CourseWork Notebook

California Housing Prices Dataset Preprocessing

In this notebook, we will preprocess the California Housing Prices dataset to prepare it for machine learning model development.

1. Loading the Dataset

In [147…

```
import pandas as pd
data = pd.read_csv('housing.csv')
```

The California Housing Dataset was loaded from a publicly available source on Kaggle. In the code above we are loading the dataset ready to be used.

2. Exploratory Data Analysis (EDA)

Conducting EDA helps in understanding the structure and contents of the dataset, identifying potential issues such as missing values or outliers, and providing insights into the relationship between features.

Initial Inspection

An initial inspection was done to understand the data types, check for missing values, and assess the basic statistics of the features.

In [150…

```
# Display the first few rows of the dataset
print("Head")
data.head()

# Get a summary of the dataset, including data types and missing values
print("Info")
data.info()

# Statistical summary of the dataset
print("Describe")
data.describe()
```

```
Head
Info
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   longitude           20640 non-null  float64
 1   latitude            20640 non-null  float64
 2   housing_median_age  20640 non-null  int64
 3   total_rooms         20640 non-null  int64
 4   total_bedrooms      20433 non-null  float64
 5   population          20640 non-null  int64
 6   households          20640 non-null  int64
 7   median_income       20640 non-null  float64
 8   median_house_value  20640 non-null  int64
 9   ocean_proximity     20640 non-null  object
dtypes: float64(4), int64(5), object(1)
memory usage: 1.6+ MB
Describe
```

Out[150…

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553
std	2.003532	2.135952	12.585558	2181.615252	421.385070
min	-124.350000	32.540000	1.000000	2.000000	1.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000

From the summary we can see some basic information we can interpret:

The dataset consists of 20640 rows and 10 columns (features, including continuous variables like longitude, latitude, median_income, and the target variable median_house_value.

Missing values were detected in the total_bedrooms column.

Continuous variables such as total_rooms and population showed a wide range, indicating potential outliers that could affect model performance.

3. Handling Missing Values

The feature total_bedrooms contained missing values. To maintain the integrity of the dataset, these missing values were imputed using the median value of the feature. Median imputation was chosen to reduce the impact of outliers on the imputed values.

In [153…
```
# Fill missing values in 'total_bedrooms' with the median value
data['total_bedrooms'] = data['total_bedrooms'].fillna(data['total_bedrooms'].me
```

4. Handling Categorical Variables

The dataset contains one categorical variable, ocean_proximity, which describes the proximity of the neighborhood to the ocean. Since most machine learning models work with numerical data, this categorical feature was transformed into numeric form using one-hot encoding. One-hot encoding creates new binary columns for each unique category in the ocean_proximity feature.

In [155…
```
# One-hot encode the 'ocean_proximity' column
data = pd.get_dummies(data, columns=['ocean_proximity'], drop_first=True)
```

5. Handling Outliers

Outliers in the dataset can distort model performance, especially in regression tasks. Visual inspection through boxplots and histograms helped identify potential outliers in continuous features like total_rooms, population, and households.

Outliers can distort model performance, especially in regression tasks and models like SVM and Neural Networks. We will detect and remove outliers using th**Interquartile Range (IQ** method for continuous features like `total_rooms` and `population`. Any values outside 1.5 times the IQR from the first and third quartiles will be considered outliers and removed.

In [157…
```
# Detecting and removing outliers using IQR for 'total_rooms'
Q1 = data['total_rooms'].quantile(0.25)
Q3 = data['total_rooms'].quantile(0.75)
IQR = Q3 - Q1

# Define lower and upper bounds for outliers
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Filter out outliers for 'total_rooms'
data = data[(data['total_rooms'] >= lower_bound) & (data['total_rooms'] <= upper

# Detecting and removing outliers using IQR for 'population'
Q1 = data['population'].quantile(0.25)
Q3 = data['population'].quantile(0.75)
IQR = Q3 - Q1

lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Filter out outliers for 'population'
data = data[(data['population'] >= lower_bound) & (data['population'] <= upper_b
```

6. Feature Engineering

To enrich the dataset, several new features were engineered from existing ones. These derived features help models capture additional relationships within the data:

- `rooms_per_household` : The average number of rooms per household.
- `bedrooms_per_room` : The ratio of bedrooms to rooms.
- `population_per_household` : The average number of people per household.

In [159…
```
# Create new features
data['rooms_per_household'] = data['total_rooms'] / data['households']
data['bedrooms_per_room'] = data['total_bedrooms'] / data['total_rooms']
data['population_per_household'] = data['population'] / data['households']
```

7. Normalization and Standardization

Many machine learning algorithms, such as SVM and Neural Networks, perform better when features are normalized or standardized. Normalization ensures that all features are on the same scale, preventing features with large ranges (like population) from dominating the models. Continuous variables like median_income, housing_median_age, and population were standardized to have a mean of 0 and a standard deviation of 1.

In [161…
```
from sklearn.preprocessing import StandardScaler

# Select the continuous variables for scaling
continuous_features = ['median_income', 'housing_median_age', 'total_rooms', 'to

# Initialize the StandardScaler
scaler = StandardScaler()

# Apply scaling
data[continuous_features] = scaler.fit_transform(data[continuous_features])
```

8. Creating the Classification Target

In this section, we will create a new target variable for the classification task. The median_house_value column, which represents the median house price, will be converted into a binary classification target. We'll classify houses as "affordable" (0) or "expensive" (1), using a threshold of $300,000.

In [163…
```
import numpy as np
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler

# Creating the classification target: 1 for "expensive", 0 for "affordable"
data['price_category'] = np.where(data['median_house_value'] > 300000, 1, 0)

# Check the distribution of the new target variable
data['price_category'].value_counts()
```

Out[163…
```
price_category
0    15251
1     3498
Name: count, dtype: int64
```

9. Train-Test Split

Before applying machine learning models, the dataset needs to be split into training and testing sets. We'll use 70% of the data for training and 30% for testing, ensuring that the model is evaluated on unseen data.

In [165…
```
from sklearn.model_selection import train_test_split

# Define features (X) and target (y) for regression and classification
X = data.drop(columns=['median_house_value', 'price_category'])
y_regression = data['median_house_value']
y_classification = data['price_category']

# Split the data for regression
X_train_reg, X_test_reg, y_train_reg, y_test_reg = train_test_split(X, y_regress

# Split the data for classification
X_train_clf, X_test_clf, y_train_clf, y_test_clf = train_test_split(X, y_classif

# Check the shape of the training and testing sets
X_train_reg.shape, X_test_reg.shape, X_train_clf.shape, X_test_clf.shape
```

Out[165…
```
((13124, 15), (5625, 15), (13124, 15), (5625, 15))
```

Final Dataset Preparation

After preprocessing, the dataset is ready for use in the machine learning models. The dataset now consists of both transformed and engineered features, with missing values handled and all continuous variables standardized. The preprocessing ensures that the dataset is clean, structured, and suitable for model training and evaluation.

In [167…
```
# Check the final processed data
data.head()
```

Out[167…

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	
0	-122.23	37.88	0.917835	-1.163092	-1.371840	-1.466437	
2	-122.24	37.85	1.808854	-0.634163	-1.108125	-1.166312	
3	-122.25	37.85	1.808854	-0.808070	-0.913581	-1.059371	
4	-122.25	37.85	1.808854	-0.489992	-0.719037	-1.047297	
5	-122.25	37.85	1.808854	-1.127950	-1.008691	-1.309475	

Summary of the Preprocessing

The preprocessing steps followed for the California Housing Dataset included loading the dataset, conducting exploratory data analysis, handling missing values, transforming categorical variables, creating new features, and standardizing the continuous variables. These steps ensure that the dataset is ready for machine learning model development, providing a solid foundation for both regression and classification tasks.

The Code in One Cell

In [170…

```
import pandas as pd
import numpy as np
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler

data = pd.read_csv('housing.csv')

# Display the first few rows of the dataset
print("Head")
data.head()

# Get a summary of the dataset, including data types and missing values
print("Info")
data.info()

# Statistical summary of the dataset
print("Describe")
data.describe()

# Fill missing values in 'total_bedrooms' with the median value
data['total_bedrooms'] = data['total_bedrooms'].fillna(data['total_bedrooms'].me

# One-hot encode the 'ocean_proximity' column
data = pd.get_dummies(data, columns=['ocean_proximity'], drop_first=True)

# Detecting and removing outliers using IQR for 'total_rooms'
Q1 = data['total_rooms'].quantile(0.25)
Q3 = data['total_rooms'].quantile(0.75)
IQR = Q3 - Q1

# Define lower and upper bounds for outliers
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Filter out outliers for 'total_rooms'
data = data[(data['total_rooms'] >= lower_bound) & (data['total_rooms'] <= upper

# Detecting and removing outliers using IQR for 'population'
Q1 = data['population'].quantile(0.25)
Q3 = data['population'].quantile(0.75)
IQR = Q3 - Q1
```

```
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Filter out outliers for 'population'
data = data[(data['population'] >= lower_bound) & (data['population'] <= upper_b

# Create new features
data['rooms_per_household'] = data['total_rooms'] / data['households']
data['bedrooms_per_room'] = data['total_bedrooms'] / data['total_rooms']
data['population_per_household'] = data['population'] / data['households']

from sklearn.preprocessing import StandardScaler

# Select the continuous variables for scaling
continuous_features = ['median_income', 'housing_median_age', 'total_rooms', 'to

# Initialize the StandardScaler
scaler = StandardScaler()

# Apply scaling
data[continuous_features] = scaler.fit_transform(data[continuous_features])

# Creating the classification target: 1 for "expensive", 0 for "affordable"
data['price_category'] = np.where(data['median_house_value'] > 300000, 1, 0)

# Check the distribution of the new target variable
data['price_category'].value_counts()

from sklearn.model_selection import train_test_split

# Define features (X) and target (y) for regression and classification
X = data.drop(columns=['median_house_value', 'price_category'])
y_regression = data['median_house_value']
y_classification = data['price_category']

# Split the data for regression
X_train_reg, X_test_reg, y_train_reg, y_test_reg = train_test_split(X, y_regress

# Split the data for classification
X_train_clf, X_test_clf, y_train_clf, y_test_clf = train_test_split(X, y_classif

# Check the shape of the training and testing sets
X_train_reg.shape, X_test_reg.shape, X_train_clf.shape, X_test_clf.shape


# Check the final processed data
data.head()
```

```
Head
Info
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   longitude           20640 non-null  float64
 1   latitude            20640 non-null  float64
 2   housing_median_age  20640 non-null  int64
 3   total_rooms         20640 non-null  int64
 4   total_bedrooms      20433 non-null  float64
 5   population          20640 non-null  int64
 6   households          20640 non-null  int64
 7   median_income       20640 non-null  float64
 8   median_house_value  20640 non-null  int64
 9   ocean_proximity     20640 non-null  object
dtypes: float64(4), int64(5), object(1)
memory usage: 1.6+ MB
Describe
```

Out[170…

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population
0	-122.23	37.88	0.917835	-1.163092	-1.371840	-1.466437
2	-122.24	37.85	1.808854	-0.634163	-1.108125	-1.166312
3	-122.25	37.85	1.808854	-0.808070	-0.913581	-1.059371
4	-122.25	37.85	1.808854	-0.489992	-0.719037	-1.047297
5	-122.25	37.85	1.808854	-1.127950	-1.008691	-1.309475

Model Development

In this section, we implement and evaluate three machine learning algorithms—Decision Trees, Support Vector Machines (SVM), and Neural Networks (Multi-Layer Perceptron, MLP)—on the preprocessed dataset. The objective is to predict whether a neighborhood is classified as "expensive" (house prices above $300,000) or "affordable" (housepricesbeloworequalto300,000) based on various features. We will also perform hyperparameter tuning for each model to improve performance.

1. Machine Learning Algorithms

We selected three distinct machine learning algorithms that represent different paradigms of machine learning. Each algorithm brings unique strengths, and their performance on this dataset will provide insights into the suitability of each approach for the classification task.

1.1 Decision Trees

Decision Trees classify data by recursively splitting the dataset based on feature values, forming a tree structure where each leaf represents a classification decision. The algorithm is easy to interpret, making it an ideal choice for understanding the relationships between the features and the target variable. However, decision trees are prone to overfitting, which is why hyperparameter tuning is crucial.

What We Are Testing for in the Decision Tree Model

Target Variable: The binary classification target, price_category, which classifies neighborhoods as "expensive" (1) or "affordable" (0).

Features: All features, including longitude, latitude, median_income, total_rooms, and engineered features like rooms_per_household and bedrooms_per_room.

Hyperparameters Tuned:

max_depth: Controls how deep the tree can grow. A deeper tree can model complex relationships but may overfit the data.

min_samples_split: The minimum number of samples required to split an internal node. Higher values prevent overfitting by limiting tree growth.

min_samples_leaf: The minimum number of samples required to be at a leaf node, controlling the complexity of the model.

In [173…

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV

# Initialize the Decision Tree Classifier
decision_tree = DecisionTreeClassifier()

# Define a parameter grid for hyperparameter tuning
param_grid_dt = {'max_depth': [3, 5, 10, None],
                 'min_samples_split': [2, 10, 20],
                 'min_samples_leaf': [1, 5, 10]}

# Perform hyperparameter tuning using GridSearchCV with 5-fold cross-validation
grid_search_dt = GridSearchCV(decision_tree, param_grid_dt, cv=5, scoring='accur
grid_search_dt.fit(X_train_clf, y_train_clf)

# Retrieve and print the best hyperparameters
best_dt_params = grid_search_dt.best_params_
print(f"Best Hyperparameters for Decision Tree: {best_dt_params}")

# Train the best model with the optimal hyperparameters
best_dt_model = grid_search_dt.best_estimator_
best_dt_model.fit(X_train_clf, y_train_clf)

# Evaluate the trained model on the test data
dt_accuracy = best_dt_model.score(X_test_clf, y_test_clf)
print(f"Decision Tree Accuracy: {dt_accuracy:.4f}")
```

```
Best Hyperparameters for Decision Tree: {'max_depth': None, 'min_samples_leaf': 1
0, 'min_samples_split': 20}
Decision Tree Accuracy: 0.9033
```

Explanation:

The decision tree model was tuned using a GridSearchCV approach to find the optimal combination of hyperparameters such as max_depth, min_samples_split, and min_samples_leaf. These parameters control the complexity of the tree and prevent overfitting. A cross-validation procedure (5-fold) was used to ensure that the model generalizes well to unseen data.

Evaluation:

The model's accuracy on the test set is evaluated, and the best-performing decision tree is chosen based on the hyperparameters that maximize the cross-validated accuracy.

1.2 Support Vector Machine (SVM)

Support Vector Machines (SVM) classify data by finding the optimal hyperplane that separates the data points of different classes with the maximum margin. SVM is particularly useful when the data is not linearly separable, as it uses kernel functions to map the data into higher-dimensional spaces. The choice of kernel function and other hyperparameters significantly affects SVM's performance.

What We Are Testing for in the SVM Model

Target Variable: The binary classification target, price_category.

Features: All features, including both continuous variables like median_income and categorical variables (after encoding) like ocean_proximity.

Hyperparameters Tuned:

C: The regularisation parameter that controls the trade-off between maximizing the margin and minimizing classification error.

kernel: Specifies the kernel function used to transform the data. Common kernels are linear and RBF (Radial Basis Function).

gamma: Determines the influence of a single training example. Higher values lead to tighter decision boundaries, while lower values result in smoother decision boundaries.

In [176…

```
from sklearn.svm import SVC

# Initialize the SVM model
svm_model = SVC()

# Define a parameter grid for tuning the hyperparameters of the SVM
param_grid_svm = {'C': [0.1, 1, 10],
```

```
                  'kernel': ['linear', 'rbf'],
                  'gamma': ['scale', 'auto']}

# Perform hyperparameter tuning using GridSearchCV with 5-fold cross-validation
grid_search_svm = GridSearchCV(svm_model, param_grid_svm, cv=5, scoring='accurac
grid_search_svm.fit(X_train_clf, y_train_clf)

# Retrieve and print the best hyperparameters
best_svm_params = grid_search_svm.best_params_
print(f"Best Hyperparameters for SVM: {best_svm_params}")

# Train the best SVM model with the optimal hyperparameters
best_svm_model = grid_search_svm.best_estimator_
best_svm_model.fit(X_train_clf, y_train_clf)

# Evaluate the trained model on the test data
svm_accuracy = best_svm_model.score(X_test_clf, y_test_clf)
print(f"SVM Accuracy: {svm_accuracy:.4f}")
```

```
Best Hyperparameters for SVM: {'C': 10, 'gamma': 'auto', 'kernel': 'rbf'}
SVM Accuracy: 0.9164
```

Explanation:

The SVM model was tuned using the parameters C, kernel, and gamma. The C parameter controls the trade-off between maximizing the margin and minimizing classification error, while the kernel determines the transformation used for mapping the data into higher dimensions. The RBF kernel was tested for non-linear relationships, while the linear kernel was included for simpler cases. Gamma controls the influence of a single training example, determining the shape of the decision boundary.

Evaluation:

After hyperparameter tuning, the SVM model is evaluated on the test set, and the results are compared to other models.

1.3 Neural Network (MLP)

Neural Networks (MLP) are capable of learning complex patterns in data by using multiple layers of interconnected neurons. Each layer transforms the data, allowing the model to capture non-linear relationships. Neural Networks are particularly powerful when dealing with complex datasets, but they can be computationally expensive to train. The architecture (number of layers and neurons) and activation function play a key role in their performance.

What We Are Testing for in the Neural Network (MLP) Model

Target Variable: The binary classification target, price_category.

Features: All features, including engineered features such as rooms_per_household and population_per_household.

Hyperparameters Tuned:

hidden_layer_sizes: Controls the number of neurons and layers in the network. A larger network can capture more complex relationships but requires more data to avoid overfitting.

activation: The activation function used to determine how neurons fire. Common choices are relu (Rectified Linear Unit) and tanh.

alpha: The regularization parameter that helps prevent overfitting by controlling the magnitude of weights in the network.

In [179…

```python
from sklearn.neural_network import MLPClassifier

# Initialize the Neural Network model
mlp_model = MLPClassifier(max_iter=1000)

# Define a parameter grid for tuning the neural network's hyperparameters
param_grid_mlp = {'hidden_layer_sizes': [(50,), (100,), (50, 50)],
                  'activation': ['relu', 'tanh'],
                  'alpha': [0.0001, 0.001, 0.01]}

# Perform hyperparameter tuning using GridSearchCV with 5-fold cross-validation
grid_search_mlp = GridSearchCV(mlp_model, param_grid_mlp, cv=5, scoring='accurac
grid_search_mlp.fit(X_train_clf, y_train_clf)

# Retrieve and print the best hyperparameters
best_mlp_params = grid_search_mlp.best_params_
print(f"Best Hyperparameters for Neural Network: {best_mlp_params}")

# Train the best MLP model with the optimal hyperparameters
best_mlp_model = grid_search_mlp.best_estimator_
best_mlp_model.fit(X_train_clf, y_train_clf)

# Evaluate the trained model on the test data
mlp_accuracy = best_mlp_model.score(X_test_clf, y_test_clf)
print(f"Neural Network Accuracy: {mlp_accuracy:.4f}")
```

```
Best Hyperparameters for Neural Network: {'activation': 'tanh', 'alpha': 0.0001,
'hidden_layer_sizes': (100,)}
Neural Network Accuracy: 0.9028
```

Explanation:

The Multi-Layer Perceptron (MLP) classifier was tuned for parameters like hidden_layer_sizes (number of neurons and layers), activation functions (e.g., ReLU or tanh), and the regularization parameter alpha. These hyperparameters control the complexity and flexibility of the network. The activation function dictates how neurons fire, while the hidden layers control the network's capacity to learn complex patterns.

Evaluation:

The model's accuracy is evaluated on the test data, and its performance is compared to the SVM and Decision Tree models.

2. Hyperparameter Tuning

For each model, hyperparameter tuning was performed using GridSearchCV, which automates the process of testing various hyperparameter combinations to find the optimal configuration. Cross-validation (5-fold) was applied to ensure that the models generalize well to unseen data, reducing the risk of overfitting. The following hyperparameters were tuned for each model:

- Decision Tree: max_depth, min_samples_split, min_samples_leaf
- SVM: C, kernel, gamma
- Neural Networks (MLP): hidden_layer_sizes, activation, alpha

3. Strengths and Weaknesses of Each Model

3.1 Decision Trees

Strengths:

- Interpretability: One of the key advantages of decision trees is their interpretability. They allow you to visualise the decision-making process, making it easy to understand how the model reaches its predictions. This makes decision trees particularly useful when model transparency is important.
- Handling Non-Linear Data: Decision trees are capable of handling non-linear data by splitting the feature space recursively. They can model complex decision boundaries without the need for kernel transformations.
- No Need for Feature Scaling: Unlike models like SVM or neural networks, decision trees do not require feature scaling, as they rely on the relative ordering of feature values.

Weaknesses:

- Prone to Overfitting: Decision trees tend to overfit the data if they are not pruned or limited in depth. This means that they can perform well on training data but may generalise poorly to new, unseen data.
- Instability: Small changes in the dataset can lead to entirely different tree structures, as decision trees are sensitive to the specific splits chosen early in the process.
- Limited Expressiveness for Complex Patterns: While decision trees are flexible, they may struggle to model highly complex patterns in the data without becoming overly complex and prone to overfitting.

3.2 Support Vector Machine (SVM)

Strengths:

- Effective in High-Dimensional Spaces: SVM is well-suited for classification tasks with high-dimensional feature spaces. It can be particularly powerful when the number of features exceeds the number of samples.

Robust to Overfitting with Proper Regularization: By tuning the regularization parameter C, SVM can prevent overfitting, even in complex datasets.

- Flexible Kernel Functions: The use of different kernel functions (e.g., linear, RBF) allows SVM to handle both linear and non-linear relationships in the data. The RBF kernel, in particular, is very powerful for capturing non-linear decision boundaries.
- Generalization: SVM has strong generalization capabilities, especially when tuned properly. It finds the optimal margin between classes, leading to good performance on unseen data.

Weaknesses:

- Computational Complexity: SVM can be computationally expensive, particularly for large datasets. The time complexity increases significantly with the number of samples, especially when using non-linear kernels.
- Sensitive to Feature Scaling: SVM relies heavily on the distances between data points, making it sensitive to feature scaling. Proper normalization or standardization of features is necessary for optimal performance.
- Difficult Hyperparameter Tuning: The performance of SVM is highly dependent on the proper selection of hyperparameters (e.g., C, gamma). Without careful tuning, the model may not achieve optimal results.

3.3 Neural Networks (MLP)

Strengths:

- Ability to Model Complex Relationships: Neural networks are powerful for capturing complex, non-linear relationships in the data. With enough layers and neurons, they can model intricate patterns that simpler models like decision trees or linear models may miss.
- Flexible Architecture: The architecture of a neural network can be adjusted to suit the complexity of the data. The number of hidden layers, neurons, and the choice of activation functions allow the model to be tailored to specific tasks.
- Performance with Large Datasets: Neural networks generally perform well with larger datasets, especially when there is enough data to capture and learn the underlying patterns without overfitting.

Weaknesses:

- Computationally Intensive: Neural networks require significant computational resources, particularly when the network architecture is complex. Training deep

networks can be slow and requires hardware like GPUs for efficiency.

- Requires Large Training Data: Neural networks tend to perform poorly with small datasets, as they are prone to overfitting due to their large capacity. They require large amounts of data to generalize effectively.
- Black Box Nature: Unlike decision trees, neural networks are often considered "black boxes." Their decision-making process is not easily interpretable, making it difficult to understand why a certain prediction was made.
- Sensitive to Hyperparameters: Neural networks require careful tuning of hyperparameters like the learning rate, number of hidden layers, and regularization terms to prevent overfitting and ensure good performance.

4. Reflection on Challenges and Lessons Learned

Throughout this project, several challenges were encountered that helped shape the learning experience and the evaluation of the models.

Challenges Faced:

1. **Hyperparameter Tuning**: Tuning the hyperparameters for each model, especially for Support Vector Machines and Neural Networks, was challenging. SVM's performance is highly dependent on selecting the correct `C` and `gamma` values, while Neural Networks require careful tuning of the architecture (e.g., number of layers, neurons, and activation functions) to avoid overfitting or underfitting.
2. **Computational Complexity**: Neural Networks and SVMs, particularly when using non-linear kernels or deeper architectures, required significantly more computational resources. This led to longer training times, which posed a challenge when iterating through different hyperparameter combinations.
3. **Dealing with Imbalanced Data**: Some imbalance in the `price_category` target variable required careful consideration, as models may bias toward the majority class. While we used cross-validation and tuned hyperparameters, this remained a challenge.

Lessons Learned:

1. **Model Selection for Complex Tasks**: SVM demonstrated superior generalization capabilities, making it ideal for this task. The use of the RBF kernel enabled it to model non-linear patterns, which helped it achieve the highest accuracy of the three models. This reflects the strength of SVM in handling complex classification tasks, especially when feature scaling and dimensionality are considerations.
2. **Trade-offs in Interpretability**: Decision Trees provided a level of interpretability that neither SVM nor Neural Networks could match. While its performance was slightly lower, the ease of understanding how the model made predictions

reinforced the importance of considering interpretability in model selection, especially in scenarios where understanding decisions is crucial.

3. **Neural Networks and Data Requirements**: Neural Networks, while powerful, require substantial amounts of data to perform well. In this project, MLP performed well but did not outperform SVM, likely due to the size and complexity of the dataset. This emphasized the importance of selecting the right model for the available data and resources.

4. **Why SVM Was Best Suited**: SVM was particularly effective because it balanced accuracy, generalization, and computational feasibility for this classification task. Its ability to handle non-linear decision boundaries without overfitting and its performance in high-dimensional spaces made it the best-performing model.

In conclusion, each model had its strengths, but the SVM's robustness and performance made it well-suited for this particular complex classification task.plex classification task.

In [ ]: