Group Number: 8

Section: BM7

Group Members:

- Acosta, John Paolo Miguel
- Gonzales, Dave Justine
- Perez, Ashton Miguel
- Rupisan, Anthony James

```
# Imports
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import gym
import numpy as np
import seaborn as sns
from sklearn.semi_supervised import LabelPropagation
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import KBinsDiscretizer
```

```
# Load the dataset
file_path = '/content/laptop_price - dataset.csv'
df = pd.read_csv(file_path)

# Preview the first few rows of the dataset to understand its structure
df.head()
```

	Company	Product	TypeName	Inches	ScreenResolution	CPU_Company	CPU_Type	CPU_Frequency (GHz)	RAM (GB)	Memory	GPU_Company	GPU_Type	OpS
0	Apple	MacBook Pro	Ultrabook	13.3	IPS Panel Retina Display 2560x1600	Intel	Core i5	2.3	8	128GB SSD	Intel	Iris Plus Graphics 640	mac(
1	Apple	Macbook Air	Ultrabook	13.3	1440x900	Intel	Core i5	1.8	8	128GB Flash Storage	Intel	HD Graphics 6000	mac(

Next steps: Generate code with df | View recommended plots | New interactive sheet

```
# Encode categorical variables with LabelEncoder
categorical_columns = ['Company', 'TypeName', 'ScreenResolution', 'CPU_Company', 'CPU_Type',
                       'Memory', 'GPU_Company', 'GPU_Type', 'OpSys']

# Create a dictionary to store encoders for each categorical column for possible decoding later
label_encoders = {}
for column in categorical_columns:
    le = LabelEncoder()
    df[column + '_encoded'] = le.fit_transform(df[column])
    label_encoders[column] = le

# Display the encoded columns in the dataset
X = df[[col + '_encoded' for col in categorical_columns]]
y = df['Price (Euro)']
```

```
# Define features (X) and label (y)
X = df.drop(columns=['Price (Euro)'] + categorical_columns)
y = df['Price (Euro)']

# Split the dataset into training and testing sets (70% training, 30% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

```
# Display the shapes of the resulting splits
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

```
((892, 14), (383, 14), (892,), (383,))
```

Supervised Learning

Obeservation of Supervised Learning

In supervised learning, a model is trained using a labeled dataset that has labels assigned to the input features and output features. The model learns the relationship between input attributes and target variables to estimate pricing for fresh, unseen data.

The model was evaluated using fresh, unpublished data that represented an HP and an Apple laptop. The accuracy of predictions depends on the quality and amount of the labeled data.When past data with known results may be used to inform future forecasts, such as when predicting pricing, sales, or any other continuous variable, this supervised learning approach is helpful. Based on their features, the Random Forest model can be used to forecast the cost of new computers or comparable goods.

When a labeled dataset is available for a problem, supervised learning is advantageous. The Random Forest model makes good use of the correlation between prices and features. We can assess its performance and make further refinements to the model to achieve more precise predictions by utilizing the metrics MAE, MSE, and R^2.

```
# Define features (X) and label (y)
X = df.drop(columns=['Price (Euro)'] + categorical_columns + ['Product'])
y = df['Price (Euro)']

# Split the dataset into training and testing sets (70% training, 30% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Display the shapes of the resulting splits
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

```
((892, 13), (383, 13), (892,), (383,))
```

```
# Initialize and train the Random Forest Regressor
model = RandomForestRegressor(random_state=42)
model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = model.predict(X_test)

# Evaluate model performance
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

mae, mse, r2
```

```
(184.92043200360558, 78463.15801777369, 0.8477041093586216)
```

```
categorical_columns = ['Company', 'TypeName', 'ScreenResolution', 'CPU_Company', 'CPU_Type',
                       'Memory', 'GPU_Company', 'GPU_Type', 'OpSys']
label_encoders = {}
for column in categorical_columns:
    le = LabelEncoder()
    df[column + '_encoded'] = le.fit_transform(df[column])
    label_encoders[column] = le

# Define features (X) and label (y)
X = df[[col + '_encoded' for col in categorical_columns]]
y = df['Price (Euro)']

# Split the dataset into training and testing sets (70% training, 30% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Initialize and train the Random Forest Regressor
model = RandomForestRegressor(random_state=42)
```

```
model.fit(X_train, y_train)
```

```
RandomForestRegressor
RandomForestRegressor(random_state=42)
```

```
# Make predictions on the test set
y_pred = model.predict(X_test)


# Evaluate model performance
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"Mean Absolute Error: {mae}")
print(f"Mean Squared Error: {mse}")
print(f"R2 Score: {r2}")
```

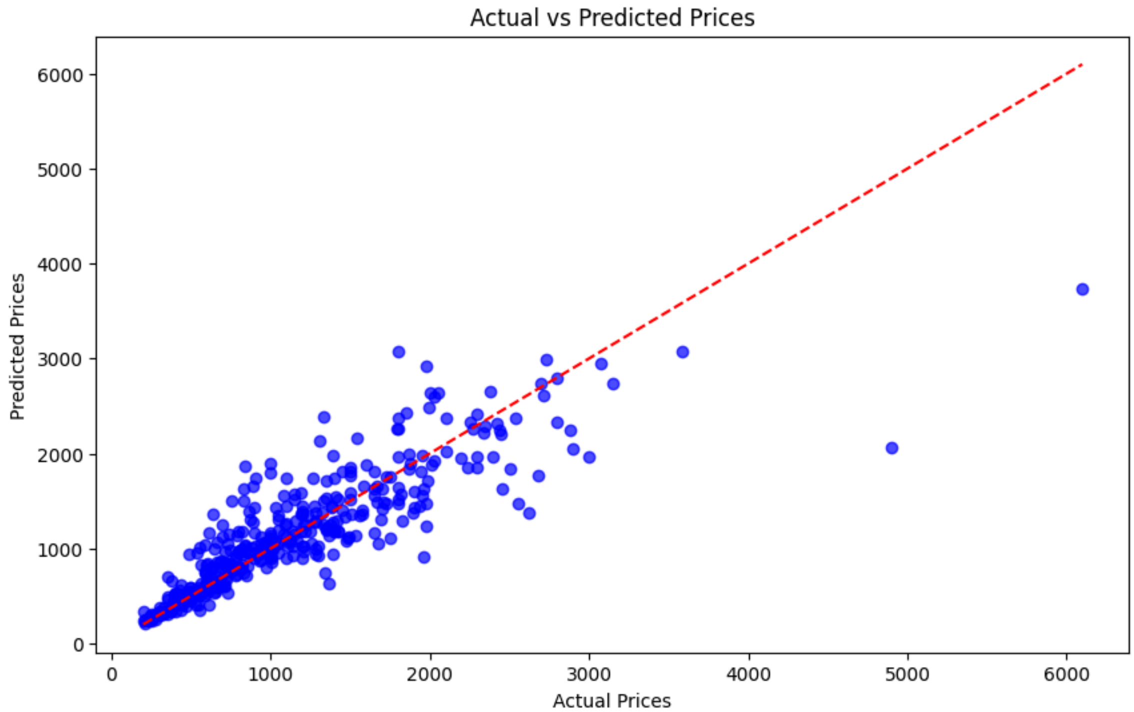
```
Mean Absolute Error: 225.42978511106634
Mean Squared Error: 136259.52081973566
R2 Score: 0.7355221787414121
```

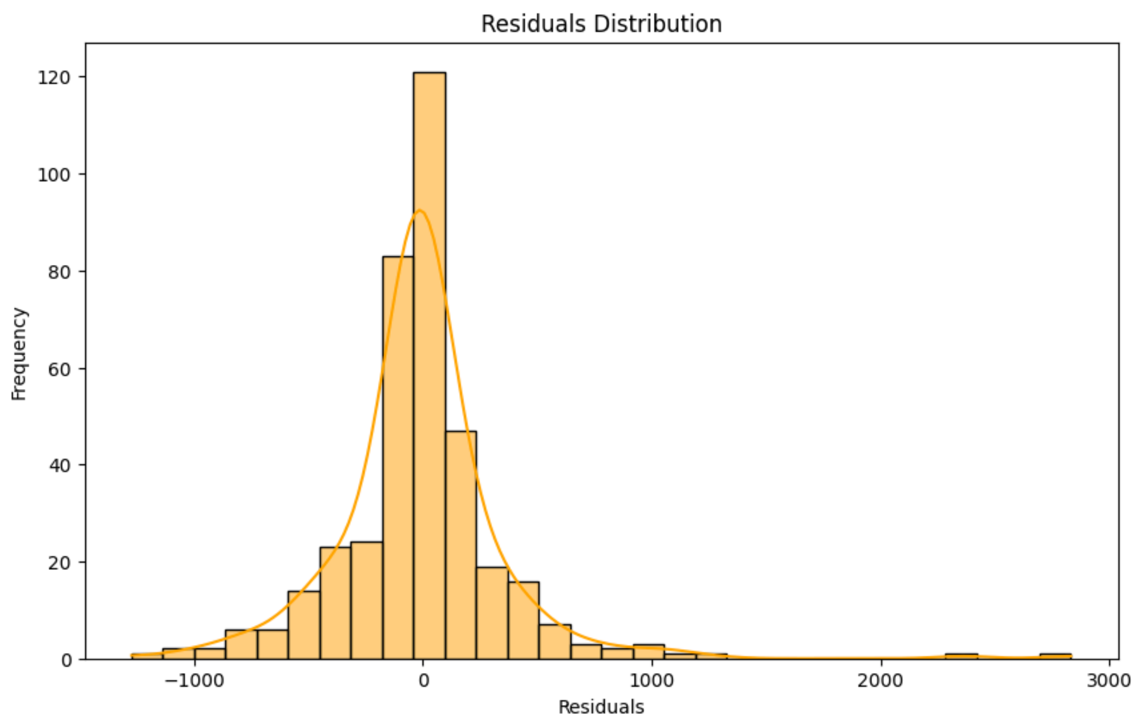
```
# Test the model with new data (example)
new_data = {
    "Company": ['Apple', 'HP'],
    "TypeName": ['Ultrabook', 'Notebook'],
    "ScreenResolution": ['IPS Panel Retina Display 2560x1600', 'Full HD 1920x1080'],
    "CPU_Company": ['Intel', 'Intel'],
    "CPU_Type": ['Core i5', 'Core i7'],
    "Memory": ['128GB SSD', '512GB SSD'],
    "GPU_Company": ['Intel', 'AMD'],
    "GPU_Type": ['Iris Plus Graphics 640', 'Radeon Pro 455'],
    "OpSys": ['macOS', 'Windows 10']
}

# Create DataFrame for new data and encode it
df_new = pd.DataFrame(new_data)
for column in categorical_columns:
    df_new[column + '_encoded'] = label_encoders[column].transform(df_new[column])

# Predict on new data
X_new = df_new[[col + '_encoded' for col in categorical_columns]]
new_pred = model.predict(X_new)

# Display results for new data predictions
df_new['Predicted_Price'] = new_pred
print(df_new[['Company', 'TypeName', 'Predicted_Price']])
```

```
  Company   TypeName  Predicted_Price
0   Apple  Ultrabook      1343.542600
1      HP   Notebook      1396.593109
```

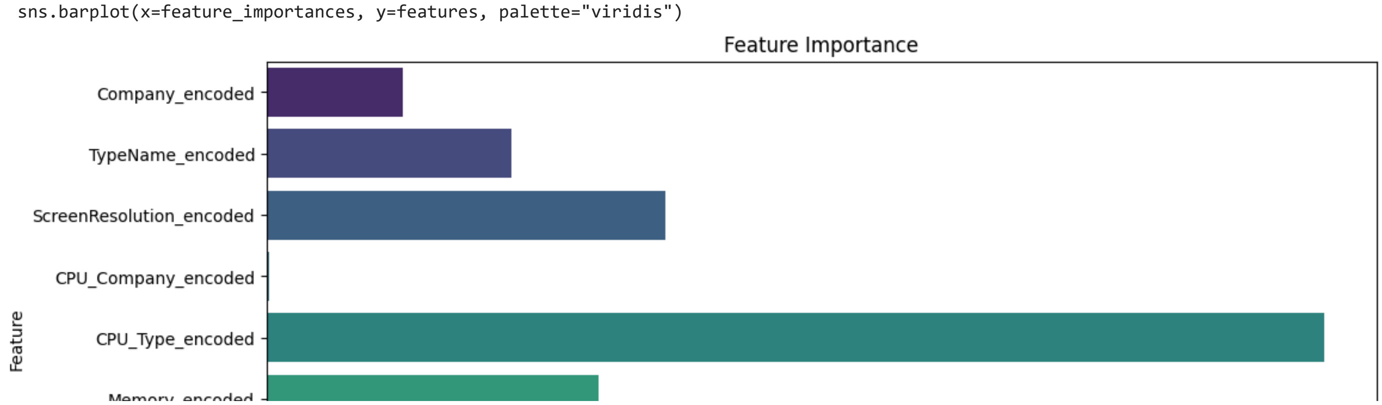
```
# Plot Actual vs Predicted Prices
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_pred, alpha=0.7, color='blue')
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], color='red', linestyle='--')  # Line of perfect prediction
plt.title('Actual vs Predicted Prices')
plt.xlabel('Actual Prices')
plt.ylabel('Predicted Prices')
plt.show()

# Plot Residuals
residuals = y_test - y_pred
plt.figure(figsize=(10, 6))
sns.histplot(residuals, kde=True, color='orange', bins=30)
plt.title('Residuals Distribution')
plt.xlabel('Residuals')
plt.ylabel('Frequency')
plt.show()

# Plot Feature Importance
feature_importances = model.feature_importances_
features = X.columns
```

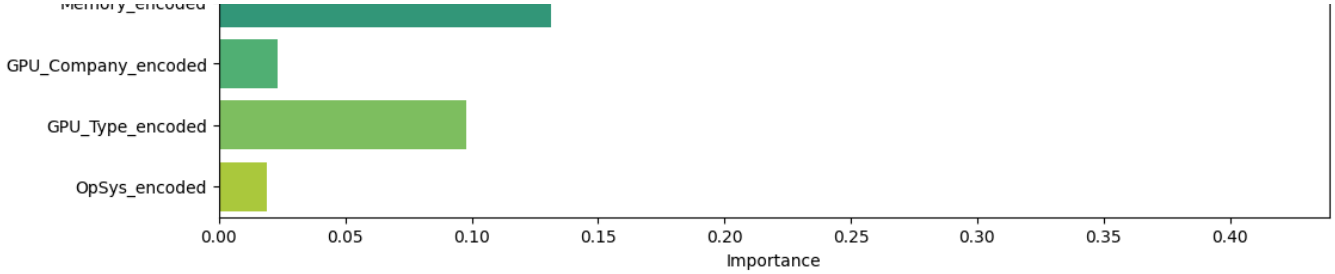
```
plt.figure(figsize=(12, 6))
sns.barplot(x=feature_importances, y=features, palette="viridis")
plt.title('Feature Importance')
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.show()
```






```
<ipython-input-128-affe4881efde>:23: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `y` variable to `hue` and set `legend

  sns.barplot(x=feature_importances, y=features, palette="viridis")
```





Unsupervised Learning

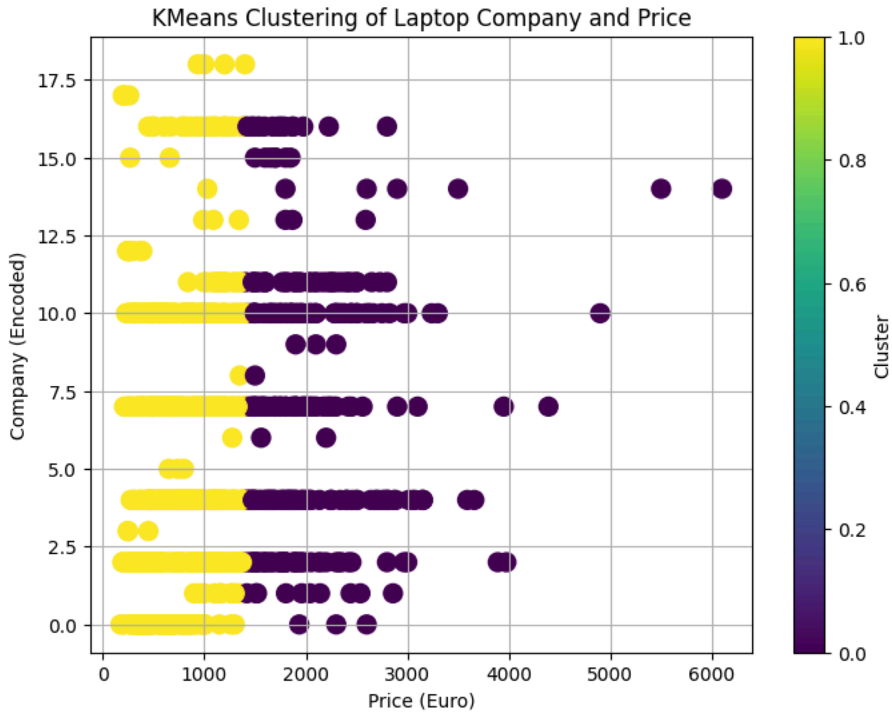
Obeservation of Unsupervised Learning

The code snippet shows how to group data points based on similarities using KMeans clustering, an unsupervised learning technique. The algorithm is used to find groupings or patterns in data that don't have labels. Here, the code snippet divides laptops into two categories according to the encoded "Company" and numeric "Price (Euro)" attributes.

The clusters show various pricing trends connected to various laptop brands. The laptops classified into two clusters by the algorithm are depicted in the color-coded scatter plot. An understanding of the algorithm's interpretation of the relationships within the data is given by the scatter plot.

Without requiring labeled data, the clustering uncovers implicit patterns in the data that were not specifically specified. Without requiring labeled data, the clustering uncovers implicit patterns in the data that were not specifically specified. This is very helpful for identifying hidden structures in unlabeled datasets. To sum up, unsupervised learning methods such as KMeans clustering allow natural categories to be found in unlabeled data.

```
# Select relevant features for clustering
# Using encoded values for 'Company', 'TypeName', and the numeric 'Price (Euro)'
df_cluster = df[['Company_encoded', 'TypeName_encoded', 'Price (Euro)']]


# Perform KMeans clustering with 2 clusters (for simplicity)
kmeans = KMeans(n_clusters=2, random_state=42)
df['Cluster'] = kmeans.fit_predict(df_cluster)


plt.figure(figsize=(8, 6))
plt.scatter(df['Price (Euro)'], df['Company_encoded'], c=df['Cluster'], cmap='viridis', s=100)
plt.title('KMeans Clustering of Laptop Company and Price')
plt.xlabel('Price (Euro)')
plt.ylabel('Company (Encoded)')
plt.colorbar(label='Cluster')
plt.grid(True)
plt.show()
```



Reinforcement Learning

Observation of Reinforcement Learning

A machine learning technique called reinforcement learning (RL) teaches an agent to make decisions by having interactions with its surroundings. The main idea is to maximize cumulative reward by making mistakes and learning from them, using action feedback as a guide. In order to arrive at the best decisions possible, this method expands on the idea of exploration and exploitation.

Finding a balance between exploration and exploitation is one of RL's main challenges. Exploitation is the avaricious selection of activities that have historically produced large benefits, whereas exploration entails attempting new things to find possible profits. For learning to be effective, these two must be balanced.

Both excessive exploration and exploitation can produce less-than-ideal outcomes. Excessive exploration might result in random behavior. An ideal policy is learned by Q-learning, a model-free reinforcement learning algorithm, through Q-table updates. A data structure where each state-action pair's Q-values are kept. By striking a balance between exploration and exploitation, this method ensures effective learning and decision-making and assists agents in making better decisions.

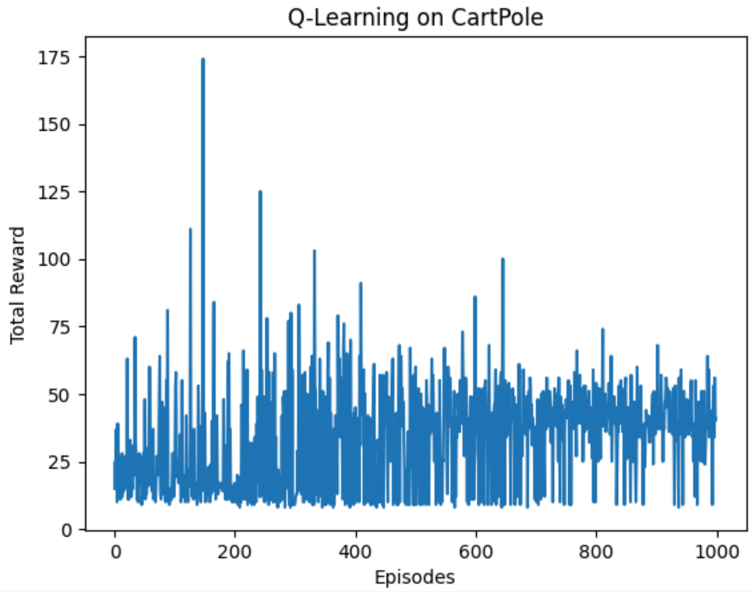
```
# Create the CartPole environment
env = gym.make('CartPole-v1')
```

```
/usr/local/lib/python3.10/dist-packages/gym/core.py:317: DeprecationWarning: WARN: Initializing wrapper in old step API which returns or
  deprecation(
/usr/local/lib/python3.10/dist-packages/gym/wrappers/step_api_compatibility.py:39: DeprecationWarning: WARN: Initializing environment ir
  deprecation(
```

```
# Q-learning settings
learning_rate = 0.1
discount_factor = 0.99
epsilon = 1.0  # Exploration rate
epsilon_decay = 0.995
min_epsilon = 0.01
episodes = 1000
max_steps = 200

# Create the Q-table
state_space = (1, 1, 6, 3)
q_table = np.zeros(state_space + (env.action_space.n,))
```

```
# Function to discretize the continuous state space
def discretize_state(state):
    bins = [
        np.linspace(-4.8, 4.8, state_space[0] - 1),
        np.linspace(-4, 4, state_space[1] - 1),
        np.linspace(-0.418, 0.418, state_space[2] - 1),
        np.linspace(-3.5, 3.5, state_space[3] - 1)
    ]
    return tuple(np.digitize(state[i], bins[i]) for i in range(len(state)))

# Q-learning algorithm
def q_learning():
    global epsilon  # Ensure epsilon can be updated
    rewards = []

    for episode in range(episodes):
        state = discretize_state(env.reset())
        total_reward = 0
        done = False

        for step in range(max_steps):
            if np.random.rand() < epsilon:
                action = env.action_space.sample()  # Explore
            else:
                action = np.argmax(q_table[state])  # Exploit

            next_state, reward, done, _ = env.step(action)
            next_state = discretize_state(next_state)

            total_reward += reward

            # Update Q-value
            q_table[state + (action,)] = q_table[state + (action,)] + learning_rate * (
                reward + discount_factor * np.max(q_table[next_state]) - q_table[state + (action,)]
            )

            state = next_state

            if done:
                break

        rewards.append(total_reward)

        # Decay exploration rate
        epsilon = max(min_epsilon, epsilon * epsilon_decay)

        if (episode + 1) % 100 == 0:
            print(f"Episode {episode + 1}: Total reward: {total_reward}")

    return rewards


# Train the agent
rewards = q_learning()

# Plot the rewards
plt.plot(range(episodes), rewards)
plt.xlabel('Episodes')
plt.ylabel('Total Reward')
plt.title('Q-Learning on CartPole')
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/gym/utils/passive_env_checker.py:241: DeprecationWarning: `np.bool8` is a deprecated alias for `
  if not isinstance(terminated, (bool, np.bool8)):
Episode 100: Total reward: 12.0
Episode 200: Total reward: 17.0
Episode 300: Total reward: 9.0
Episode 400: Total reward: 39.0
Episode 500: Total reward: 27.0
Episode 600: Total reward: 86.0
Episode 700: Total reward: 37.0
Episode 800: Total reward: 10.0
Episode 900: Total reward: 37.0
Episode 1000: Total reward: 41.0
```




```
# Test the trained agent
state = discretize_state(env.reset())
done = False

for step in range(max_steps):
    env.render()  # Render the environment
    action = np.argmax(q_table[state])  # Choose the best action
    next_state, _, done, _ = env.step(action)
    state = discretize_state(next_state)
    if done:
        break

env.close()
```

```
/usr/local/lib/python3.10/dist-packages/gym/core.py:49: DeprecationWarning: WARN: You are calling render method, but you didn't specifie
If you want to render in human mode, initialize the environment in this way: gym.make('EnvName', render_mode='human') and don't call the
See here for more information: https://www.gymlibrary.ml/content/api/
  deprecation(
```

```
# Display the dataframe with the new 'Cluster' column
df[['Company', 'TypeName', 'Price (Euro)', 'Cluster']].head()
```

	Company	TypeName	Price (Euro)	Cluster
0	Apple	Ultrabook	1339.69	1
1	Apple	Ultrabook	898.94	1
2	HP	Notebook	575.00	1
3	Apple	Ultrabook	2537.45	0
4	Apple	Ultrabook	1803.60	0

Semi-Supervised Learning

Obeservation of Semi-Supervised Learning

In this implementation, we use both labeled and unlabeled data to improve the performance of the model. Specifically, we mask 50% of the price labels to simulate a real-world scenario where only a portion of the data is labeled. This is particularly useful when labeling data is expensive or time-consuming. The missing labels are then inferred using the Label Propagation model, which learns from the similarities between features like other price-related factors.

By using this method, the model can predict prices for data points where the price label is missing. It leverages patterns in the available data to "spread" label information across similar items. For example, the model looks at the feature space (which could include factors like size, condition, or location of an item) to infer missing price categories based on the relationships it has learned from the labeled data.

The model's accuracy is evaluated by comparing the predicted prices with the actual labeled prices. Metrics like Mean Absolute Error (MAE) and Mean Squared Error (MSE) give us a clear sense of how close the model's predictions are to the real values. A scatter plot visually compares actual vs. predicted prices, where points closer to the red line indicate better performance.

This semi-supervised learning approach offers a practical solution for cases where obtaining full labels is difficult. By distributing label information across similar data points, the model becomes more robust and can still make accurate predictions with a reduced amount of labeled data. Overall, semi-supervised learning proves to be an efficient method for training models in scenarios with

```
# Mask 50% of the price labels as unlabeled (-1) to simulate semi-supervised learning
n_unlabeled_points = int(0.5 * len(y))  # Mask 50% of the target labels
random_unlabeled_points = np.random.choice(len(y), n_unlabeled_points, replace=False)
y_semi_supervised = y.copy()
y_semi_supervised[random_unlabeled_points] = -1  # Assign -1 to simulate unlabeled data


# Check the labeled and unlabeled data
df['Price_Label'] = y_semi_supervised
print(df[['Price (Euro)', 'Price_Label']].head(10))
```

```
   Price (Euro)  Price_Label
0       1339.69      1339.69
1        898.94       898.94
2        575.00        -1.00
3       2537.45      2537.45
4       1803.60        -1.00
5        400.00        -1.00
6       2139.97      2139.97
7       1158.70        -1.00
8       1495.00      1495.00
9        770.00       770.00
```

```
# Assuming you want to create 5 price categories:
n_bins = 5
discretizer = KBinsDiscretizer(n_bins=n_bins, encode='ordinal', strategy='uniform')
y_semi_supervised_discretized = discretizer.fit_transform(y_semi_supervised.values.reshape(-1, 1))


# Now use the discretized target for Label Propagation:
lp_model = LabelPropagation()
lp_model.fit(X_semi_supervised, y_semi_supervised_discretized.ravel())
```

```
▾ LabelPropagation
LabelPropagation()
```

```
if 'Price_Label' not in df.columns:
    # Assuming y_semi_supervised is still available in the environment
    df['Price_Label'] = y_semi_supervised

y_pred_lp = lp_model.predict(X_semi_supervised)
df['Predicted_Price'] = y_pred_lp
print(df[['Price (Euro)', 'Price_Label', 'Predicted_Price']].head(10))
```

```
   Price (Euro)  Price_Label  Predicted_Price
0       1339.69      1339.69              1.0
1        898.94       898.94              1.0
2        575.00        -1.00              0.0
3       2537.45      2537.45              3.0
4       1803.60        -1.00              0.0
5        400.00        -1.00              0.0
6       2139.97      2139.97              2.0
```

```
7      1158.70       -1.00             0.0
8      1495.00     1495.00             1.0
9       770.00      770.00             0.0
```

```
# Evaluate the model on the originally labeled data
labeled_mask = y_semi_supervised != -1  # Only consider originally labeled data
mae_lp = mean_absolute_error(y[labeled_mask], y_pred_lp[labeled_mask])
mse_lp = mean_squared_error(y[labeled_mask], y_pred_lp[labeled_mask])
r2_lp = r2_score(y[labeled_mask], y_pred_lp[labeled_mask])


print(f"Mean Absolute Error (Label Propagation): {mae_lp}")
print(f"Mean Squared Error (Label Propagation): {mse_lp}")
print(f"R2 Score (Label Propagation): {r2_lp}")
```

```
Mean Absolute Error (Label Propagation): 1099.2775705329154
Mean Squared Error (Label Propagation): 1618596.825380094
R2 Score (Label Propagation): -2.9371169872684417
```

```
plt.figure(figsize=(10, 6))
plt.scatter(y[labeled_mask], y_pred_lp[labeled_mask], alpha=0.7, color='green')
plt.plot([min(y), max(y)], [min(y), max(y)], color='red', linestyle='--')  # Line of perfect prediction
plt.title('Actual vs Predicted Prices (Label Propagation)')
plt.xlabel('Actual Prices')
plt.ylabel('Predicted Prices')
plt.show()
```