of 790 clients or 34.3% of the total client base. The second largest type of customer identifies as male with a total of 765 clients (33.3% of total customer base), while females make up the smallest group of customers with 745 clients (32.4% of the base) falling into this category.

**Screenshot 6. Customer Breakdown by Gender Code**



**Chart 1 Customer Genders**



The results show that neither conventional gender, “Male” or “Female” are overrepresented in the client base. The fact that the others category is the most dominant type of client indicates that most of Numpy Supermarket’s clients are either relatively young or progressive to identify as “Other”. Furthermore, all three categories fall within the same range of values, all above 740 clients, with no category forming an outright majority of the client base.

# Product Category Distribution

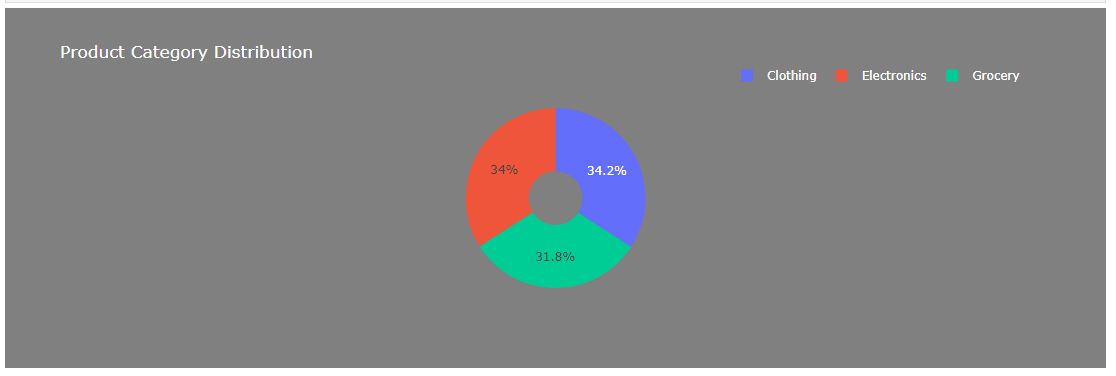
The data further showed that the most popular product category was clothing, which represented 34,2% of the total products sold. **Screenshot 7** below provides an excerpt of the code used to plot Product Distribution Pie Chart.

**Screenshot 7. Product Category Breakdown Code**



When the product category is evaluated in relation to the gender classifications, it is plausible that the reason we see a higher demand for electronics is influenced by the type of client who frequents Numpy Supermarket the most: the young or progressive client. Like the results on gender classification, all major product categories fall within the same range of distribution, all capturing over 30% of the total product distribution. The second most popular product category: clothing, represents 32.4% of all products purchased. Meanwhile the groceries categories represent 31.8% of the products sold by Numpy Supermarket. **Chart 2** below provides a visual representation of the product category analysis.

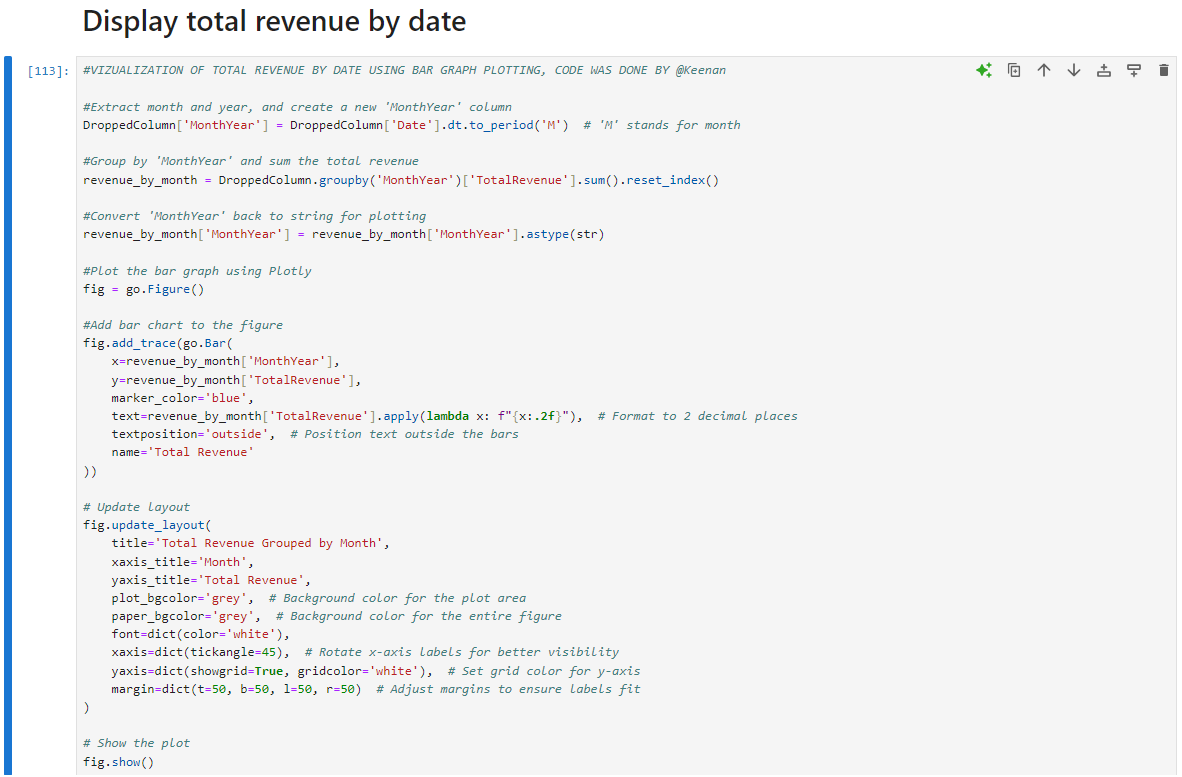
**Chart 2**



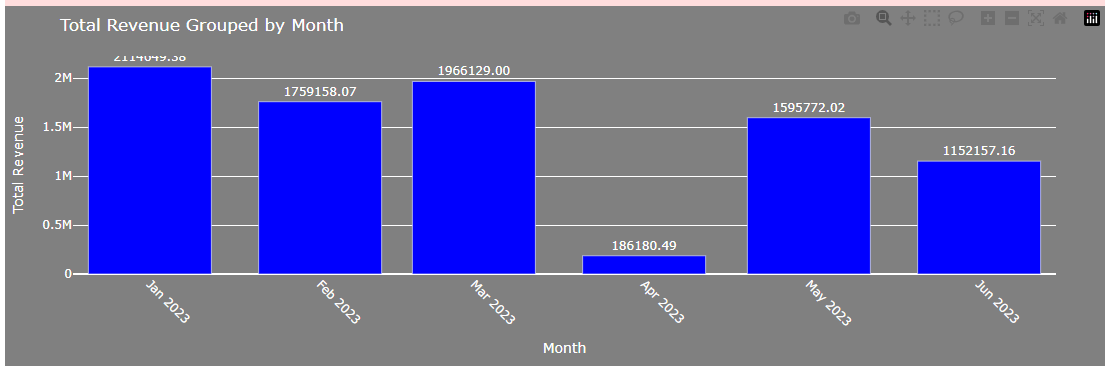
# Total Revenue per month

The data shows that January 2023 was the top month in terms of revenue with N$2.11 million worth of sales recorded for that month (see **Chart 3** below). March 2023 was the second-best month for sales as N$1.97 million worth of sales were recorded during that month. February 2023 closes of the top three with N$1.76 million worth of sales. Additional information obtained from the Revenue Analysis is that the is an alarming drop in sale during the month of April 2023 with only N$186,18 thousand worth of sales. Numpy Supermarket will have to investigate the possible causes of the drastic drop in sales for the month of April to be able to rectify the situation.

**Screenshot 8. Total Revenue by Date Code**



**Chart 3**



# Machine learning

# Random Forest Regression Model

# The Random Forest Regression model achieved a Mean Squared Error (MSE) of 55122.99, and an R² Score of 1.00, indicating a perfect fit. The model was successfully saved for future use.

# Prediction model for Random Forest Regression

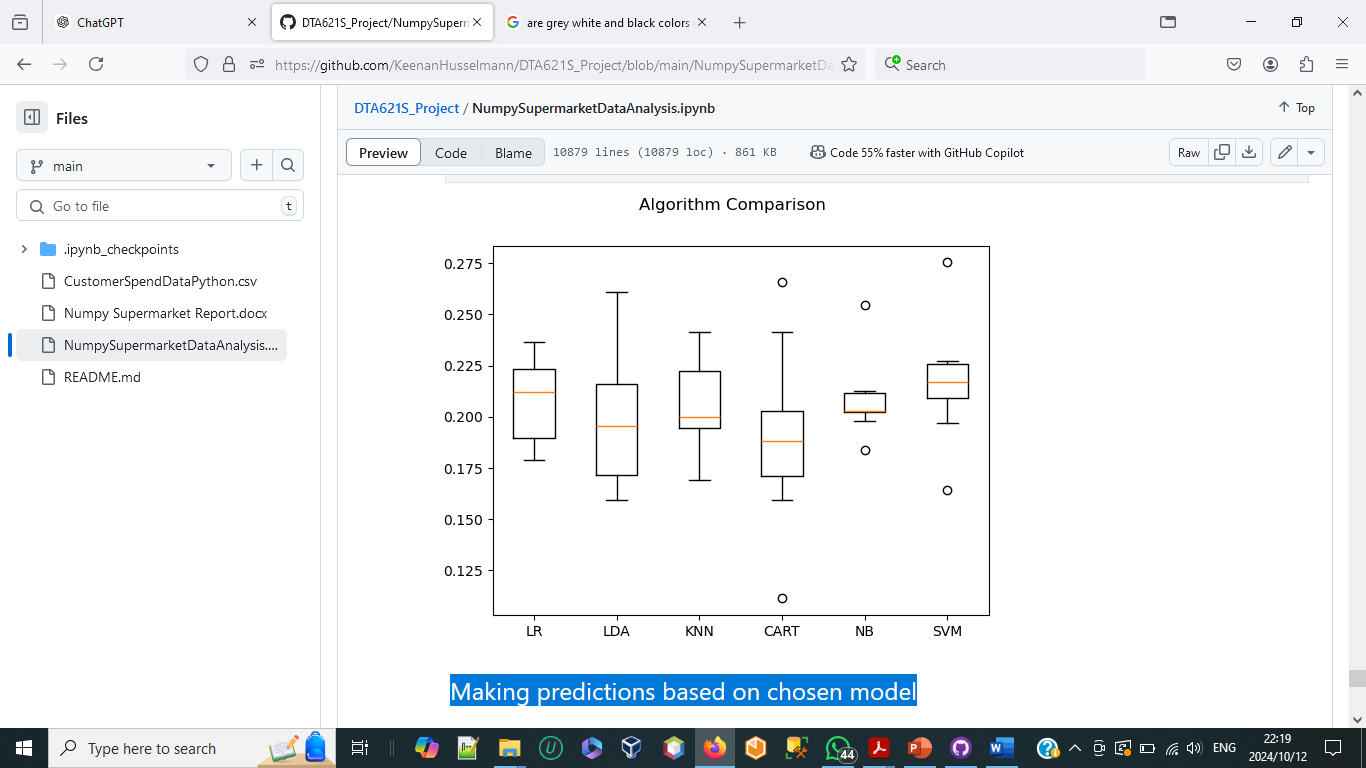
Using the trained model, a new data point was processed, and the Predicted Total Mean Revenue was estimated to be N$3783.05.

# Machine Learning evaluation

Six machine learning models (Logistic Regression, Linear Discriminant Analysis, K-Nearest Neighbors, Decision Trees, Naive Bayes, and SVM) were evaluated. Among these, SVM had the highest average accuracy score of 0.2167, while Decision Trees (CART) showed the lowest performance with an accuracy of 0.1906.

# Comparing algorithms

A boxplot was used to visually compare the performance of these models. The variability in accuracy was reflected, with SVM displaying a tighter distribution compared to others.



# Making predictions based on chosen model

The K-Nearest Neighbors (KNN) model yielded an accuracy of 18%. The confusion matrix and classification report revealed low precision, recall, and F1 scores across all classes, indicating poor classification performance with imbalanced support across the dataset.

# conclusion