📱 Mobile Price Range Classification (ML Project)

This project predicts the price range (0 to 3) of mobile phones based on their specifications using machine learning models such as Logistic Regression, SVM, and Random Forest. The goal is to build and compare models, tune them using GridSearchCV, and evaluate their accuracy.

About Dataset

Context

- Bob has started his own mobile company. He wants to give tough fight to big companies like Apple,Samsung etc.He does not know how to estimate price of mobiles his company creates. In this competitive mobile phone market you cannot simply assume things. To solve this problem he collects sales data of mobile phones of various companies.Bob wants to find out some relation between features of a mobile phone(eg:- RAM,Internal Memory etc) and its selling price. But he is not so good at Machine Learning. So he needs your help to solve this problem.

- In this problem you do not have to predict actual price but a price range indicating how high the price is

Step 0: Import All Required Libraries

In [1]:
```
# For Data handling
import pandas as pd
import numpy as np

# For Visualization
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

In [2]:
```
# For Machine Learning Models
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_ma

import warnings
warnings.filterwarnings("ignore")
```

Step 1: Load and Understand the Dataset

- **Load the Data**

In [3]:
```
Train_df = pd.read_csv("train.csv")
Test_df = pd.read_csv("test.csv")
```

In [4]:
```
Train_df.head()
```

Out[4]:

	battery_power	blue	clock_speed	dual_sim	fc	four_g	int_memory	m_dep	mobile_wt
0	842	0	2.2	0	1	0	7	0.6	188
1	1021	1	0.5	1	0	1	53	0.7	136
2	563	1	0.5	1	2	1	41	0.9	145
3	615	1	2.5	0	0	0	10	0.8	131
4	1821	1	1.2	0	13	1	44	0.6	141

5 rows × 21 columns

- **Understand the Structure**

In [5]:
```
print("Train Shape:", Train_df.shape)
print("Test Shape:", Test_df.shape)
```

```
Train Shape: (2000, 21)
Test Shape: (1000, 21)
```

- **Check Column Info & Missing Data**

In [6]:
```
Train_df.info()
print("\nMissing values in train:\n", Train_df.isnull().sum())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2000 entries, 0 to 1999
Data columns (total 21 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   battery_power  2000 non-null   int64  
 1   blue           2000 non-null   int64  
 2   clock_speed    2000 non-null   float64
 3   dual_sim       2000 non-null   int64  
 4   fc             2000 non-null   int64  
 5   four_g         2000 non-null   int64  
 6   int_memory     2000 non-null   int64  
 7   m_dep          2000 non-null   float64
 8   mobile_wt      2000 non-null   int64  
 9   n_cores        2000 non-null   int64  
 10  pc             2000 non-null   int64  
 11  px_height      2000 non-null   int64  
 12  px_width       2000 non-null   int64  
 13  ram            2000 non-null   int64  
 14  sc_h           2000 non-null   int64  
 15  sc_w           2000 non-null   int64  
 16  talk_time      2000 non-null   int64  
 17  three_g        2000 non-null   int64  
 18  touch_screen   2000 non-null   int64  
 19  wifi           2000 non-null   int64  
 20  price_range    2000 non-null   int64  
dtypes: float64(2), int64(19)
memory usage: 328.3 KB

Missing values in train:
 battery_power    0
blue             0
clock_speed      0
dual_sim         0
fc               0
four_g           0
int_memory       0
m_dep            0
mobile_wt        0
n_cores          0
pc               0
px_height        0
px_width         0
ram              0
sc_h             0
sc_w             0
talk_time        0
three_g          0
touch_screen     0
wifi             0
price_range      0
dtype: int64
```

In [7]:

```
Test_df.info()
print("\nMissing values in test:\n", Test_df.isnull().sum())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 21 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   id             1000 non-null   int64  
 1   battery_power  1000 non-null   int64  
 2   blue           1000 non-null   int64  
 3   clock_speed    1000 non-null   float64
 4   dual_sim       1000 non-null   int64  
 5   fc             1000 non-null   int64  
 6   four_g         1000 non-null   int64  
 7   int_memory     1000 non-null   int64  
 8   m_dep          1000 non-null   float64
 9   mobile_wt      1000 non-null   int64  
 10  n_cores        1000 non-null   int64  
 11  pc             1000 non-null   int64  
 12  px_height      1000 non-null   int64  
 13  px_width       1000 non-null   int64  
 14  ram            1000 non-null   int64  
 15  sc_h           1000 non-null   int64  
 16  sc_w           1000 non-null   int64  
 17  talk_time      1000 non-null   int64  
 18  three_g        1000 non-null   int64  
 19  touch_screen   1000 non-null   int64  
 20  wifi           1000 non-null   int64  
dtypes: float64(2), int64(19)
memory usage: 164.2 KB

Missing values in test:
 id                0
battery_power    0
blue             0
clock_speed      0
dual_sim         0
fc               0
four_g           0
int_memory       0
m_dep            0
mobile_wt        0
n_cores          0
pc               0
px_height        0
px_width         0
ram              0
sc_h             0
sc_w             0
talk_time        0
three_g          0
touch_screen     0
wifi             0
dtype: int64
```

- **Check the Target Column(Price Range)**

In [8]:
```
Train_df["price_range"].value_counts()
```

Out[8]:
```
price_range
1    500
2    500
3    500
0    500
Name: count, dtype: int64
```

Step 2: EDA (Exploratory Data Analysis)

Our goal here: Understand relationships between features and the target (price_range) using plots & stats.

- **View Summary Stats**

In [9]:
```
Train_df.describe()
```

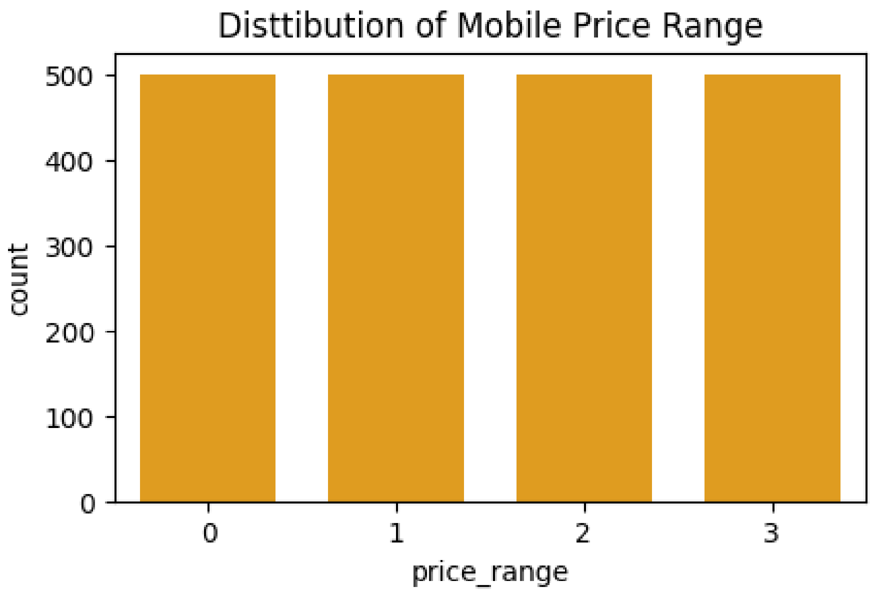
Out[9]:

	battery_power	blue	clock_speed	dual_sim	fc	four_g	int_r
count	2000.000000	2000.0000	2000.000000	2000.000000	2000.000000	2000.000000	2000
mean	1238.518500	0.4950	1.522250	0.509500	4.309500	0.521500	32
std	439.418206	0.5001	0.816004	0.500035	4.341444	0.499662	18
min	501.000000	0.0000	0.500000	0.000000	0.000000	0.000000	2
25%	851.750000	0.0000	0.700000	0.000000	1.000000	0.000000	16
50%	1226.000000	0.0000	1.500000	1.000000	3.000000	1.000000	32
75%	1615.250000	1.0000	2.200000	1.000000	7.000000	1.000000	48
max	1998.000000	1.0000	3.000000	1.000000	19.000000	1.000000	64

8 rows × 21 columns

- **Visualize Target Distribution**

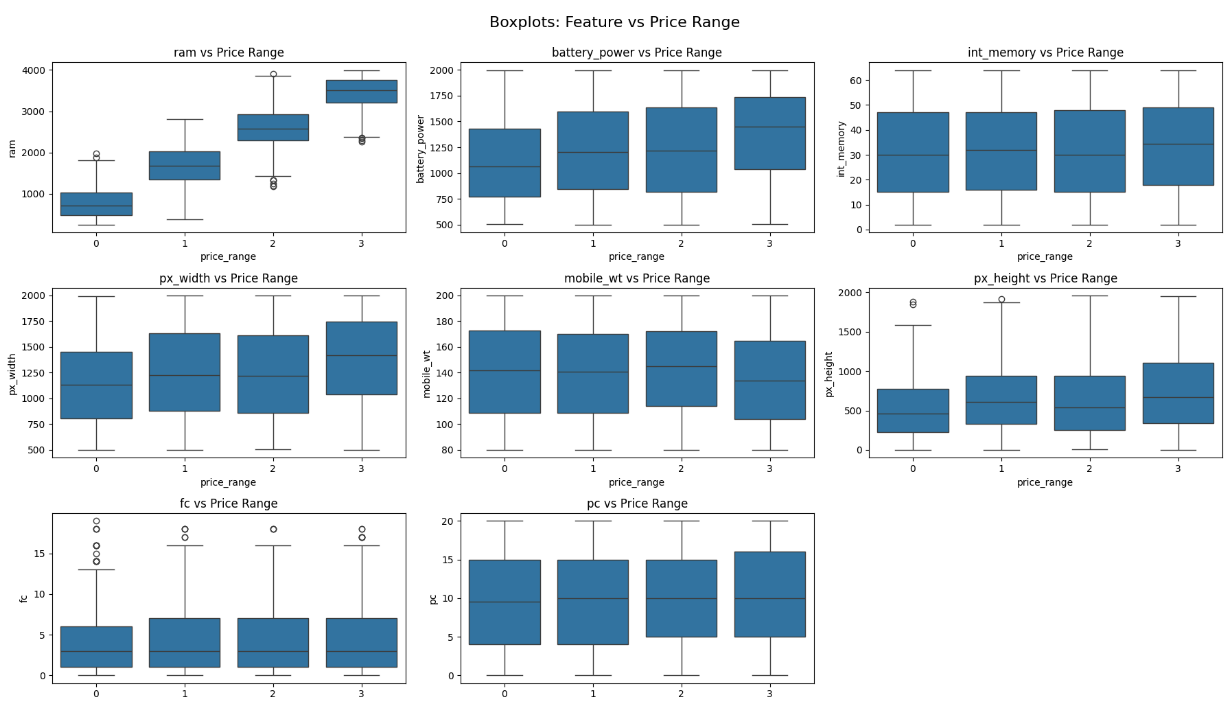
In [10]:
```
plt.figure(figsize=(5,3))
sns.countplot(x="price_range", data=Train_df, color="orange", gap=0.1)
plt.title("Disttibution of Mobile Price Range")
plt.show()
```



- **Feature vs Price Range (Box Plots)**

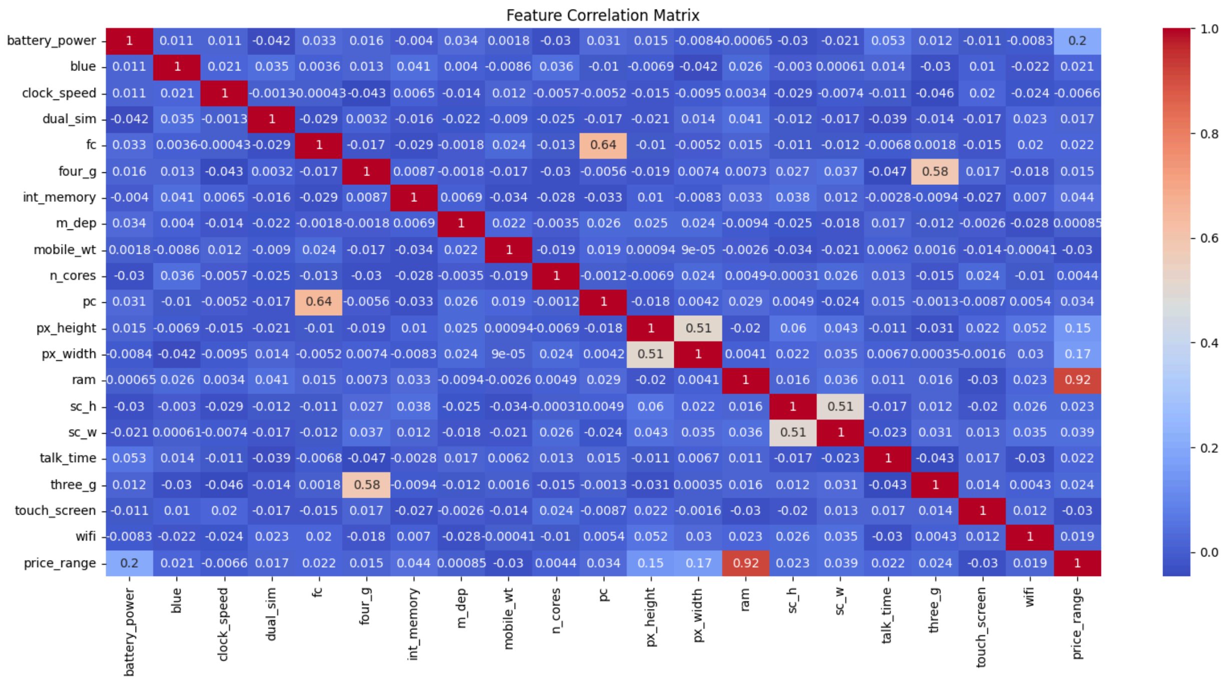
In [11]:
```
features = ['ram', 'battery_power', 'int_memory', 'px_width', 'mobile_wt', 'px_
plt.figure(figsize=(18,10))
for i,feat in enumerate(features):
    plt.subplot(3, 3 , i+1)
    sns.boxplot(x='price_range', y=feat, data=Train_df)
    plt.title(f"{feat} vs Price Range")
    plt.tight_layout()

plt.suptitle("Boxplots: Feature vs Price Range", fontsize=16, y=1.03)
plt.show()
```



- **Correlation Heatmap**

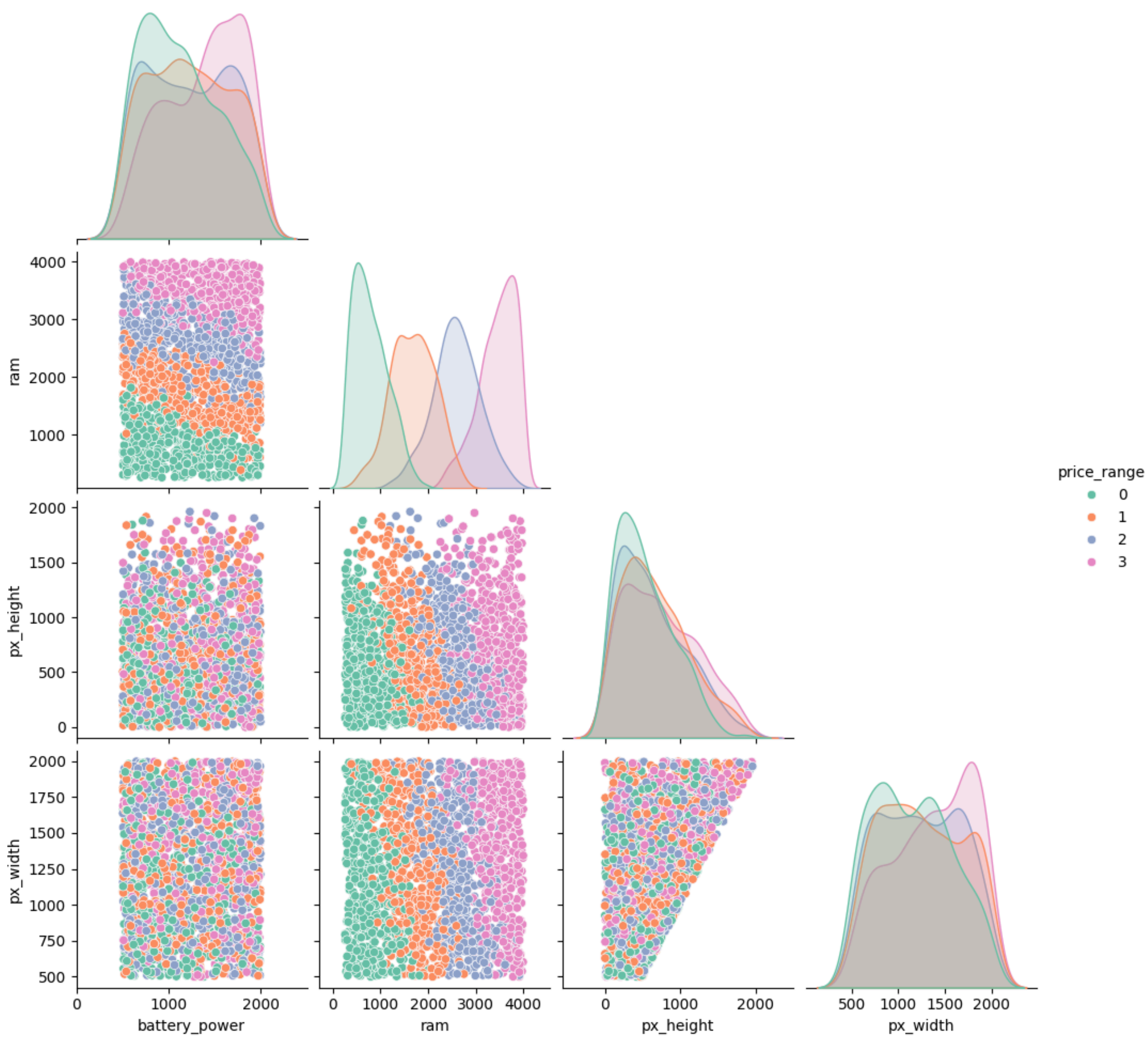
In [12]:
```
plt.figure(figsize=(18, 8))
sns.heatmap(Train_df.corr(), annot=True, cmap='coolwarm')
plt.title("Feature Correlation Matrix")
plt.show()
```



- **Pairplot**

In [13]:
```
sns.pairplot(Train_df[['battery_power', 'ram', 'px_height', 'px_width', 'price_
```

Out[13]: <seaborn.axisgrid.PairGrid at 0x710c8e147410>



Step 3: Preprocessing + Feature Scaling

- **Split Data into Features & Labels**

In [14]:
```
x = Train_df.drop("price_range", axis=1)
y = Train_df["price_range"]
```

- **Train-Test Split for Validation**

In [15]:
```
x_train, x_val, y_train, y_val = train_test_split(x, y, test_size=0.2, random_s
```

In [16]:
```
x_train.shape
```

Out[16]: (1600, 20)

In [17]:
```
x_val.shape
```

Out[17]: (400, 20)

- **Feature Scaling (Logistic & SVM need it!--> StandardScaler or Min Max Scalling, etc)**

In [18]:
```
scaler = StandardScaler()
x_train_scaled = scaler.fit_transform(x_train)
x_val_scaled = scaler.transform(x_val)
```

- **Prepare Test.csv for Final Prediction Later**

In [19]:
```
x_test_final = Test_df.drop("id", axis=1)
x_test_final = scaler.transform(x_test_final)
```

Step 4: Train & Compare Multiple Models

- **Logistic Regression**

In [20]:
```
log_model = LogisticRegression(multi_class='multinomial', solver='lbfgs', max_i
log_model.fit(x_train_scaled, y_train)
log_preds = log_model.predict(x_val_scaled)
```

- **Support Vector Machine (SVM)**

In [21]:
```
svm_model = SVC(kernel="rbf", C=1, gamma='scale')
svm_model.fit(x_train_scaled, y_train)
svm_preds = svm_model.predict(x_val_scaled)
```

- **Random Forest (No Scaling Needed)**

In [22]:
```
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(x_train, y_train)
rf_preds = rf_model.predict(x_val)
```

- **Evaluate All Models**

In [23]:
```
models = {
    "Logistic Regression": log_preds,
    "Support Vector Machine": svm_preds,
    "Random Forest": rf_preds
}

for name, preds in models.items():
    print(f"----- {name} -----")
    print("Accuracy:", accuracy_score(y_val, preds))
    print(classification_report(y_val, preds))
    print()
```

```
----- Logistic Regression -----
Accuracy: 0.975
              precision    recall  f1-score   support

           0       1.00      0.96      0.98       105
           1       0.94      1.00      0.97        91
           2       0.99      0.95      0.97        92
           3       0.97      0.99      0.98       112

    accuracy                           0.97       400
   macro avg       0.98      0.97      0.97       400
weighted avg       0.98      0.97      0.98       400


----- Support Vector Machine -----
Accuracy: 0.8925
              precision    recall  f1-score   support

           0       0.95      0.93      0.94       105
           1       0.80      0.89      0.84        91
           2       0.84      0.82      0.83        92
           3       0.96      0.92      0.94       112

    accuracy                           0.89       400
   macro avg       0.89      0.89      0.89       400
weighted avg       0.90      0.89      0.89       400


----- Random Forest -----
Accuracy: 0.8925
              precision    recall  f1-score   support

           0       0.95      0.96      0.96       105
           1       0.89      0.87      0.88        91
           2       0.78      0.87      0.82        92
           3       0.94      0.87      0.90       112

    accuracy                           0.89       400
   macro avg       0.89      0.89      0.89       400
weighted avg       0.90      0.89      0.89       400
```

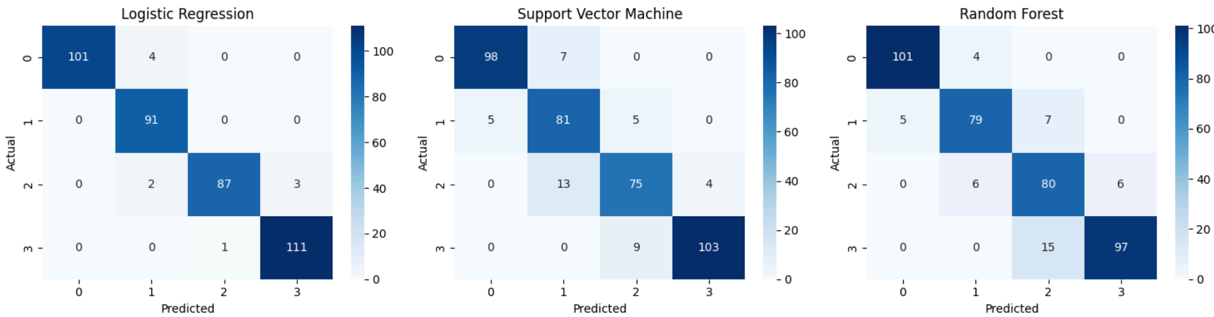
- **Confusion Matrix Heatmap (Optional but Cool)**

In [24]:

```python
plt.figure(figsize=(15, 4))

for i, (name, preds) in enumerate(models.items()):
    plt.subplot(1, 3, i+1)
    cm = confusion_matrix(y_val, preds)
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
    plt.title(name)
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
```

```
plt.tight_layout()
plt.show()
```



Step 5: Grid Search for Hyperparameter Tuning

- ### Grid Search for SVM Model

In [25]:

```
svm_gs = SVC(random_state=42)

param_grid_svm = {
    'C': [0.1, 1, 10],
    'gamma': [0.001, 0.01, 0.1],
    'kernel': ['rbf']
}

grid_search_svm = GridSearchCV(estimator=svm_gs,
                               param_grid=param_grid_svm,
                               cv=5,
                               n_jobs=-1,
                               scoring='accuracy',
                               verbose=2)

grid_search_svm.fit(x_train_scaled, y_train)
print("Best Parameters:", grid_search_svm.best_params_)

best_svm = grid_search_svm.best_estimator_
y_pred_best_svm = best_svm.predict(x_val_scaled)

print("Accuracy of Tuned SVM:", accuracy_score(y_val, y_pred_best_svm))
print(classification_report(y_val, y_pred_best_svm))
```

```
Fitting 5 folds for each of 9 candidates, totalling 45 fits
Best Parameters: {'C': 10, 'gamma': 0.01, 'kernel': 'rbf'}
Accuracy of Tuned SVM: 0.9275
              precision    recall  f1-score   support

           0       0.97      0.94      0.96       105
           1       0.85      0.95      0.90        91
           2       0.91      0.87      0.89        92
           3       0.97      0.95      0.96       112

    accuracy                           0.93       400
   macro avg       0.93      0.93      0.93       400
weighted avg       0.93      0.93      0.93       400
```

- **Grid Search for Random Forest Model**

In [26]:
```
rf = RandomForestClassifier(random_state=42)

param_grid = {
    'n_estimators': [50, 100],
    'max_depth': [10, 20, None],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 2]
}

grid_search = GridSearchCV(estimator= rf,
                           param_grid=param_grid,
                           cv=5,
                           scoring='accuracy',
                           n_jobs=-1,
                           verbose=2)

grid_search.fit(x_train, y_train)
print("Best Parameters:", grid_search.best_params_)

best_rf = grid_search.best_estimator_
y_pred_best_rf = best_rf.predict(x_val)

print("Accuracy of Tuned RF:", accuracy_score(y_val, y_pred_best_rf))
print(classification_report(y_val, y_pred_best_rf))
```

```
Fitting 5 folds for each of 24 candidates, totalling 120 fits
Best Parameters: {'max_depth': 20, 'min_samples_leaf': 1, 'min_samples_split':
2, 'n_estimators': 100}
Accuracy of Tuned RF: 0.8875
              precision    recall  f1-score   support

           0       0.95      0.96      0.96       105
           1       0.89      0.86      0.87        91
           2       0.77      0.87      0.82        92
           3       0.94      0.86      0.90       112

    accuracy                           0.89       400
   macro avg       0.89      0.89      0.89       400
weighted avg       0.89      0.89      0.89       400
```

Step 6: Final Prediction on test.csv

- **Predict on Final Test Data**

In [27]:
```
final_predictions = log_model.predict(x_test_final)
```

- **Combine with Test IDs (if needed)**

In [28]:
```
submission_df = pd.DataFrame({
    'id': Test_df['id'],
    'price_range': final_predictions
})
```

In [29]:
```
submission_df.to_csv("final_submission.csv", index=False)
print("Submission file saved as final_submission.csv")
```

```
Submission file saved as final_submission.csv
```

✅ Final Results

Model	Accuracy
Logistic Regression	97.5%
Tuned SVM	92.75%
Random Forest	89.2%

📌 **Conclusion**: Logistic Regression performed best on this dataset. This suggests that the features are well-separated and a linear model is sufficient for high accuracy.

📤 Final Test Predictions

The best model was used to predict values on the unseen test dataset. These predictions can be used for further submission or validation.

Author:

- ***Prince Raj***