



Smart Phones Prices Project

Assigned T.A

Hossam Sherif

Esraa Mahmoud	20231700207
Samaa Hossam	20231700226
Mehrael Sawiris	20231700249
Yousef Samir	20231700263
Abdalkeren Ehap	20231700265

Contents

1. Data Cleaning	1
2. Visualization	4
3. Feature Engineering & Selection	7
4. Modeling	10
• Grid Search	10
• Decision Tree	11
• SVM	13
• Random Forest	15
• XGB boost	18
• KNN	21
• Testing results	23

Data Cleaning

Data cleaning and preprocessing are essential steps to ensure data quality, consistency, and suitability for machine learning models. The following procedures were applied to both the training and testing datasets.

1- Column Name Standardization

Column names were standardized by removing leading and trailing spaces and replacing internal spaces with underscores.

This improves readability and prevents potential errors during feature access and manipulation.

```
df.columns = df.columns.str.strip().str.replace(" ", "_")
```

2- Duplicate Record Removal

Duplicate records can bias the learning process and negatively impact model performance

```
df.drop_duplicates(inplace=True)
```

3- Processor Series Transformation

Processor generation information was converted from textual format (e.g., *Gen1*, *Gen2*) into numerical values. Unknown entries were replaced with the mode, and the column was cast to a numeric type.

This transformation was made because machine learning algorithms require numerical inputs

```
df["Processor_Series"] = df["Processor_Series"].str.replace(" Gen1", ".1").str.replace(" Gen2", ".2").str.replace("Unknown", "35").astype(float)
```

4- Memory Card Size Normalization

The `memory_card_size` feature was cleaned by removing unit labels (GB, TB), converting values expressed in terabytes to gigabytes, and creating a new numerical feature (`memory_card_size_GB`). The original column was then removed. Standardizing units and data types ensures consistency and allows the feature to be effectively used by the model.

```
df["memory_card_size_GB"] = df["memory_card_size"].astype(str).  
str.replace("GB", "").str.replace("TB", "*1000").map(lambda x:eval(x)) # eval execute Python code  
df["memory_card_size_GB"] = df["memory_card_size_GB"].astype(int)  
df.drop(columns="memory_card_size", inplace=True)
```

5 - Operating System Version Processing

The operating system version was cleaned by removing unnecessary characters and correcting formatting inconsistencies. Values were converted to a numerical format, and scaling was applied where required.

This enables the OS version to be treated as a meaningful numerical feature rather than raw text.

```
# Convert 'os_version' column to numerical:
# Remove 'v' and dots
df["os_version"] = df["os_version"].str.replace("v", "")
df["os_version"] = df["os_version"].str.replace(".", "", 1).astype(float)
# Apply condition: if > 17, divide by 10 to correct the formatting
df["os_version"] = df["os_version"].apply(lambda x: x / 10 if x > 17 else x)
```

6- Categorical Feature Standardization

All categorical features were converted to lowercase and stripped of extra whitespace.

This prevents the same category from being interpreted as multiple distinct values due to case or formatting differences.

```
categorical_columns = df.select_dtypes(include=['object']).columns.tolist()
for c in categorical_columns:
    df[c] = df[c].str.lower().str.strip()
```

7- Rare Category Handling

Categories with a frequency below a defined threshold were grouped into a single category labeled “**Other.**”

Reducing the number of rare categories minimizes noise and improves model generalization.

```
for c in df:
    if df[c].dtype in ["object"]:
        count = df[c].value_counts()
        to_replace = count[count < min_freq].index
        df[c] = df[c].replace(to_replace, 'other')
```

8- Missing Value Treatment

Numerical features were imputed using the mean value.

Categorical features were imputed using the most frequent value (mode).

Most machine learning algorithms cannot handle missing values; this approach preserves the overall data distribution while maintaining simplicity.

```
for c in df_train:
    # Fill numeric columns with the mean
    if df_train[c].dtype in ["int64", "float64"]:
        df_train[c].fillna(df_train[c].mean())

    # Fill categorical columns with the mode
    if df_train[c].dtype in ["object"]:
        df_train[c].fillna(df_train[c].mode()[0])
        df_train[c].str.replace("unknown", df_train[c].mode()[0])

for c in df_test:
    if df_test[c].dtype in ["int64", "float64"]:
        df_test[c].fillna(df_train[c].mean())

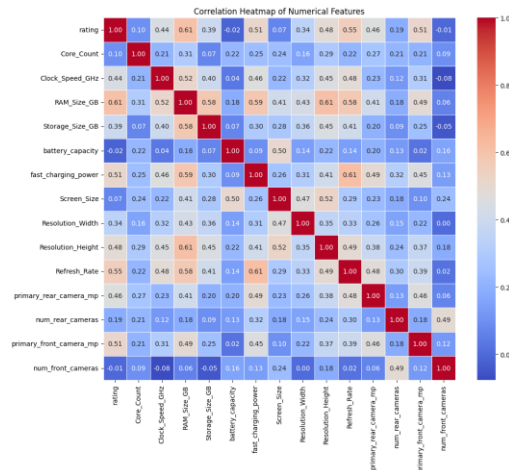
    if df_test[c].dtype in ["object"]:
        df_test[c].fillna(df_train[c].mode()[0])
        df_test[c].str.replace("unknown", df_train[c].mode()[0])
```

Note:

Identical preprocessing steps were applied to both the training and test datasets. Ensures consistency and prevents data leakage, leading to a fair and reliable model evaluation.

Visualization

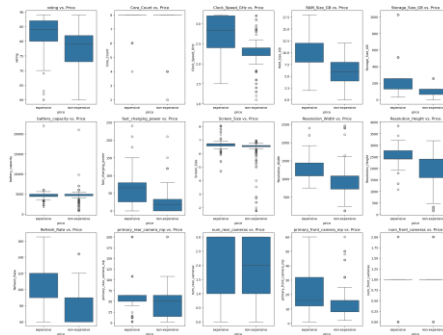
1. Correlation Analysis of Numerical Features



Key Insights:

- Strong positive correlations are observed, particularly between RAM Size GB and Storage Size GB (indicating that devices with larger RAM often come with more storage).
- RAM Size GB also shows a positive correlation with Rating, suggesting that higher-rated phones tend to have more RAM.
- Other potential correlations include moderate links between Clock Speed GHz and Refresh Rate, or negative ones like Refresh Rate and Battery Capacity (due to power consumption trade-offs).

2. Box Plots (Numerical Features vs. Price Category)



Key Insights:

- **Core Count:** Most phones have 8 cores; lower-priced phones may have fewer, but it's not a strong price differentiator.
- **Clock Speed:** Expensive phones have higher processor speeds (~2.8 GHz vs. ~2.2 GHz).
- **RAM Size:** Clear distinction—expensive: 8–12 GB, non-expensive: 4–6 GB; RAM strongly influences price.
- **Storage Size:** Expensive phones offer 128–256 GB (up to 1 TB), non-expensive: 64–128 GB.
- **Battery Capacity:** Similar across prices (~4500–5000 mAh); not a strong differentiator.
- **Fast Charging:** Expensive phones support 65–120 W, non-expensive: 10–30 W.
- **Screen Size:** Similar (~6.5 inches); minimal effect on price.
- **Screen Resolution:** Expensive phones have higher resolution; resolution strongly correlates with price.

3. Count Plots(Categorical Features vs. Price Category)

Key Insights:

- **Dual SIM:** Common in both categories; more prevalent among non-expensive phones.
- **4G & 5G:** 5G mainly in expensive phones; 4G widespread in cheaper phones.
- **NFC:** Mostly found in expensive phones; rare in economic phones.
- **Processor Brand:** Expensive phones mainly use Snapdragon; cheap phones often use Helio or Unisoc.
- **RAM Tier:** flagship RAM in expensive phones; mid-range/high-end in cheaper phones.
- **Notch Type:** Punch-hole common in expensive phones; Water drop common in cheaper phones.
- **OS Name:** Android dominates across both categories.

4. Scatter Plot(Clock Speed vs. Performance Tier by Price)

Key Insights:

Expensive phones generally occupy the higher clock speed range, particularly above 2.6 GHz, while non-expensive phones mostly remain below 2.4 GHz. The "flagship" tier consists almost entirely of devices with high clock speeds (up to 3 GHz), while the "budget" tier is limited to non-expensive devices with speeds generally below 2.4 GHz. A few non-expensive phones reach flagship-level speeds, but these are exceptions rather than the norm.

5. Pie Charts(RAM Tier in Expensive vs. Non-Expensive Phones)

Key Insights:

- Expensive phones are heavily skewed toward High-End and Flagship RAM tiers.
- Non-expensive phones primarily feature Mid-Range RAM.

6 . Histogram (Refresh Rate Distribution by Price)

Key Insights:

- Non-expensive phones peak at 60Hz (standard for cost savings).
- Expensive phones dominate 120Hz+ bins, with some at 145Hz/160Hz.

7. Box Plot(Battery Capacity by OS and Price)

Key Insights:

- Android phones show similar battery sizes for expensive and non-expensive devices, making battery capacity a weak price indicator.
- Most operating systems in the dataset show **class imbalance** across price categories. Android is the only OS with meaningful representation of both expensive and non-expensive phones, while other OS types are dominated by a single price class.

Feature Engineering:

This pipeline prepares a smartphone dataset for machine learning by applying feature engineering, encoding, transformation, and feature selection techniques to produce clean, numeric, and informative training and testing datasets.

1. Derived Features

- **performance_score**: Combines CPU cores, clock speed, and RAM to represent device performance.
- **camera_quality_score**: Weighted score combining rear (70%) and front (30%) camera megapixels.

```
df['performance_score'] = df['Core_Count'] * df['Clock_Speed_GHz'] * (df['RAM_Size_GB'] / 4)
df['camera_quality_score'] = (df['primary_rear_camera_mp'] * 0.7 + df['primary_front_camera_mp'] * 0.3)
```

2. Skewness & Outlier Treatment

Applies log1p transformation to selected numerical columns to reduce skewness, minimize outlier impact and Improve model stability.

```
# --- Outlier/Skewness Treatment ---
# List columns that need transformation
outliers_col = ['rating', 'Processor_Series', 'Core_Count', 'Clock_Speed_GHz',
                'RAM_Size_GB', 'Storage_Size_GB', 'fast_charging_power',
                'Screen_Size', 'Resolution_Width', 'Resolution_Height', 'Refresh_Rate',
                'primary_rear_camera_mp', 'primary_front_camera_mp', 'num_front_cameras', 'memory_card_size_GB']

# Apply log transformation (np.log1p) to reduce skewness in numerical features
outliers_col_clean = [col.strip() for col in outliers_col]
for c in outliers_col_clean:
    df[c] = np.log1p(df[c])
```

3. Encoding Techniques

- **Binary Encoding**
Maps binary categorical features (price, 5G, NFC) into numerical values using predefined mappings.
- **Ordinal Encoding**
Encodes ordered categorical features (Performance_Tier, RAM_Tier) while preserving their natural hierarchy.
The fitted encoder is saved for reuse.

```

bmap = {'yes':1, 'no':0, 'expensive': 1, 'non-expensive': 0}
binary_columns = ['price', '5G', 'NFC']
for col in binary_columns:
    df[col] = df[col].apply(lambda x: bmap[x])

# Define the order for ordinal features (tiers)
tiers_order = [
    ['unknown', 'budget', 'low-end', 'mid-range', 'high-end', 'flagship'],
    ['unknown', 'budget', 'low-end', 'mid-range', 'high-end', 'flagship']
]

# Encode ordinal columns (Performance_Tier, RAM_Tier) based on their defined order
oe = OrdinalEncoder(categories=tiers_order)
df[['Performance_Tier', 'RAM_Tier']] = oe.fit_transform(df[['Performance_Tier', 'RAM_Tier']])

```

- **Target Encoding**

Target Encoding is applied to the (brand) feature by replacing each brand with a numerical value representing the smoothed mean of the target variable (**price**) for that brand.

The encoder calculates the average price per brand while applying smoothing and a minimum sample threshold to reduce noise and overfitting for rare brands.

It is fitted only on the training data to avoid data leakage, then reused to transform the test data using the learned mappings.

```

# Target Encoding
# Initialize the Target Encoder for the 'brand' column
te = TargetEncoder(cols=['brand'], min_samples_leaf=20, smoothing=10)
df_train['brand'] = te.fit_transform(df_train['brand'], df_train['price'])
df_test['brand'] = te.transform(df_test['brand'])

```

- **One-Hot Encoding**

One-Hot Encoding transforms nominal categorical variables into binary indicator features.

Each category is represented as a separate column, and the first category is dropped to prevent multicollinearity.

The encoder is fitted on the training set and then applied to the test set to ensure consistent feature alignment.

```

one_hot_cols = ['Processor_Brand', 'Notch_Type', 'os_name']
ohn = OneHotEncoder(drop='first', sparse_output=False)

# Fit and transform the training data
encoded_train = ohn.fit_transform(df_train[one_hot_cols])
encoded_train_cols = ohn.get_feature_names_out(one_hot_cols)
encoded_train_df = pd.DataFrame(encoded_train, columns=encoded_train_cols)

# Transform the test data using the fitted encoder
encoded_test = ohn.transform(df_test[one_hot_cols])
encoded_test_cols = ohn.get_feature_names_out(one_hot_cols)
encoded_test_df = pd.DataFrame(encoded_test, columns=encoded_test_cols)

#train df
df_train.reset_index(drop=True, inplace=True)
df_train = pd.concat([df_train, encoded_train_df], axis=1)
df_train.drop(columns=one_hot_cols, inplace=True)

#test df
df_test.reset_index(drop=True, inplace=True)
df_test = pd.concat([df_test, encoded_test_df], axis=1)
df_test.drop(columns=one_hot_cols, inplace=True)

```

- **Feature Selection & Mutual Information**

Mutual Information is used to rank features based on their relevance to the target variable, enabling the selection of the most informative features.

Based on this analysis, the selected features include: (**Processor_Series**, **performance_score**, **Resolution_Height**, **primary_front_camera_mp**, **Clock_Speed_GHz**, **NFC**, **camera_quality_score**, **Resolution_Width**, **RAM_Size_GB**, **Storage_Size_GB**, **RAM_Tier**, **rating**, **fast_charging_power**, **Screen_Size**, **5G**, **Refresh_Rate**, **brand**, **primary_rear_camera_mp**) which together capture the most influential hardware, performance, and connectivity characteristics affecting price prediction.

```

mi = mutual_info_regression(X_train, y_train)
mi_series = pd.Series(mi, index=X_train.columns)
mi_series = mi_series.sort_values(ascending=False)

# Print the top 20 features with the highest MI scores
print(mi_series.head(20))

```

Modeling

1- Grid Search

Model Selection & Hyperparameter Tuning

- Implemented 5 ML models: Decision Tree, Random Forest, SVM, XGBoost, KNN.
- Defined hyperparameter grids for each model.
- Used GridSearchCV with 5-fold cross-validation to find the best parameters for each model.
- Evaluated using accuracy, confusion matrix, and classification report.
- Importance: Ensures each model is optimized and fairly compared.
- Next Step: Use the best parameters from GridSearch to train each model individually for final predictions.

```
> def run_grid_search(models, parameters, X_train, y_train, X_test, y_test):
    results = {}
    for model_name, model in models.items():
        print(f"Running GridSearchCV for {model_name}...")

        grid_search = GridSearchCV(model, parameters[model_name], cv=5, scoring='accuracy', n_jobs=-1, verbose=1)

        grid_search.fit(X_train, y_train)
        results[model_name] = grid_search

        print(f"Best parameters for {model_name}: {grid_search.best_params_}")

        y_pred = grid_search.predict(X_test)

        print(f"--- Results for {model_name} ---")
        print("Confusion Matrix:")
        print(confusion_matrix(y_test, y_pred))
        print("\nClassification Report:")
        print(classification_report(y_test, y_pred))
        print("=" * 40)

    run_grid_search(models, parameters, X_train, y_train, X_test, y_test)
```

24] ✓ 1m 37.0s Python

2- Decision Tree

Hyperparameters & Their Importance:

1. `class_weight='balanced'`

- Ensures that the model accounts for imbalanced classes in the dataset by giving higher weight to minority classes.
- Important to prevent the model from being biased toward majority classes.

2. `criterion='gini'`

- The function used to measure split quality at each node.
- `'gini'` focuses on minimizing impurity; `'entropy'` could also be used, but `'gini'` is computationally faster.

3. `max_depth=20`

- Limits the maximum depth of the tree.
- Prevents overfitting by stopping the tree from growing too deep while still capturing sufficient patterns.

4. `max_features=None`

- Uses all features when looking for the best split.
- Helps the tree consider all available information to make optimal decisions.

5. `min_samples_leaf=2`

- Minimum number of samples required in a leaf node.
- Avoids creating leaves with very few samples, reducing overfitting and making predictions more generalizable.

6. `min_samples_split=2`

- Minimum number of samples required to split an internal node.
- A small value allows more splits, making the tree flexible, while a larger value would make it more conservative.

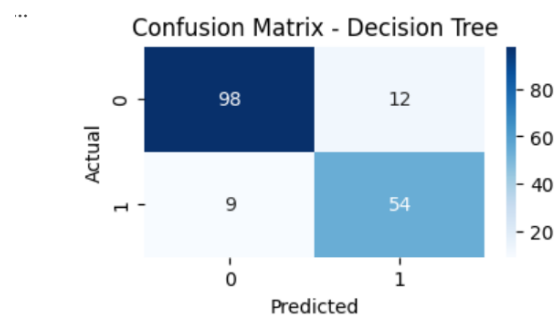
```
DT=DecisionTreeClassifier(class_weight='balanced', criterion='gini', max_depth=20, max_features=None, min_samples_leaf=2, min_samples_split=2)
DT.fit(X_train,y_train)
y_pred=DT.predict(X_test)
print("classification report")
print(classification_report(y_test,y_pred))
#plot confusion matrix
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(8,6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix - Decision Tree')
plt.show()
joblib.dump(DT, 'DT.pkl')
```

[25] ✓ 0.1s Python

Decision Tree Model Evaluation

- Accuracy: 88% overall.
- Class 0: Precision 92%, Recall 89%, F1-score 0.90
- Class 1: Precision 82%, Recall 86%, F1-score 0.84
- Confusion Matrix Highlights:
 - Correctly predicted: 98 (class 0), 54 (class 1)
 - Misclassified: 12 ($0 \rightarrow 1$), 9 ($1 \rightarrow 0$)
- Insight: Model performs slightly better on class 0; overall well-balanced and effective.

classification report					
	precision	recall	f1-score	support	
0	0.92	0.89	0.90	110	
1	0.82	0.86	0.84	63	
accuracy			0.88	173	
macro avg	0.87	0.87	0.87	173	
weighted avg	0.88	0.88	0.88	173	



3- SVM model

Hyperparameters & Their Importance:

1. `C = 10` (Regularization Strength)

- Controls the balance between margin maximization and classification accuracy.
- Lower values (e.g., 0.1 or 1) were tested but caused underfitting, leading to lower accuracy.
- Higher values increase sensitivity to noise.
- Chosen value (10) provided the best balance between bias and variance, improving accuracy without overfitting.

2. `kernel = 'rbf'` (Radial Basis Function)

- Enables the model to learn non-linear decision boundaries, which are common in real-world data.
- Linear kernel failed to capture complex feature interactions.
- Polynomial kernel increased complexity without performance gains.
- RBF kernel achieved the highest cross-validation accuracy.

3. `gamma = 'scale'`

- Determines how far the influence of each training sample reaches.
- `'auto'` and fixed gamma values (0.1, 0.01) were tested but resulted in either overfitting or underfitting.
- `'scale'` adapts gamma automatically based on feature variance, producing a smoother and more stable decision boundary.

4. `class_weight = None`

- Indicates that class imbalance was either minimal or already handled during preprocessing.
- Using `'balanced'` was tested but slightly reduced precision for the majority class.
- No class weighting produced better overall performance.

```

SVM=SVC(C=10, class_weight=None, degree=2, gamma='scale', kernel='rbf') #with 30 features
SVM.fit(X_train,y_train)
y_pred=SVM.predict(X_test)
print("classification report")
print(classification_report(y_test,y_pred))
#plot confusion matrix
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(8,6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')

plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix - SVM')
plt.show()

joblib.dump(SVM, 'SVM.pk1')

```

[26] ✓ 0.1s Python

SVM Model Evaluation

- Accuracy: 92%
- Class 0: Precision 0.92, Recall 0.95, F1-score 0.94
- Class 1: Precision 0.92, Recall 0.86, F1-score 0.89
- Confusion Matrix:
 - Correct predictions: 105 (class 0), 54 (class 1)
 - Misclassifications are low, mainly in class 1
- Insight: Model shows strong overall performance with good class separation and reliable generalization.

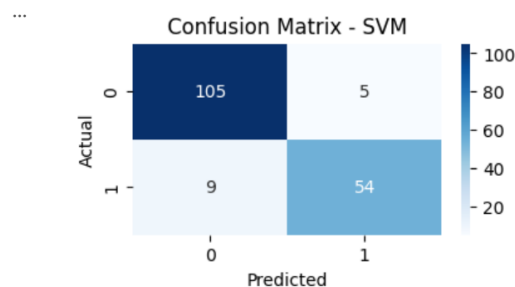
```

...  classification report
           precision    recall  f1-score   support

         0       0.92      0.95      0.94       110
         1       0.92      0.86      0.89        63

    accuracy          0.92          173
   macro avg       0.92      0.91      0.91       173
  weighted avg       0.92      0.92      0.92       173

```



4- Random Forest Model

Intro:

Random forest model is a subset model came from developing the decision tree methodology

In this algorithm we take multiple decision tree and asking different question but based on what ?

We can set our goal to divide the tree until we reach pure group
To do this we use

1. Gini impurity (cost function)

$$Gini(S) = 1 - \sum_{i=1}^c (p_i)^2$$

Our goal is to set this number to as close as 0 (pure) and far a way from 0.5 (completely random)

2.gain of split

$$Gini_{Split} = \left(\frac{N_{Left}}{N_{Total}} \right) \times Gini_{Left} + \left(\frac{N_{Right}}{N_{Total}} \right) \times Gini_{Right}$$

Because if we only depend on the Gini it will be super confusing to the model we must evaluate the question by the gain of split

Hyperparameters & Their Importance:

1. **N_estimators** the number of trees that will be created high number high accuracy high sources usage.
2. **Max_depth** maximum number of levels tree allowed to go for we use it to prevent overfitting.

3. **Min_samples_split** the minimum number of samples that the tree can split to reduce the noise
4. **min_samples_leaf** same as the above but that's for leaf node and it is worked before the splitting not after
5. **max_feature** this will define how the tree would be blind to all features
6. **Bootstrap** if the data used some feature as a question it is allowed to use it again.
7. **Class_weight** how the model will be punished for small sample

```
rf = RandomForestClassifier(bootstrap=True, class_weight='balanced',
                           max_depth=20, max_features='sqrt',
                           min_samples_leaf=1,
                           min_samples_split=2, n_estimators=100)

rf.fit(X_train, y_train)
y_pred = rf.predict(X_test)
print("classification report")
print(classification_report(y_test, y_pred))

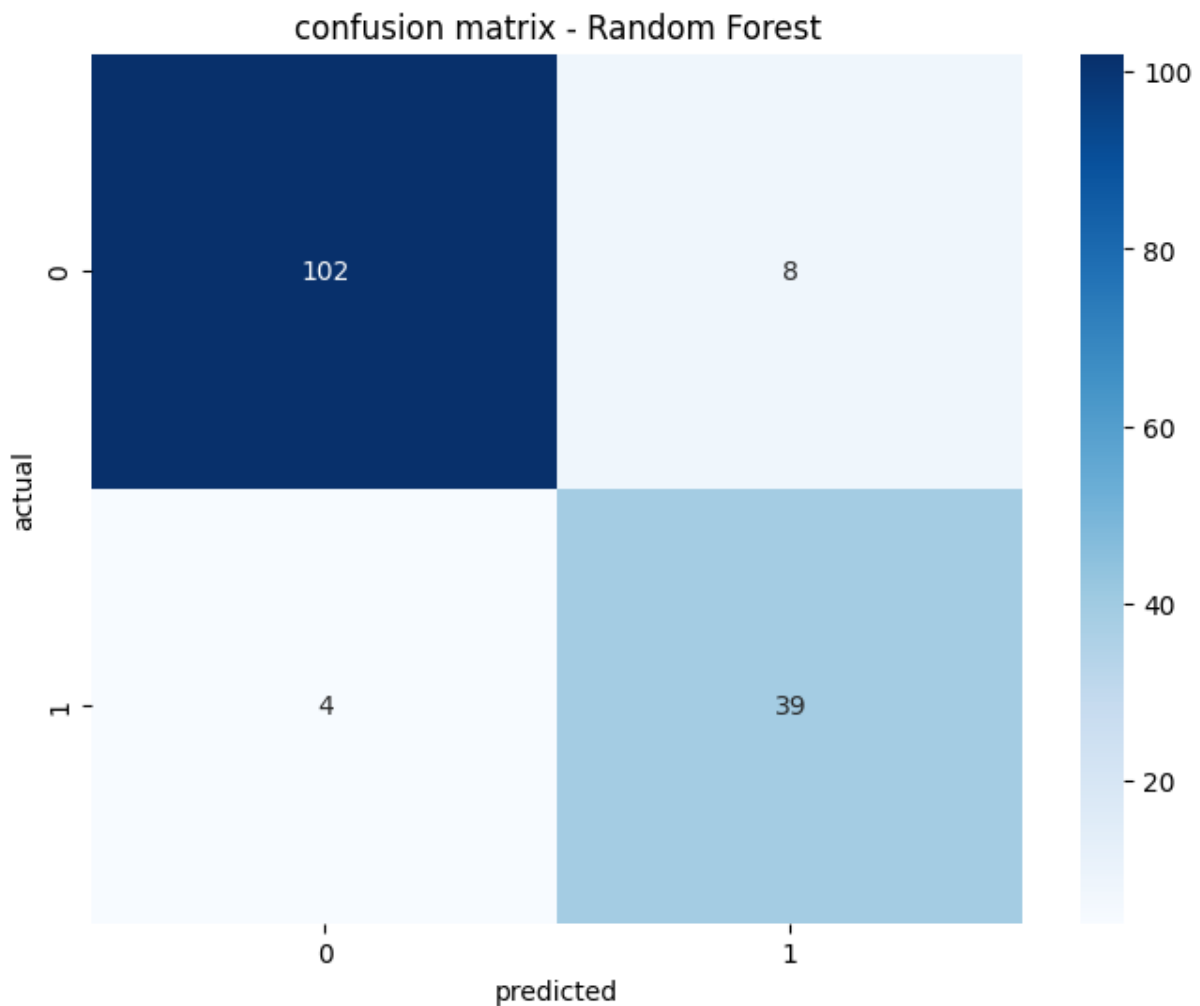
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(8,6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('predicted')
plt.ylabel('actual')
plt.title('confusion matrix - Random Forest')
plt.show()

joblib.dump(rf, 'RF.pkl')
```

Random forest Model Evaluation

- Accuracy: 92%
- Class 0: Precision 0.96, Recall 0.93, F1-score 0.94
- Class 1: Precision 0.83, Recall 0.9, F1-score 0.87

- Confusion Matrix:
 - Correct predictions: 102 (class 0), 39 (class 1)
- Insight: The model shows strong overall accuracy (92%). While Class 0 performance is excellent, **Class 1 shows a drop in precision (0.83)** despite good recall (0.91). This suggests the model captures most Class 1 cases well but occasionally mistakes Class 0 for Class 1.



5- XG boost Model

Intro: the xgboost algorithm is a subset of a random forest algorithm but the twist here is only the first node can predict the price (target) the remaining nodes will try to fix the error that happened in the first node so in random forest the trees is isolated from each other but here the trees is learning from each other

1. The Residual

How far the kth tree from the true answer ?

$x = \text{predict} - \text{true}$

So the next tree will work only on x

$$\text{Prediction}(z) = \log\left(\frac{p}{1-p}\right)$$

After getting the p we will scale it using sigmoid

$$\text{Probability}(p) = \frac{1}{1 + e^{-z}}$$

After scaling it we will use the logloss

$$\text{LogLoss} = -[y \ln(p) + (1 - y) \ln(1 - p)]$$

And its second derivative

$$\text{Hessian} = p \times (1 - p)$$

Hyperparameters & Their Importance:

1. **N_estimators** the number of trees that will be created high number high accuracy high sources usage.
2. **Max_depth** maximum number of levels tree allowed to go for we use it to prevent overfitting.
3. **learning_rate** the value of the step taken in each Gradient decent
4. **subsample** this will defined how the tree would be blind to rows
5. **Colsample_bytree** this will defined how the tree would be blind to all feature
6. **gamma** minimum value of gain to allow the tree to split

```
XGB = XGBClassifier(colsample_bytree=0.8, gamma=0,
                    learning_rate=0.1, max_depth=3,
                    n_estimators=200, scale_pos_weight=1,
                    subsample=0.7, use_label_encoder=False, eval_metric='logloss')

XGB.fit(X_train, y_train)
y_pred = XGB.predict(X_test)

print("classification report")
print(classification_report(y_test, y_pred))

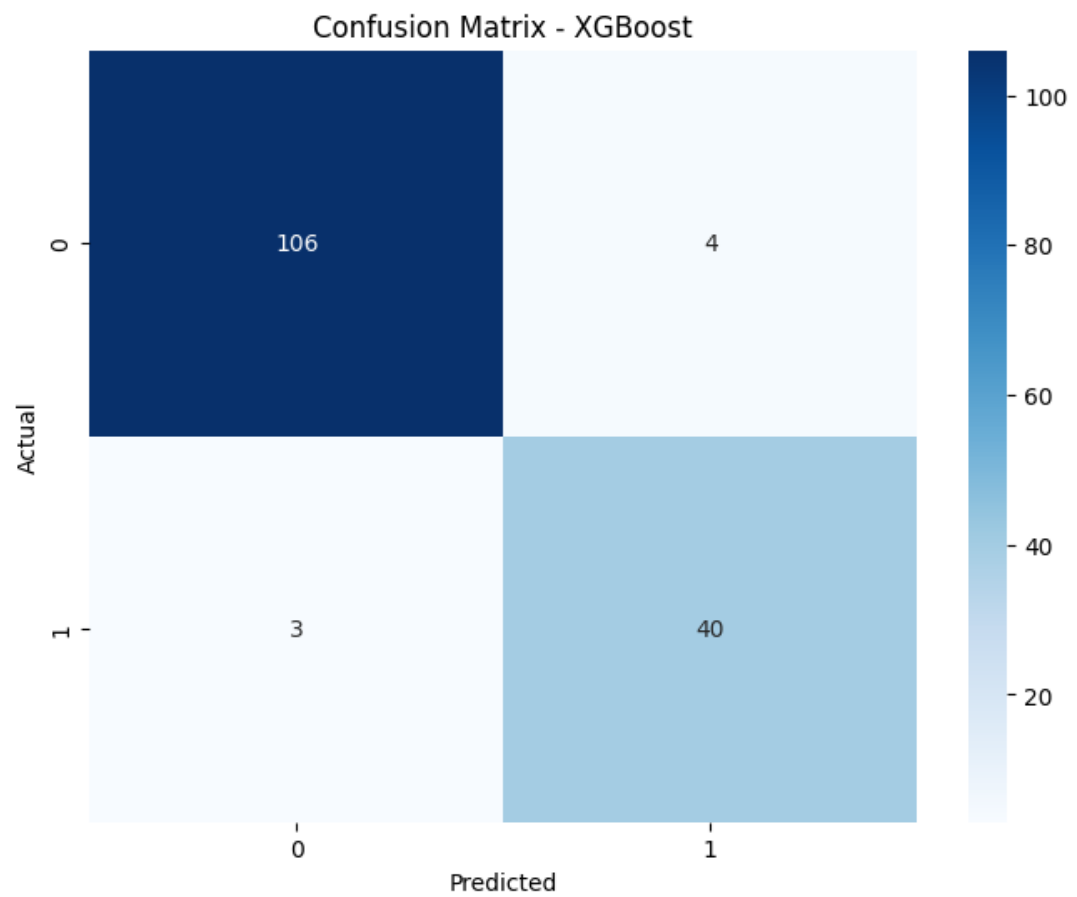
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(8,6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix - XGBoost')
plt.show()

joblib.dump(XGB, 'XGB.pkl')
```

XGBOOST Model Evaluation

Accuracy: 95%

- Class 0: Precision 0.97, Recall 0.96, F1-score 0.97
- Class 1: Precision 0.91, Recall 0.93, F1-score 0.92
- Confusion Matrix:
 - Correct predictions: 106 (class 0), 40 (class 1)
 - Misclassifications are low, mainly in class 1



6- KNN Model

Hyperparameters & Their Importance:

1. `n_neighbors = 7` (Number of Neighbors)

- Controls how many nearest data points are considered when making a prediction.
- Lower values (e.g., 3 or 5) were tested but made the model sensitive to noise and outliers.
- Higher values reduced model sensitivity to local patterns and slightly decreased accuracy.
- The chosen value (7) provided a balanced trade-off between bias and variance, leading to more stable predictions.

2. `metric = 'manhattan'` (Distance Metric)

- Defines how the distance between data points is calculated.
- Euclidean distance was tested but performed slightly worse on this dataset. outliers.
- Manhattan distance better captured feature differences after scaling and improved classification accuracy.

3. `p= 1` (Minkowski Power Parameters)

- Determines the type of distance used in the Minkowski metric.
- A value of `p = 1` corresponds to Manhattan distance.
- This choice reinforced the effectiveness of the selected distance metric for this problem..

4. `Weights = 'distance'` (Neighbor Weighting)

- Assigns higher influence to closer neighbors when making predictions.
- Uniform weighting was tested but gave equal importance to all neighbors, reducing performance.
- Distance-based weighting improved decision quality, especially near class boundaries.

```
knn = KNeighborsClassifier(n_neighbors = 7, p = 1, weights = 'distance', metric = 'manhattan')

knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)

print("classification report")
print(classification_report(y_test, y_pred))

cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(8,6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix - knn')
plt.show()

joblib.dump(knn, 'knn.pkl')
```

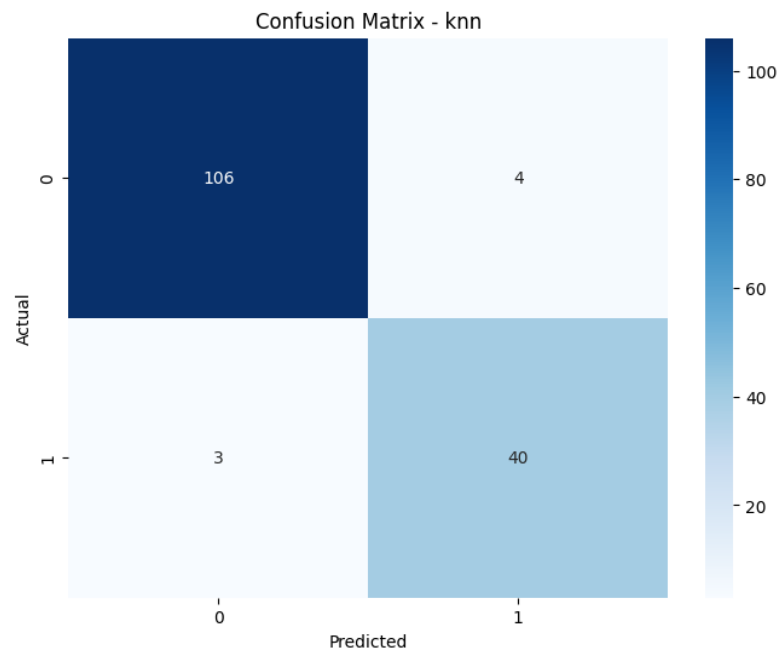
KNN Model Evaluation

- Accuracy: 95%
- Class 0: Precision 0.97, Recall 0.96, F1-score 0.97
- Class 1: Precision 0.91, Recall 0.93, F1-score 0.92
- Confusion Matrix:
 - Correct predictions: 106 (class 0), 40 (class 1)
 - Misclassifications are low, mainly in class 1
- Insight: Model shows strong overall performance with good class separation and reliable generalization

```
classification report
precision    recall  f1-score   support

     0       0.97     0.96     0.97     110
     1       0.91     0.93     0.92      43

 accuracy          0.95     153
 macro avg          0.94     153
 weighted avg       0.95     153
```



7- testing on test data

- Tested all trained models on unseen test data.
- Applied the same scaling and preprocessing used during training.
- Evaluated performance using confusion matrix and classification report.
- Compared models to identify the best generalizing model.

```
print("DT model")
x_test=df_test_final.drop(columns='price', axis=1)
y_test=df_test_final['price']
x_test=scaler.transform(x_test)
y_pred=DT.predict(x_test)
print("confusion metrix")
print(confusion_matrix(y_test, y_pred))
print("classification report",classification_report(y_test, y_pred))
print("SVM model")
y_pred=SVM.predict(x_test)
print("confusion metrix")
print(confusion_matrix(y_test, y_pred))
print("classification report",classification_report(y_test, y_pred))
print("RF model")
y_pred=rf.predict(x_test)
print("confusion metrix")
print(confusion_matrix(y_test, y_pred))
print("classification report",classification_report(y_test, y_pred))
print("XGBoosting model")
y_pred=XGB.predict(x_test)
print("confusion metrix")
print(confusion_matrix(y_test, y_pred))
print("classification report",classification_report(y_test, y_pred))
print("Knn model")
y_pred=knn.predict(x_test)
print("confusion metrix")
print(confusion_matrix(y_test, y_pred))
```

Model Comparison Using F1-Score

F1-score is the harmonic mean of precision and recall, making it a strong metric for evaluating models when class distribution is uneven. It reflects how well a model balances false positives and false negatives, which is especially important in classification problems.

F1-Score Results by Model

- Decision Tree
 - Class 0: 0.93
 - Class 1: 0.82
 - Lower F1-score for class 1 indicates weaker performance on the minority class.
- Support Vector Machine (SVM)
 - Class 0: 0.93
 - Class 1: 0.82

- Similar behavior to DT with limited improvement on class 1.
- Random Forest
 - Class 0: 0.94
 - Class 1: 0.85
 - Better balance between precision and recall than DT and SVM.
- K-Nearest Neighbors (KNN)
 - Class 0: 0.95
 - Class 1: 0.88
 - Strong and consistent performance across both classes.
- XGBoost (Final Model)
 - Class 0: 0.96
 - Class 1: 0.89
 - Highest and most balanced F1-scores for both classes.

Why F1-Score Is Important

- Considers both precision and recall, unlike accuracy.
- Penalizes models that perform well on one class but poorly on another.
- Especially useful when classes are imbalanced.
- Provides a fair and reliable comparison between models.

Final Decision

XGBoost was selected as the final model because it achieved the highest and most balanced F1-scores across both classes, indicating superior overall performance and generalization.

```

XGBoosting model
confusion metrix
[[107  3]
 [ 6 37]]
classification report

```

			precision	recall	f1-score	support
	0	0.95	0.97	0.96	110	
	1	0.93	0.86	0.89	43	
	accuracy			0.94	153	
	macro avg	0.94	0.92	0.93	153	
	weighted avg	0.94	0.94	0.94	153	

```

Knn model
confusion metrix
[[106  4]
 [ 6 37]]
classification report

```

			precision	recall	f1-score	support
	0	0.95	0.96	0.95	110	
	1	0.90	0.86	0.88	43	
	accuracy			0.93	153	
	macro avg	0.92	0.91	0.92	153	
	weighted avg	0.93	0.93	0.93	153	