```
#!/usr/bin/env python
# coding: utf-8

# # Imports and Instantiation

# In[ ]:


import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import re
from rich import print
from rich.table import Table
from rich.console import Console
import seaborn as sns

console = Console()


# # Loading the Data
#
#

# In[ ]:


# Defining the file paths of the train dataset
train_set = "data/laptop_data_train.csv"
test_set = 'data/laptop_data_test.csv'

# Reading the Datasets into their respective Pandas Dataframes
train_data = pd.read_csv(train_set, header=0)
test_data = pd.read_csv(test_set, header=0)

# Dropping the 'Number' Column as it is surplus to requirements
train_data = train_data.drop('Number', axis=1)
test_data = test_data.drop('Number', axis=1)

# Storing all the features in a list of attributes
attributes = train_data.columns.tolist()


# # Feature Engineering
#
# Clean data - eliminate redundant values. For numerical data, replace missing values with NaN. For categorical dat
#
# ## The Data

# In[ ]:


print('Unique Values recorded in each feature (before processing)')

# Printing all unique values of each feature in the dataset to understand the data better
for attribute in attributes[:-1]:
    att_table = Table()
    att_table.add_column(f'{attribute}: Unique Count = {len(set(train_data[attribute]))}; Data type = {type(train_d

    for element in set(train_data[attribute]):
        att_table.add_row(str(element))

    console.print(att_table)


# In[ ]:


# Function to drop the 'GB' suffix in the RAM data
def transform_ram(ram_string):
    return int(ram_string.replace('GB', ''))

# Function to drop the 'kg' suffix in the Weight data
def transform_weight(weight_string):
    return float(weight_string.replace('kg', ''))
```

```
# Function to drop the 'GHz' suffix in the ClockSpeed data
def transform_clockspeed(speed_string):
    return float(speed_string.replace('GHz', ''))


# ## Checking for Extra/Unnecessary Spaces in Columns
# Specifically, Cpu and Gpu, so we can remove extra spaces so they are not counted as new features

# In[ ]:


'''

Define a Base String for each unique value in a column. The base string contains no spaces and is entirely in lower
case. This allows thorough comparison between values to check for uniqueness. If two or more values have the same
base string, they represent the same entity but have different casing or spacing in the dataset. Therefore all
values are then changed to have uniform casing and spacing to have cleaner data.

'''

print(f'Number of unique values of the GPU feature before cleaning: {len(set(train_data["Gpu"]))}')

# Create an empty dictionary to store all feature values with their base strings
str_compare = {}
for element in set(train_data['Gpu']):
    # Convert the current element to its base string
    base_str = element.replace(' ', '').lower()


    # If the base string exists, replace the element with the key of the base string in the dictionary
    if base_str in str_compare.values():
        inverted_dict = {value: key for key, value in str_compare.items()}
        key = inverted_dict[base_str]
        train_data.loc[train_data['Gpu'] == element, 'Gpu'] = key

    # Add the element-base string pair to the dictionary if the base string doesn't already exist
    else:
        str_compare[element] = base_str

print(f'Number of unique values of the GPU feature after cleaning: {len(set(train_data["Gpu"]))}')




print(f'Number of unique values of the CPU feature before cleaning: {len(set(train_data["Cpu"]))}')

str_compare = {}
for element in set(train_data['Cpu']):
    base_str = element.replace(' ', '').lower()

    if base_str in str_compare.values():
        inverted_dict = {value: key for key, value in str_compare.items()}
        key = inverted_dict[base_str]
        train_data.loc[train_data['Cpu'] == element, 'Cpu'] = key

    else:
        str_compare[element] = base_str

print(f'Number of unique values of the CPU Model feature after cleaning: {len(str_compare.values())}')


# In[ ]:


# Function for feature pre-processing
def pre_process(data_set):
    # Define a regular expression pattern to match the desired format
    pattern = r'(.*)\b(\d+)x(\d+)\b'

    # Apply the regular expression pattern to the 'ScreenResolution' column and extract the two terms
    new_columns = data_set['ScreenResolution'].str.extract(pattern)

    # Rename the columns for clarity
    new_columns.columns = ['Display', 'Width', 'Height']

    # Get the index of the 'ScreenResolution' column
    screen_resolution_column_index = data_set.columns.get_loc('ScreenResolution')
```

```
# Drop the original 'ScreenResolution' column
data_set.drop(columns=['ScreenResolution'], inplace=True)

# Insert the new columns at the index where the 'ScreenResolution' column was
for col in reversed(new_columns.columns):
    data_set.insert(screen_resolution_column_index, col, new_columns[col])

# Get the index of the 'Width' column
width_column_index = data_set.columns.get_loc('Width')

new_column = data_set['Width'].astype(int) * data_set['Height'].astype(int)
# new_column.columns = ['ScreenArea']

# Drop the original 'Width' and 'Height' columns
data_set.drop(columns=['Width', 'Height'], inplace=True)

# Insert the new 'ScreenArea' column
data_set.insert(width_column_index, 'ScreenArea', new_column)

# Check if each row contains "Touchscreen" and assign 1 or 0 accordingly
data_set['Touchscreen'] = data_set['Display'].str.contains('Touchscreen').astype(int)

# Remove "Touchscreen", " / Touchscreen", and "Touchscreen / " from the original sentence
data_set['Display'] = data_set['Display'].str.replace(r'\s*(?:/)?\s*Touchscreen\s*(?:/)?\s*', '', regex=True)

# Get the index of the Display column
display_index = data_set.columns.get_loc('Display')

# Insert the Touchscreen column right after the Display column
data_set.insert(display_index + 1, 'Touchscreen', data_set.pop('Touchscreen'))

# Trim all the spaces padding the values of the 'Display' column
data_set['Display'] = data_set['Display'].str.strip()

# Fill missing values in the 'Display' column with 'None'
data_set['Display'] = data_set['Display'].replace('', 'None')

new_columns = data_set['Cpu'].str.rsplit(n=1, expand=True)

# Rename the columns for clarity
new_columns.columns = ['CpuModel', 'ClockSpeed']

# Get the index of the 'Cpu' column
cpu_column_index = data_set.columns.get_loc('Cpu')

# Drop the original 'Cpu' column
data_set.drop(columns=['Cpu'], inplace=True)

# Insert the new columns at the index where the 'Cpu' column was
for col in reversed(new_columns.columns):
    data_set.insert(cpu_column_index, col, new_columns[col])

# Convert Clock Speed's data type to Float
data_set['ClockSpeed'] = data_set['ClockSpeed'].apply(transform_clockspeed)
###


# Applying the custom transformations on RAM and Weight data
# Dropping the unit suffixes and converting the data type to a number
data_set['Ram'] = data_set['Ram'].apply(transform_ram)
data_set['Weight'] = data_set['Weight'].apply(transform_weight)

# Splitting the Memory feature into two Memory Components
new_columns = data_set['Memory'].str.split(' + ', expand=True)
new_columns.columns = ['MemoryComponent1', 'MemoryComponent2']

# Saving the index of the Memory feature and dropping the column
memory_column_index = data_set.columns.get_loc('Memory')
data_set.drop(columns=['Memory'], inplace=True)

# Inserting the two new Memory features where the original Memory column was located
for col in reversed(new_columns.columns):
    data_set.insert(memory_column_index, col, new_columns[col])

# Cleaning all the string values in the new features by removing '+' and all padding spaces
data_set['MemoryComponent1'] = data_set['MemoryComponent1'].str.replace('+', '', regex=False).str.strip()
```

```
    data_set['MemoryComponent2'] = data_set['MemoryComponent2'].str.strip()

    # Strip extra spaces from column 'Gpu'
    data_set['Gpu'] = data_set['Gpu'].str.replace(r'\s+', ' ').str.strip()

    # Strip extra spaces from column 'Cpu'
    data_set['CpuModel'] = data_set['CpuModel'].str.replace(r'\s+', ' ').str.strip()

    return data_set

# Call pre-processing function
eda_train_data = train_data.copy()
eda_train_data = pre_process(eda_train_data)

eda_test_data = test_data.copy()
eda_test_data = pre_process(eda_test_data)

# print(eda_train_data.groupby('MemoryComponent1')['Price'].mean().sort_values())


# ## Exploratory Data Analysis - Categorical Data
# Plots and Visualization of each feature to highlight feature characteristics and explain your feature representat

# In[ ]:


categorical_features = ['Company', 'TypeName', 'Display', 'MemoryComponent1', 'MemoryComponent2', 'OpSys']

# Plotting bar graphs for each categorical feature
plt.figure(figsize=(12, 6*len(categorical_features)))
plt_ctr = 1

# Plotting Bar and/or Seaborn Bar Plots for each Categorical Feature to visually infer characteristics of the data
for feature in categorical_features:
    if feature not in ['Company', 'TypeName', 'OpSys']:
        plt.subplot(len(categorical_features), 2, plt_ctr)
        eda_train_data[feature].value_counts().plot(kind='bar', color='teal')
        plt.title(f'Bar Plot for {feature}', fontsize=16)
        plt.xlabel(feature, fontsize=14)
        plt.ylabel('Frequency', fontsize=14)
        plt.xticks(rotation='vertical')
        plt.grid(axis='y', linestyle='--', alpha=0.7)
        plt_ctr += 1

    plt.subplot(len(categorical_features), 2, plt_ctr)
    sns.barplot(x = eda_train_data[feature], y = eda_train_data['Price'], hue=eda_train_data[feature], palette='Set
    plt.title(f'Price vs. {feature}', fontsize=16)
    plt.xticks(rotation='vertical')

    plt_ctr += 1
plt.show()

# Plot for the GPU feature. Values vs. Price Comparison
plt.figure(figsize=(20, 6))
sns.barplot(x = eda_train_data['Gpu'], y = eda_train_data['Price'], hue=eda_train_data['Gpu'], palette='Set2')
plt.title('Price vs. GPU', fontsize=16)
plt.xticks(rotation='vertical')
plt.show()


# ## Data Classes After Feature Pre-Processing

# In[ ]:


# Storing all the features in a list of attributes
attributes = eda_train_data.columns.tolist()

print('Unique Values recorded in each feature (after processing)')

# Printing all unique values of each feature in the dataset to understand the data better
for attribute in attributes[:-1]:
    att_table = Table()
    att_table.add_column(f'{attribute}: Unique Count = {len(set(eda_train_data[attribute]))}; Data type = {type(eda

    for element in set(eda_train_data[attribute]):
        att_table.add_row(str(element))
```

```
    console.print(att_table)


# In[ ]:


def compare_features(data_set, eda_data):

    # Storing all the features in a list of attributes
    attributes2 = data_set.columns.tolist()

    print('Comparison of Before and After Pre-Processing')

    att_table = Table()
    att_table.add_column(f'Features Before: (Total = {len(attributes2)})')
    att_table.add_column(f'Features After: (Total = {len(attributes)})')

    # Find the minimum length between attributes and attributes2
    min_length = min(len(attributes), len(attributes2))


    # Add rows for attributes that are present in both lists
    for i in range(min_length):
        att_table.add_row(
            f'{attributes2[i]}: Unique Count = {len(set(data_set[attributes2[i]]))}',
            f'{attributes[i]}: Unique Count = {len(set(eda_data[attributes[i]]))}'
        )

    # Add remaining attributes if it's longer
    if len(attributes) > min_length:
        for i in range(min_length, len(attributes)):
            att_table.add_row(
                '', f'{attributes[i]}: {len(set(eda_data[attributes[i]]))}'
            )

        # att_table.add_row(f'{attribute2}: Unique Count = {len(set(data_set[attribute2]))}',
        # f'{attribute}: Unique Count = {len(set(eda_data[attribute]))}')

    console.print(att_table)

compare_features(train_data, eda_train_data)
# compare_features(test_data, eda_test_data)


# ## Final Table Visualization After Feature Pre-Processing
# After this, use the EDA information to label encode and one-hot-encode otherwise

# In[ ]:


# Display the Dataframe after initial data processing

styled_df = eda_train_data.style

# Apply various formatting options
styled_df = styled_df.set_properties(**{'text-align': 'center'})  # Align text to center
styled_df = styled_df.set_table_styles([{'selector': 'th', 'props': [('font-size', '11pt')]}])  # Set font size for

# Display the styled DataFrame

styled_df


# ## Export EDA Dataset
# Export EDA dataset to csv after visualizations and feature splitting complete

# In[ ]:


eda_train_set = 'data/eda_train_data.csv'
eda_train_data.to_csv(eda_train_set, index_label='Number')

eda_test_set = 'data/eda_test_data.csv'
eda_test_data.to_csv(eda_test_set, index_label='Number')
```

```
#

# <a style='text-decoration:none;line-height:16px;display:flex;color:#5B5B62;padding:10px;justify-content:end;' hre
# <img alt='Created in deepnote.com' style='display:inline;max-height:16px;margin:0px;margin-right:7.5px;' src='dat
# Created in <span style='font-weight:600;margin-left:4px;'>Deepnote</span></a>
```

```
#!/usr/bin/env python
# coding: utf-8

# # Imports and Instantiation

# In[ ]:


import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold

from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score

from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor

import matplotlib.pyplot as plt
import re
from rich import print
from rich.table import Table
from rich.console import Console
import seaborn as sns

console = Console()


# # Loading the Data
#
#

# In[ ]:


# Defining the file paths of the train and test datasets
train_set = "data/eda_train_data.csv"
test_set = "data/eda_test_data.csv"

# Reading the Datasets into their respective Pandas Dataframes
train_data = pd.read_csv(train_set, header=0)
test_data = pd.read_csv(test_set, header=0)

# Dropping the 'Number' Column as it is surplus to requirements
train_data = train_data.drop('Number', axis=1)
test_data = test_data.drop('Number', axis=1)

# Storing all the features in a list of attributes
attributes = train_data.columns.tolist()


# ## Comparing test set features to training for encoding

# In[ ]:


def compare_unique_vals(train_data, test_data, col):
    # Get unique values from train and test
    unique_values_train = set(train_data[f'{col}'])
    unique_values_test = set(test_data[f'{col}'])

    # Find unique values in train that are not in test
    unique_in_train_not_in_test = unique_values_train - unique_values_test

    # Find unique values in test that are not in train
    unique_in_test_not_in_train = unique_values_test - unique_values_train

    # Find unique values common to both train and test
    common_unique_values = unique_values_train.intersection(unique_values_test)

    unique_in_train_not_in_test = unique_values_train - unique_values_test
    unique_in_test_not_in_train = unique_values_test - unique_values_train
    common_unique_values = unique_values_train.intersection(unique_values_test)

    print(f"For {col}: Unique values in train but not in test: {unique_in_train_not_in_test}\nTotal = {len(unique_i
```

```
    print(f"For {col} — Unique values in test but not in train: {unique_in_test_not_in_train}\nTotal = {len(unique_
    print(f"For {col}: Common unique values: {common_unique_values}\nTotal = {len(common_unique_values)}")

    return unique_in_test_not_in_train

# For label encoding mapping
unseen_val_display = compare_unique_vals(train_data,test_data, 'Display')
unseen_val_mc1 = compare_unique_vals(train_data,test_data, 'MemoryComponent1')
unseen_val_mc2 = compare_unique_vals(train_data,test_data, 'MemoryComponent2')
unseen_val_cpu = compare_unique_vals(train_data,test_data, 'CpuModel')
unseen_val_gpu = compare_unique_vals(train_data,test_data, 'Gpu')

# print(train_data['Display'].unique())
# print(test_data['Display'].unique())


# ## Categorical Feature Encoding
# Applying necessary encoding to all categorical data

# In[ ]:


def one_hot_test(test_data, train_data, col, col_idx):
    # Step 1: Extract unique features from the train dataset
    unique_features = train_data[col].unique()

    one_hot = {}
    for val in unique_features:
        one_hot[f"one_hot_{col}_{val}"] = (test_data[col] == val).astype(int)

    # Convert one_hot dictionary to DataFrame
    one_hot = pd.DataFrame(one_hot)

    test_data.drop(columns=[f'{col}'], inplace=True)
    test_data = pd.concat([test_data.iloc[:, :col_idx], one_hot, test_data.iloc[:, col_idx:]], axis=1)

    return test_data

def encoding(train_data, test_data):
    # Perform one-hot encoding on the 'Company' feature
    one_hot_encoded_company = pd.get_dummies(train_data['Company'], prefix='one_hot_Company').astype(int)

    # Store the column index and drop the 'Company' feature
    company_column_index = train_data.columns.get_loc('Company')
    test_data = one_hot_test(test_data, train_data, 'Company', company_column_index) # Encode for test
    train_data.drop(columns=['Company'], inplace=True)

    # Concatenate the one-hot encoded Column features with the original DataFrame at the specified index
    train_data = pd.concat([train_data.iloc[:, :company_column_index], one_hot_encoded_company, train_data.iloc[:,

    # Perform one-hot encoding on the 'TypeName' feature
    one_hot_encoded_type = pd.get_dummies(train_data['TypeName'], prefix='one_hot_TypeName').astype(int)

    # Store the column index and drop the 'TypeName' feature
    type_column_index = train_data.columns.get_loc('TypeName')
    test_data = one_hot_test(test_data, train_data, 'TypeName', type_column_index) # Encode for test
    train_data.drop(columns=['TypeName'], inplace=True)

    # Concatenate the one-hot encoded TypeName features with the original DataFrame at the specified index
    train_data = pd.concat([train_data.iloc[:, :type_column_index], one_hot_encoded_type, train_data.iloc[:, type_c

    # Mapping the values of the Display to their ordered integer ranks based on their average prices

    Display_mapping = {}

    for value in unseen_val_display:
        Display_mapping[value] = -1

    # Fill nans to None
    train_data['Display'] = train_data['Display'].fillna('None')
    test_data['Display'] = test_data['Display'].fillna('None')

    # Computing the average price of a laptop when grouped by the unique values of Display
    average_prices = train_data.groupby('Display')['Price'].mean().sort_values()
    # Ranking the Display in ascending order of prices
    ordered_displays = average_prices.index.tolist()
```

```
# Assign unique integer values to each string
for i, string in enumerate(ordered_displays, start=1):
    Display_mapping[string] = i

# Apply custom label encoding using map function
train_data['Display'] = train_data['Display'].map(Display_mapping)
test_data['Display'] = test_data['Display'].map(Display_mapping)

# Mapping the values of the Memory Components to their ordered integer ranks based on their average prices

mc1_mapping = {}

for value in unseen_val_mc1:
    mc1_mapping[value] = -1

#Computing the average price of a laptop when grouped by the unique values of Memory Component 1
average_prices = train_data.groupby('MemoryComponent1')['Price'].mean().sort_values()
# Ranking the memory components in ascending order of prices
ordered_mc1 = average_prices.index.tolist()

# Manually mapping None to 0 to fill all missing values for laptops don't have secondary memory
mc2_mapping = {'None': 0}

for value in unseen_val_mc2:
    mc2_mapping[value] = -1

#Computing the average price of a laptop when grouped by the unique values of Memory Component 2
average_prices = train_data.groupby('MemoryComponent2')['Price'].mean().sort_values()
# Ranking the memory components in ascending order of prices
ordered_mc2 = average_prices.index.tolist()

# Assign unique integer values to each string
for i, string in enumerate(ordered_mc1, start=1):
    mc1_mapping[string] = i

# Assign unique integer values to each string
for i, string in enumerate(ordered_mc2, start=1):
    mc2_mapping[string] = i

# Apply custom label encoding using map function
train_data['MemoryComponent1'] = train_data['MemoryComponent1'].map(mc1_mapping)
train_data['MemoryComponent2'] = train_data['MemoryComponent2'].map(mc2_mapping)
test_data['MemoryComponent1'] = test_data['MemoryComponent1'].map(mc1_mapping)
test_data['MemoryComponent2'] = test_data['MemoryComponent2'].map(mc2_mapping)

train_data['MemoryComponent2'] = train_data['MemoryComponent2'].fillna(mc2_mapping['None'])
train_data['MemoryComponent2'] = train_data['MemoryComponent2'].astype(int)
test_data['MemoryComponent2'] = test_data['MemoryComponent2'].fillna(mc2_mapping['None'])
test_data['MemoryComponent2'] = test_data['MemoryComponent2'].astype(int)

# Perform one-hot encoding on the 'OpSys' feature
one_hot_encoded_os = pd.get_dummies(train_data['OpSys'], prefix='one_hot_OpSys').astype(int)

# Store the column index and drop the 'OpSys' feature
os_column_index = train_data.columns.get_loc('OpSys')
test_data = one_hot_test(test_data, train_data, 'OpSys', type_column_index) # Encode for test
train_data.drop(columns=['OpSys'], inplace=True)

# Concatenate the one-hot encoded OpSys features with the original DataFrame at the specified index
train_data = pd.concat([train_data.iloc[:, :os_column_index], one_hot_encoded_os, train_data.iloc[:, os_column_

# Mapping the values of the CPUs to their ordered integer ranks based on their average prices
Cpu_mapping = {}

for value in unseen_val_cpu:
    Cpu_mapping[value] = -1

#Computing the average price of a laptop when grouped by the unique values of CPU
average_prices = train_data.groupby('CpuModel')['Price'].mean().sort_values()
# Ranking the CPUs in ascending order of prices
ordered_cpus = average_prices.index.tolist()

# Assign unique integer values to each string
for i, string in enumerate(ordered_cpus, start=1):
    Cpu_mapping[string] = i

# Apply custom label encoding using map function
```

```
    train_data['CpuModel'] = train_data['CpuModel'].map(Cpu_mapping)
    test_data['CpuModel'] = test_data['CpuModel'].map(Cpu_mapping)

    # Mapping the values of the GPUs to their ordered integer ranks based on their average prices
    Gpu_mapping = {}

    for value in unseen_val_gpu:
        Gpu_mapping[value] = -1

    #Computing the average price of a laptop when grouped by the unique values of GPU
    average_prices = train_data.groupby('Gpu')['Price'].mean().sort_values()
    # Ranking the GPUs in ascending order of prices
    ordered_gpus = average_prices.index.tolist()

    # Assign unique integer values to each string
    for i, string in enumerate(ordered_gpus, start=1):
        Gpu_mapping[string] = i

    # Apply custom label encoding using map function
    train_data['Gpu'] = train_data['Gpu'].map(Gpu_mapping)
    test_data['Gpu'] = test_data['Gpu'].map(Gpu_mapping)

    # # Initialize StandardScaler
    # scaler = StandardScaler()

    # # Fit the scaler to the data and transform the desired feature
    # # For example, let's standardize 'Feature1'
    # train_data['Ram'] = scaler.fit_transform(train_data[['Ram']])
    # train_data['Weight'] = scaler.fit_transform(train_data[['Weight']])
    # train_data['MemoryComponent1'] = scaler.fit_transform(train_data[['MemoryComponent1']])
    # train_data['MemoryComponent2'] = scaler.fit_transform(train_data[['MemoryComponent2']])

    return train_data, test_data

# # Get unique features for each column
# unique_features = {}
# for col in train_data.columns:
#     unique_features[col] = train_data[col].unique()

# print(unique_features)



# ### Encoding Training and Test Data

# In[ ]:


# Encoding training data
train_data, test_data = encoding(train_data, test_data)


# In[ ]:


# # # Print unique training attributes
# attributes = train_data.columns.tolist()

# for i, attribute in enumerate(attributes):
#     if attribute != 'Number' and attribute != 'Weight' and attribute != 'Price':
#         print(f'Feature {i}, x{i}: {attribute} - *(Total: {len(set(train_data[attribute]))})*\n {set(train_data[a

# # Print unique testing attributes
attributes = test_data.columns.tolist()

for i, attribute in enumerate(attributes):
    if attribute == 'Display' or attribute == 'MemoryComponent1'or attribute == 'MemoryComponent2'or attribute == '
        print(f'Feature {i}, x{i}: {attribute} - *(Total: {len(set(test_data[attribute]))})*\n {set(test_data[attri
    # if attribute != 'Number' and attribute != 'Weight' and attribute != 'Price':
    #     print(f'Feature {i}, x{i}: {attribute} - *(Total: {len(set(test_data[attribute]))})*\n {set(test_data[att


# In[ ]:


# styled_df = train_data.style
styled_df = test_data.style
```

```
# Apply various formatting options
styled_df = styled_df.set_properties(**{'text-align': 'center'})  # Align text to center
styled_df = styled_df.set_table_styles([{'selector': 'th', 'props': [('font-size', '11pt')]}])  # Set font size for

# Display the styled DataFrame

styled_df


# # Export the Datasets

# In[ ]:


final_train_set = 'data/final_train_data.csv'
train_data.to_csv(final_train_set, index_label='Number')

final_test_set = 'data/final_test_data.csv'
test_data.to_csv(final_test_set, index_label='Number')


# In[ ]:


def compare_features(data_set, eda_data, eda2_data):

    # Storing all the features in a list of attributes
    attributes2 = data_set.columns.tolist()
    attributes = eda_data.columns.tolist()
    attributes3 = eda2_data.columns.tolist()

    print('Comparison of Before and After Pre-Processing')

    att_table = Table()
    att_table.add_column(f'Features Before: (Total = {len(attributes2)})')
    att_table.add_column(f'Features After Preprocessing: (Total = {len(attributes)})')
    att_table.add_column(f'Features After Encoding: (Total = {len(attributes3)})')

    # Find the minimum length between attributes and attributes2
    min_length1 = min(len(attributes), len(attributes2))
    min_length2 = min(len(attributes), len(attributes3))
    print(min_length2)

    # Add rows for attributes that are present in both lists
    for i in range(min_length1):
        att_table.add_row(
            f'{attributes2[i]}: {len(set(data_set[attributes2[i]]))}',
            f'{attributes[i]}: {len(set(eda_data[attributes[i]]))}',
            f'{attributes3[i]}: {len(set(eda2_data[attributes3[i]]))}'
        )

    # Add remaining attributes if it's longer
    if len(attributes3) > min_length1:
        for i in range(min_length1, len(attributes)):
            att_table.add_row(
                '', f'{attributes[i]}:{len(set(eda_data[attributes[i]]))}',
                f'{attributes3[i]}: {len(set(eda2_data[attributes3[i]]))}'
            )
    if len(attributes3) > min_length2:
        for i in range(min_length2, len(attributes3)):
            att_table.add_row(
                '', '', f'{attributes3[i]}: {len(set((eda2_data)[attributes3[i]]))}'
            )

        # att_table.add_row(f'{attribute2}: Unique Count = {len(set(data_set[attribute2]))}',
        # f'{attribute}: Unique Count = {len(set(eda_data[attribute]))}')

    console.print(att_table)

eda2_data = pd.read_csv('data/laptop_data_train.csv', header=0)
eda2_data = eda2_data.drop('Number', axis=1)
eda_data = pd.read_csv(train_set, header=0)
eda_data = eda_data.drop('Number', axis=1)
compare_features(eda2_data, eda_data, train_data)
# compare_features(test_data, eda_test_data)
```

```
# ## Run your Baseline Models to get initial accuracy and loss values
# This provides a marker on performance of the model, and allows us to verify whether the feature engineering that
#
# ### Trivial Model
# • A system that always outputs the mean output value y from the training set

# In[ ]:


class TrivialModel:
    def __init__(self, data):
        self.feature_data = data.iloc[:, :-1].copy()
        self.labels = data.iloc[:, -1].copy()
        self.labels = self.labels.round(2)

    def fit(self):
        self.output = self.labels.mean().round(2)

    def predict(self, user_input):
        return self.output

    def RMSE(self, labels, pred):
        mse_value = mean_squared_error(labels, pred)

        # Calculate RMSE
        rmse_value = np.sqrt(mse_value)

        return rmse_value

    def MAE(self, labels, pred):
        mae = mean_absolute_error(labels, pred)

        return mae

    def R2E(self, labels, pred):
        r_squared_value = r2_score(labels, pred).round(2)

        return r_squared_value


def trivial_system(train_data, test_data, split):
    # Shuffle the DataFrame (optional but recommended)
    train_data = train_data.sample(frac=1).reset_index(drop=True)

    # Determine the size of the training set (e.g., 80%)
    train_size = split

    # Split the DataFrame into train and validation sets
    train_df = train_data.iloc[:int(len(train_data) * train_size)]
    val_df = train_data.iloc[int(len(train_data) * train_size):]

    # Optionally, reset the index of the new DataFrames
    train_df.reset_index(drop=True, inplace=True)
    val_df.reset_index(drop=True, inplace=True)

    val_inputs = val_df.iloc[:, :-1].copy()
    val_labels = val_df.iloc[:, -1].copy().round(2)

    test_inputs = test_data.iloc[:, :-1].copy()
    test_labels = test_data.iloc[:, -1].copy().round(2)

    model = TrivialModel(train_df)
    model.fit()

    train_predictions = []
    val_predictions = []
    predictions = []

    for _, data_pt in model.feature_data.iterrows():
        output = model.predict(data_pt)
        train_predictions.append(output)

    console.print(f'Trivial Model\'s Laptop Price Prediction: {output}\n\n')

    console.print('Performance on Training Set: \n')
```

```
    rmse = model.RMSE(model.labels, train_predictions)
    mae = model.MAE(model.labels, train_predictions)
    r2e = model.R2E(model.labels, train_predictions)

    att_table = Table()
    att_table.add_column(f'RMSE', style='blue')
    att_table.add_column(f'MAE', style='green')
    att_table.add_column(f'R-squared Error', style='red')

    att_table.add_row(str(rmse), str(mae), str(r2e))

    console.print(att_table)

    for _, data_pt in val_inputs.iterrows():
        output = model.predict(data_pt)
        val_predictions.append(output)

    console.print('Performance on Validation Set: \n')

    rmse = model.RMSE(val_labels, val_predictions)
    mae = model.MAE(val_labels, val_predictions)
    r2e = model.R2E(val_labels, val_predictions)

    att_table = Table()
    att_table.add_column(f'RMSE', style='blue')
    att_table.add_column(f'MAE', style='green')
    att_table.add_column(f'R-squared Error', style='red')

    att_table.add_row(str(rmse), str(mae), str(r2e))

    console.print(att_table)

    for _, data_pt in test_inputs.iterrows():
        output = model.predict(data_pt)
        predictions.append(output)

    console.print('Performance on Testing Set: \n')

    rmse = model.RMSE(test_labels, predictions)
    mae = model.MAE(test_labels, predictions)
    r2e = model.R2E(test_labels, predictions)

    att_table = Table()
    att_table.add_column(f'RMSE', style='blue')
    att_table.add_column(f'MAE', style='green')
    att_table.add_column(f'R-squared Error', style='red')

    att_table.add_row(str(rmse), str(mae), str(r2e))

    console.print(att_table)


training_splits = [0.85, 0.8, 0.7, 0.6, 0.5]

for split in training_splits:
    print(f'Training-Validation Split: {split * 100}%')
    trivial_system(train_data, test_data, split)


# ### 1-Nearest Neighbour Model

# In[ ]:


def RMSE(labels, pred):
    mse_value = mean_squared_error(labels, pred)

    # Calculate RMSE
    rmse_value = np.sqrt(mse_value)

    return rmse_value

def MAE(labels, pred):
    mae = mean_absolute_error(labels, pred)

    return mae
```

```
def R2E(labels, pred):
    r_squared_value = r2_score(labels, pred).round(2)

    return r_squared_value

def nearest_neighbour_system(train_data, test_data, split):
    # Shuffle the DataFrame (optional but recommended)
    train_data = train_data.sample(frac=1).reset_index(drop=True)

    # Determine the size of the training set (e.g., 80%)
    train_size = split

    # Split the DataFrame into train and validation sets
    train_df = train_data.iloc[:int(len(train_data) * train_size)]
    val_df = train_data.iloc[int(len(train_data) * train_size):]

    # Optionally, reset the index of the new DataFrames
    train_df.reset_index(drop=True, inplace=True)
    val_df.reset_index(drop=True, inplace=True)

    train_inputs = train_df.iloc[:, :-1].copy()
    train_labels = train_df.iloc[:, -1].copy()

    val_inputs = val_df.iloc[:, :-1].copy()
    val_labels = val_df.iloc[:, -1].copy().round(2)

    test_inputs = test_data.iloc[:, :-1].copy()
    test_labels = test_data.iloc[:, -1].copy().round(2)

    # Initialize the linear regression model
    model = KNeighborsRegressor(n_neighbors=1)
    # Train the model
    model.fit(train_inputs, train_labels)

    train_predictions = model.predict(train_inputs)
    val_predictions = model.predict(val_inputs)
    predictions = model.predict(test_inputs)

    console.print(f'1-Nearest Neighbour Model\'s Laptop Price Prediction Performance:\n\n')

    console.print('Performance on Training Set: \n')

    rmse = RMSE(train_labels, train_predictions)
    mae = MAE(train_labels, train_predictions)
    r2e = R2E(train_labels, train_predictions)

    att_table = Table()
    att_table.add_column(f'RMSE', style='blue')
    att_table.add_column(f'MAE', style='green')
    att_table.add_column(f'R-squared Error', style='red')

    att_table.add_row(str(rmse), str(mae), str(r2e))

    console.print(att_table)

    console.print('Performance on Validation Set: \n')

    rmse = RMSE(val_labels, val_predictions)
    mae = MAE(val_labels, val_predictions)
    r2e = R2E(val_labels, val_predictions)

    att_table = Table()
    att_table.add_column(f'RMSE', style='blue')
    att_table.add_column(f'MAE', style='green')
    att_table.add_column(f'R-squared Error', style='red')

    att_table.add_row(str(rmse), str(mae), str(r2e))

    console.print(att_table)

    console.print('Performance on Testing Set: \n')

    rmse = RMSE(test_labels, predictions)
    mae = MAE(test_labels, predictions)
    r2e = R2E(test_labels, predictions)

    att_table = Table()
```

```
    att_table.add_column(f'RMSE', style='blue')
    att_table.add_column(f'MAE', style='green')
    att_table.add_column(f'R-squared Error', style='red')

    att_table.add_row(str(rmse), str(mae), str(r2e))

    console.print(att_table)


training_splits = [0.85, 0.8, 0.7, 0.6, 0.5]

for split in training_splits:
    print(f'Training-Validation Split: {split * 100}%')
    nearest_neighbour_system(train_data, test_data, split)


# ### Linear Regression Model

# In[ ]:


def linear_regression_system(train_data, test_data):

    # Define the number of folds for cross-validation
    k = 5  # You can choose any value of k

    # Initialize the KFold splitter
    kf = KFold(n_splits=k, shuffle=True, random_state=42)

    # Initialize lists to store evaluation scores
    scores = []

    # Iterate through each fold
    for train_index, val_index in kf.split(train_data):
        # Split data into training and testing sets
        train_df, val_df = train_data.iloc[train_index], train_data.iloc[val_index]

        # Optionally, reset the index of the new DataFrames
        train_df.reset_index(drop=True, inplace=True)
        val_df.reset_index(drop=True, inplace=True)

        train_inputs = train_df.iloc[:, :-1].copy()
        train_labels = train_df.iloc[:, -1].copy()

        val_inputs = val_df.iloc[:, :-1].copy()
        val_labels = val_df.iloc[:, -1].copy().round(2)

        # Initialize the linear regression model
        model = LinearRegression()
        # Train the model
        model.fit(train_inputs, train_labels)

        train_predictions = model.predict(train_inputs)
        val_predictions = model.predict(val_inputs)

        console.print(f'Linear Regression Model\'s Laptop Price Prediction Performance:\n\n')
        console.print('Performance on Training Set: \n')

        rmse = RMSE(train_labels, train_predictions)
        mae = MAE(train_labels, train_predictions)
        r2e = R2E(train_labels, train_predictions)

        att_table = Table()
        att_table.add_column(f'RMSE', style='blue')
        att_table.add_column(f'MAE', style='green')
        att_table.add_column(f'R-squared Error', style='red')
        att_table.add_row(str(rmse), str(mae), str(r2e))
        console.print(att_table)

        console.print('Performance on Validation Set: \n')

        rmse = 0
        mae = 0
        r2e = 0

        rmse = RMSE(val_labels, val_predictions)
        mae = MAE(val_labels, val_predictions)
```

```
        r2e = R2E(val_labels, val_predictions)

        att_table = Table()
        att_table.add_column(f'RMSE', style='blue')
        att_table.add_column(f'MAE', style='green')
        att_table.add_column(f'R-squared Error', style='red')
        att_table.add_row(str(rmse), str(mae), str(r2e))
        console.print(att_table)

        # Evaluate the model
        score = model.score(val_inputs, val_labels)
        scores.append(score)

    # Compute the average score
    average_score = np.mean(scores)
    print("Average score:", average_score)




    # # Shuffle the DataFrame (optional but recommended)
    # train_data = train_data.sample(frac=1).reset_index(drop=True)

    # # Determine the size of the training set (e.g., 80%)
    # train_size = split

    # # Split the DataFrame into train and validation sets
    # train_df = train_data.iloc[:int(len(train_data) * train_size)]
    # val_df = train_data.iloc[int(len(train_data) * train_size):]

    # # Optionally, reset the index of the new DataFrames
    # train_df.reset_index(drop=True, inplace=True)
    # val_df.reset_index(drop=True, inplace=True)



    test_inputs = test_data.iloc[:, :-1].copy()
    test_labels = test_data.iloc[:, -1].copy().round(2)

    predictions = model.predict(test_inputs)

    console.print('Performance on Testing Set: \n')

    rmse = RMSE(test_labels, predictions)
    mae = MAE(test_labels, predictions)
    r2e = R2E(test_labels, predictions)

    att_table = Table()
    att_table.add_column(f'RMSE', style='blue')
    att_table.add_column(f'MAE', style='green')
    att_table.add_column(f'R-squared Error', style='red')
    att_table.add_row(str(rmse), str(mae), str(r2e))
    console.print(att_table)


# training_splits = [0.85, 0.8, 0.7, 0.6, 0.5]

# for split in training_splits:
#     print(f'Training-Validation Split: {split * 100}%')
linear_regression_system(train_data, test_data)


# ### Baseline Models
# • 1NN
# • Linear Regression (no regularization)
#
# ### Loss Functions
# • Root Mean Squared Error (RMSE)
# • Mean Absolute Error (MAE)
# • R-squared (R2)
#
# ## Normalization of features
# Normalize Data (especially numerical features) if that can improve performance
#
# ## Feature Importance and Selection
# List of techniques to test: `Pearson Correlation, Sequential Feature Selection, PCA, ICA, UFS`
```

```
#
# Test the processed dataset to compare the performance of the new model with the initial values. Iterate over diff

# ## Model 1: Ridge Regression

# In[ ]:


# Import necessary libraries
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error

# Instantiate the Ridge regressor
alpha = 1.0  # Regularization strength, you can adjust this value
ridge_model = Ridge(alpha=alpha)

# Train the model
ridge_model.fit(X_train, y_train)

# Predict on the test set
y_pred = ridge_model.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error:", mse)


# ## Model 2: SVR

# In[ ]:


# Import necessary libraries
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error

# Instantiate the SVR model
svr_model = SVR(kernel='linear')  # You can choose different kernels like 'rbf', 'poly', etc.

# Train the model
svr_model.fit(X_train, y_train)

# Predict on the test set
y_pred = svr_model.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error:", mse)


# ## Model 3: RBF Neural Network

# In[ ]:


# Import necessary libraries
import numpy as np
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.kernel_approximation import RBFSampler
from sklearn.linear_model import Ridge

# Create an RBF regression pipeline
rbf_features = RBFSampler(gamma=1, random_state=42)
scaler = StandardScaler()
ridge = Ridge(alpha=1.0)
rbf_regressor = make_pipeline(rbf_features, scaler, ridge)

# Train the RBF regression model
rbf_regressor.fit(X_train, y_train)
```

```
# Predict on the test set
y_pred = rbf_regressor.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error:", mse)


#

# <a style='text-decoration:none;line-height:16px;display:flex;color:#5B5B62;padding:10px;justify-content:end;' hre
# <img alt='Created in deepnote.com' style='display:inline;max-height:16px;margin:0px;margin-right:7.5px;' src='dat
# Created in <span style='font-weight:600;margin-left:4px;'>Deepnote</span></a>
```

```python
#!/usr/bin/env python
# coding: utf-8

# # Imports and Instantiation

# In[ ]:


import numpy as np
import pandas as pd

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold
from sklearn.decomposition import PCA
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.feature_selection import SequentialFeatureSelector

from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score

from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor

import matplotlib.pyplot as plt
import re
from rich import print
from rich.table import Table
from rich.console import Console
import seaborn as sns

console = Console()


# # Loading the Data
#
#

# In[ ]:


# Defining the file paths of the train and test datasets
train_set = "data/final_train_data.csv"
test_set = "data/final_test_data.csv"

# Reading the Datasets into their respective Pandas Dataframes
train_data = pd.read_csv(train_set, header=0)
test_data = pd.read_csv(test_set, header=0)

# Dropping the 'Number' Column as it is surplus to requirements
train_data = train_data.drop('Number', axis=1)
test_data = test_data.drop('Number', axis=1)

# Storing all the features in a list of attributes
attributes = train_data.columns.tolist()

# Reorder the columns in test_data to match the column order of train_data
test_data = test_data[attributes]


# In[ ]:


def standardize_features(data, features_to_standardize):
    # Initialize the StandardScaler
    scaler = StandardScaler()

    # Fit the scaler to the selected features and transform them
    scaled_features = scaler.fit_transform(data[features_to_standardize])

    # Create a copy of the original data to retain non-standardized features
    scaled_data = data.copy()

    # Replace the selected features with the standardized values
    scaled_data.loc[:, features_to_standardize] = scaled_features

    return scaled_data
```

```
features_to_standardize = []
for column_name in train_data.columns:
    if 'one_hot' not in column_name and column_name != 'Price':
        features_to_standardize.append(column_name)

standardized_train_data = standardize_features(train_data, features_to_standardize)
standardized_test_data = standardize_features(test_data, features_to_standardize)


# In[ ]:


# styled_df = train_data.style
styled_df = train_data.style

# Apply various formatting options
styled_df = styled_df.set_properties(**{'text-align': 'center'})  # Align text to center
styled_df = styled_df.set_table_styles([{'selector': 'th', 'props': [('font-size', '11pt')]}])  # Set font size for

# Display the styled DataFrame

styled_df


# In[ ]:


def sequential_feature_selection(train_data, num_features):
    # Initialize the KNN model
    model = KNeighborsRegressor(n_neighbors=1)

    # Perform Sequential Forward Selection (SFS)
    sfs = SequentialFeatureSelector(model, n_features_to_select=num_features, scoring='neg_mean_squared_error', cv=
    sfs.fit(train_data.iloc[:, :-1], train_data.iloc[:, -1])

    # Get selected features
    selected_features = train_data.columns[sfs.get_support(indices=True)]

    # Append the last column of the original dataset to selected features
    selected_features = list(selected_features) + [train_data.columns[-1]]

    # Return the modified dataset with selected features
    return selected_features


# In[ ]:


def ufs_feature_selection(train_data, n_features):
    # Initialize UFS
    selector = SelectKBest(score_func=f_regression, k=n_features)

    # Fit UFS to the training data
    selector.fit(train_data.iloc[:, :-1], train_data.iloc[:, -1])

    # Get selected features based on UFS
    selected_features = train_data.columns[selector.get_support(indices=True)]

    # Append the last column of the original dataset to selected features
    selected_features = list(selected_features) + [train_data.columns[-1]]

    return selected_features


# # Error Calculation

# In[ ]:


def RMSE(labels, pred):
    mse_value = mean_squared_error(labels, pred)

    # Calculate RMSE
    rmse_value = np.sqrt(mse_value).round(4)

    return rmse_value
```

```
def MAE(labels, pred):
    mae = mean_absolute_error(labels, pred).round(4)

    return mae

def R2E(labels, pred):
    r_squared_value = r2_score(labels, pred).round(4)

    return r_squared_value


# ## 1-Nearest Neighbour Model

# In[ ]:


def nearest_neighbour_system(train_data, test_data):

    # Define the number of folds for cross-validation
    k = 5  # You can choose any value of k

    # Initialize lists to store error values for each fold
    train_rmse_list, train_mae_list, train_r2e_list = [], [], []
    val_rmse_list, val_mae_list, val_r2e_list = [], [], []

    # Initialize the KFold splitter
    kf = KFold(n_splits=k, shuffle=True, random_state=42)

    # Iterate through each fold
    for train_index, val_index in kf.split(train_data):
        # Split data into training and testing sets
        train_df, val_df = train_data.iloc[train_index], train_data.iloc[val_index]

        # Optionally, reset the index of the new DataFrames
        train_df.reset_index(drop=True, inplace=True)
        val_df.reset_index(drop=True, inplace=True)

        train_inputs = train_df.iloc[:, :-1].copy()
        train_labels = train_df.iloc[:, -1].copy()

        val_inputs = val_df.iloc[:, :-1].copy()
        val_labels = val_df.iloc[:, -1].copy().round(2)

        # Initialize the linear regression model
        model = KNeighborsRegressor(n_neighbors=1)
        # Train the model
        model.fit(train_inputs, train_labels)

        train_predictions = model.predict(train_inputs)
        val_predictions = model.predict(val_inputs)

        # Calculate error metrics for training set
        train_rmse = np.sqrt(mean_squared_error(train_labels, train_predictions))
        train_mae = mean_absolute_error(train_labels, train_predictions)
        train_r2e = r2_score(train_labels, train_predictions)

        # Calculate error metrics for validation set
        val_rmse = np.sqrt(mean_squared_error(val_labels, val_predictions))
        val_mae = mean_absolute_error(val_labels, val_predictions)
        val_r2e = r2_score(val_labels, val_predictions)

        # Append error values to lists
        train_rmse_list.append(train_rmse)
        train_mae_list.append(train_mae)
        train_r2e_list.append(train_r2e)

        val_rmse_list.append(val_rmse)
        val_mae_list.append(val_mae)
        val_r2e_list.append(val_r2e)

    # Compute mean and standard deviation of error metrics for training and validation sets
    train_rmse_mean = np.mean(train_rmse_list)
    train_rmse_std = np.std(train_rmse_list)
    train_mae_mean = np.mean(train_mae_list)
    train_mae_std = np.std(train_mae_list)
    train_r2e_mean = np.mean(train_r2e_list)
```

```
    train_r2e_std = np.std(train_r2e_list)

    val_rmse_mean = np.mean(val_rmse_list)
    val_rmse_std = np.std(val_rmse_list)
    val_mae_mean = np.mean(val_mae_list)
    val_mae_std = np.std(val_mae_list)
    val_r2e_mean = np.mean(val_r2e_list)
    val_r2e_std = np.std(val_r2e_list)

    # Print mean and std deviation of error metrics in rich tables
    console.print("[blue]Mean and Standard Deviation of Error Metrics over 5 Folds:[/blue]\n")
    console.print("[blue]Training Set:[/blue]")

    att_table = Table(title="Error Metrics", show_header=True, header_style="bold magenta")
    att_table.add_column("Metric", style="cyan", justify="center")
    att_table.add_column("Mean", style="cyan", justify="center")
    att_table.add_column("Std Dev", style="cyan", justify="center")

    att_table.add_row("RMSE", f"{train_rmse_mean:.4f}", f"{train_rmse_std:.4f}")
    att_table.add_row("MAE", f"{train_mae_mean:.4f}", f"{train_mae_std:.4f}")
    att_table.add_row("R-squared Error", f"{train_r2e_mean:.4f}", f"{train_r2e_std:.4f}")

    console.print(att_table)

    console.print("\n[blue]Validation Set:[/blue]")

    att_table = Table(title="Error Metrics", show_header=True, header_style="bold magenta")
    att_table.add_column("Metric", style="cyan", justify="center")
    att_table.add_column("Mean", style="cyan", justify="center")
    att_table.add_column("Std Dev", style="cyan", justify="center")

    att_table.add_row("RMSE", f"{val_rmse_mean:.4f}", f"{val_rmse_std:.4f}")
    att_table.add_row("MAE", f"{val_mae_mean:.4f}", f"{val_mae_std:.4f}")
    att_table.add_row("R-squared Error", f"{val_r2e_mean:.4f}", f"{val_r2e_std:.4f}")

    console.print(att_table)

nearest_neighbour_system(train_data, test_data)


# In[ ]:


nearest_neighbour_system(standardized_train_data, standardized_test_data)


# In[ ]:


selected_features = sequential_feature_selection(standardized_train_data, 40)
train_data_selected = standardized_train_data[selected_features].copy()
test_data_selected = standardized_test_data[selected_features].copy()
nearest_neighbour_system(train_data_selected, test_data_selected)


# In[ ]:


selected_features = ufs_feature_selection(standardized_train_data, 22)
train_data_selected = standardized_train_data[selected_features].copy()
test_data_selected = standardized_test_data[selected_features].copy()
nearest_neighbour_system(train_data_selected, test_data_selected)


# In[ ]:




# <a style='text-decoration:none;line-height:16px;display:flex;color:#5B5B62;padding:10px;justify-content:end;' hre
# <img alt='Created in deepnote.com' style='display:inline;max-height:16px;margin:0px;margin-right:7.5px;' src='dat
# Created in <span style='font-weight:600;margin-left:4px;'>Deepnote</span></a>
```

```python
#!/usr/bin/env python
# coding: utf-8

# # Imports and Instantiation

# In[ ]:


import numpy as np
import pandas as pd

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold
from sklearn.decomposition import PCA
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.feature_selection import SequentialFeatureSelector
from sklearn.neighbors import KNeighborsRegressor

from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score

from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor

import matplotlib.pyplot as plt
import re
from rich import print
from rich.table import Table
from rich.console import Console
import seaborn as sns
from tabulate import tabulate

console = Console()


# # Loading the Data
#
#

# In[ ]:


# Defining the file paths of the train and test datasets
train_set = "data/final_train_data.csv"
test_set = "data/final_test_data.csv"

# Reading the Datasets into their respective Pandas Dataframes
train_data = pd.read_csv(train_set, header=0)
test_data = pd.read_csv(test_set, header=0)

# Dropping the 'Number' Column as it is surplus to requirements
train_data = train_data.drop('Number', axis=1)
test_data = test_data.drop('Number', axis=1)

# Storing all the features in a list of attributes
attributes = train_data.columns.tolist()

# Reorder the columns in test_data to match the column order of train_data
test_data = test_data[attributes]


# In[ ]:


def standardize_features(data, features_to_standardize):
    # Initialize the StandardScaler
    scaler = StandardScaler()

    # Fit the scaler to the selected features and transform them
    scaled_features = scaler.fit_transform(data[features_to_standardize])

    # Create a copy of the original data to retain non-standardized features
    scaled_data = data.copy()

    # Replace the selected features with the standardized values
    scaled_data.loc[:, features_to_standardize] = scaled_features
```

```
    return scaled_data

features_to_standardize = []
for column_name in train_data.columns:
    if 'one_hot' not in column_name and column_name != 'Price':
        features_to_standardize.append(column_name)

standardized_train_data = standardize_features(train_data, features_to_standardize)
standardized_test_data = standardize_features(test_data, features_to_standardize)


# In[ ]:


# styled_df = train_data.style
styled_df = train_data.style

# Apply various formatting options
styled_df = styled_df.set_properties(**{'text-align': 'center'})  # Align text to center
styled_df = styled_df.set_table_styles([{'selector': 'th', 'props': [('font-size', '11pt')]}])  # Set font size for

# Display the styled DataFrame

styled_df


# In[ ]:


def sequential_feature_selection(train_data, num_features):
    # Initialize the KNN model
    model = KNeighborsRegressor(n_neighbors=1)

    # Perform Sequential Forward Selection (SFS)
    sfs = SequentialFeatureSelector(model, n_features_to_select=num_features, scoring='neg_mean_squared_error', cv=
    sfs.fit(train_data.iloc[:, :-1], train_data.iloc[:, -1])

    # Get selected features
    selected_features = train_data.columns[sfs.get_support(indices=True)]

    # Append the last column of the original dataset to selected features
    selected_features = list(selected_features) + [train_data.columns[-1]]

    # Return the modified dataset with selected features
    return selected_features


# In[ ]:


def ufs_feature_selection(train_data, n_features):
    # Initialize UFS
    selector = SelectKBest(score_func=f_regression, k=n_features)

    # Fit UFS to the training data
    selector.fit(train_data.iloc[:, :-1], train_data.iloc[:, -1])

    # Get selected features based on UFS
    selected_features = train_data.columns[selector.get_support(indices=True)]

    # Append the last column of the original dataset to selected features
    selected_features = list(selected_features) + [train_data.columns[-1]]

    return selected_features


# # Error Calculation

# In[ ]:


def RMSE(labels, pred):
    mse_value = mean_squared_error(labels, pred)

    # Calculate RMSE
    rmse_value = np.sqrt(mse_value).round(4)
```

```
    return rmse_value

def MAE(labels, pred):
    mae = mean_absolute_error(labels, pred).round(4)

    return mae

def R2E(labels, pred):
    r_squared_value = r2_score(labels, pred).round(4)

    return r_squared_value


# ### Linear Regression Model

# In[ ]:


def linear_regression_system(train_data, test_data):

    # Define the number of folds for cross-validation
    k = 5  # You can choose any value of k

    # Initialize lists to store error values for each fold
    train_rmse_list, train_mae_list, train_r2e_list = [], [], []
    val_rmse_list, val_mae_list, val_r2e_list = [], [], []

    # Initialize the KFold splitter
    kf = KFold(n_splits=k, shuffle=True, random_state=42)

    # Iterate through each fold
    for train_index, val_index in kf.split(train_data):
        # Split data into training and testing sets
        train_df, val_df = train_data.iloc[train_index], train_data.iloc[val_index]

        # Optionally, reset the index of the new DataFrames
        train_df.reset_index(drop=True, inplace=True)
        val_df.reset_index(drop=True, inplace=True)

        train_inputs = train_df.iloc[:, :-1].copy()
        train_labels = train_df.iloc[:, -1].copy()

        val_inputs = val_df.iloc[:, :-1].copy()
        val_labels = val_df.iloc[:, -1].copy().round(2)

        # Initialize the linear regression model
        model = LinearRegression()
        # Train the model
        model.fit(train_inputs, train_labels)

        train_predictions = model.predict(train_inputs)
        val_predictions = model.predict(val_inputs)

        # Find the range of values in train_labels
        min_value = train_labels.min()
        max_value = train_labels.max()

        # Add tolerance of 20000 to the range
        min_value -= 5000
        max_value += 5000

        # Calculate the mean of all values in train_labels
        mean_value = train_labels.mean()

        # Replace values in val_predictions outside of the range (with tolerance) with the mean value
        val_predictions = np.where((val_predictions < min_value) | (val_predictions > max_value), mean_value, val_p

        # Calculate error metrics for training set
        train_rmse = np.sqrt(mean_squared_error(train_labels, train_predictions))
        train_mae = mean_absolute_error(train_labels, train_predictions)
        train_r2e = r2_score(train_labels, train_predictions)

        # Calculate error metrics for validation set
        val_rmse = np.sqrt(mean_squared_error(val_labels, val_predictions))
        val_mae = mean_absolute_error(val_labels, val_predictions)
        val_r2e = r2_score(val_labels, val_predictions)
```

```
        # Append error values to lists
        train_rmse_list.append(train_rmse)
        train_mae_list.append(train_mae)
        train_r2e_list.append(train_r2e)

        val_rmse_list.append(val_rmse)
        val_mae_list.append(val_mae)
        val_r2e_list.append(val_r2e)

    # Compute mean and standard deviation of error metrics for training and validation sets
    train_rmse_mean = np.mean(train_rmse_list)
    train_rmse_std = np.std(train_rmse_list)
    train_mae_mean = np.mean(train_mae_list)
    train_mae_std = np.std(train_mae_list)
    train_r2e_mean = np.mean(train_r2e_list)
    train_r2e_std = np.std(train_r2e_list)

    val_rmse_mean = np.mean(val_rmse_list)
    val_rmse_std = np.std(val_rmse_list)
    val_mae_mean = np.mean(val_mae_list)
    val_mae_std = np.std(val_mae_list)
    val_r2e_mean = np.mean(val_r2e_list)
    val_r2e_std = np.std(val_r2e_list)

    # Print mean and std deviation of error metrics in rich tables
    console.print("[blue]Mean and Standard Deviation of Error Metrics over 5 Folds:[/blue]\n")
    console.print("[blue]Training Set:[/blue]")

    att_table = Table(title="Error Metrics", show_header=True, header_style="bold magenta")
    att_table.add_column("Metric", style="cyan", justify="center")
    att_table.add_column("Mean", style="cyan", justify="center")
    att_table.add_column("Std Dev", style="cyan", justify="center")

    att_table.add_row("RMSE", f"{train_rmse_mean:.4f}", f"{train_rmse_std:.4f}")
    att_table.add_row("MAE", f"{train_mae_mean:.4f}", f"{train_mae_std:.4f}")
    att_table.add_row("R-squared Error", f"{train_r2e_mean:.4f}", f"{train_r2e_std:.4f}")

    console.print(att_table)

    console.print("\n[blue]Validation Set:[/blue]")

    att_table = Table(title="Error Metrics", show_header=True, header_style="bold magenta")
    att_table.add_column("Metric", style="cyan", justify="center")
    att_table.add_column("Mean", style="cyan", justify="center")
    att_table.add_column("Std Dev", style="cyan", justify="center")

    att_table.add_row("RMSE", f"{val_rmse_mean:.4f}", f"{val_rmse_std:.4f}")
    att_table.add_row("MAE", f"{val_mae_mean:.4f}", f"{val_mae_std:.4f}")
    att_table.add_row("R-squared Error", f"{val_r2e_mean:.4f}", f"{val_r2e_std:.4f}")

    console.print(att_table)

linear_regression_system(train_data, test_data)


# In[ ]:


linear_regression_system(standardized_train_data, standardized_test_data)


# In[ ]:


selected_features = sequential_feature_selection(standardized_train_data, 40)
train_data_selected = standardized_train_data[selected_features].copy()
test_data_selected = standardized_test_data[selected_features].copy()
linear_regression_system(train_data_selected, test_data_selected)


# In[ ]:


selected_features = ufs_feature_selection(standardized_train_data, 28)
train_data_selected = standardized_train_data[selected_features].copy()
test_data_selected = standardized_test_data[selected_features].copy()
```

```
linear_regression_system(train_data_selected, test_data_selected)


# In[ ]:




# <a style='text-decoration:none;line-height:16px;display:flex;color:#5B5B62;padding:10px;justify-content:end;' hre
# <img alt='Created in deepnote.com' style='display:inline;max-height:16px;margin:0px;margin-right:7.5px;' src='dat
# Created in <span style='font-weight:600;margin-left:4px;'>Deepnote</span></a>
```

```python
#!/usr/bin/env python
# coding: utf-8

# # Imports and Instantiation

# In[ ]:


import numpy as np
import pandas as pd

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold
from sklearn.decomposition import PCA
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.feature_selection import SequentialFeatureSelector
from sklearn.neighbors import KNeighborsRegressor

from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score

from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor
from sklearn.linear_model import Ridge

import matplotlib.pyplot as plt
import re
from rich import print
from rich.table import Table
from rich.console import Console
import seaborn as sns

console = Console()


# # Loading the Data
#
#

# In[ ]:


# Defining the file paths of the train and test datasets
train_set = "data/final_train_data.csv"
test_set = "data/final_test_data.csv"

# Reading the Datasets into their respective Pandas Dataframes
train_data = pd.read_csv(train_set, header=0)
test_data = pd.read_csv(test_set, header=0)

# Dropping the 'Number' Column as it is surplus to requirements
train_data = train_data.drop('Number', axis=1)
test_data = test_data.drop('Number', axis=1)

# Storing all the features in a list of attributes
attributes = train_data.columns.tolist()

# Reorder the columns in test_data to match the column order of train_data
test_data = test_data[attributes]


# In[ ]:


def standardize_features(data, features_to_standardize):
    # Initialize the StandardScaler
    scaler = StandardScaler()

    # Fit the scaler to the selected features and transform them
    scaled_features = scaler.fit_transform(data[features_to_standardize])

    # Create a copy of the original data to retain non-standardized features
    scaled_data = data.copy()

    # Replace the selected features with the standardized values
    scaled_data.loc[:, features_to_standardize] = scaled_features
```

```
    return scaled_data

features_to_standardize = []
for column_name in train_data.columns:
    if 'one_hot' not in column_name and column_name != 'Price':
        features_to_standardize.append(column_name)

standardized_train_data = standardize_features(train_data, features_to_standardize)
standardized_test_data = standardize_features(test_data, features_to_standardize)


# In[ ]:


# styled_df = train_data.style
styled_df = train_data.style

# Apply various formatting options
styled_df = styled_df.set_properties(**{'text-align': 'center'})  # Align text to center
styled_df = styled_df.set_table_styles([{'selector': 'th', 'props': [('font-size', '11pt')]}])  # Set font size for

# Display the styled DataFrame

styled_df


# In[ ]:


def sequential_feature_selection(train_data, num_features):
    # Initialize the KNN model
    model = KNeighborsRegressor(n_neighbors=1)

    # Perform Sequential Forward Selection (SFS)
    sfs = SequentialFeatureSelector(model, n_features_to_select=num_features, scoring='neg_mean_squared_error', cv=
    sfs.fit(train_data.iloc[:, :-1], train_data.iloc[:, -1])

    # Get selected features
    selected_features = train_data.columns[sfs.get_support(indices=True)]

    # Append the last column of the original dataset to selected features
    selected_features = list(selected_features) + [train_data.columns[-1]]

    # Return the modified dataset with selected features
    return selected_features


# In[ ]:


def ufs_feature_selection(train_data, n_features):
    # Initialize UFS
    selector = SelectKBest(score_func=f_regression, k=n_features)

    # Fit UFS to the training data
    selector.fit(train_data.iloc[:, :-1], train_data.iloc[:, -1])

    # Get selected features based on UFS
    selected_features = train_data.columns[selector.get_support(indices=True)]

    # Append the last column of the original dataset to selected features
    selected_features = list(selected_features) + [train_data.columns[-1]]

    return selected_features


# # Error Calculation

# In[ ]:


def RMSE(labels, pred):
    mse_value = mean_squared_error(labels, pred)

    # Calculate RMSE
    rmse_value = np.sqrt(mse_value).round(4)
```

```
    return rmse_value

def MAE(labels, pred):
    mae = mean_absolute_error(labels, pred).round(4)

    return mae

def R2E(labels, pred):
    r_squared_value = r2_score(labels, pred).round(4)

    return r_squared_value


# ## k-Nearest Neighbors Regression Model

# In[ ]:


def knn_regression_system(train_data, test_data, k_neighbors=5):

    # Define the number of folds for cross-validation
    k = 5  # You can choose any value of k

    # Initialize lists to store error values for each fold
    train_rmse_list, train_mae_list, train_r2e_list = [], [], []
    val_rmse_list, val_mae_list, val_r2e_list = [], [], []

    # Initialize the KFold splitter
    kf = KFold(n_splits=k, shuffle=True, random_state=42)

    # Iterate through each fold
    for train_index, val_index in kf.split(train_data):
        # Split data into training and testing sets
        train_df, val_df = train_data.iloc[train_index], train_data.iloc[val_index]

        # Optionally, reset the index of the new DataFrames
        train_df.reset_index(drop=True, inplace=True)
        val_df.reset_index(drop=True, inplace=True)

        train_inputs = train_df.iloc[:, :-1].copy()
        train_labels = train_df.iloc[:, -1].copy()

        val_inputs = val_df.iloc[:, :-1].copy()
        val_labels = val_df.iloc[:, -1].copy().round(2)

        # Initialize the kNN regression model
        model = KNeighborsRegressor(n_neighbors=k_neighbors)
        # Train the model
        model.fit(train_inputs, train_labels)

        train_predictions = model.predict(train_inputs)
        val_predictions = model.predict(val_inputs)

        # Calculate error metrics for training set
        train_rmse = np.sqrt(mean_squared_error(train_labels, train_predictions))
        train_mae = mean_absolute_error(train_labels, train_predictions)
        train_r2e = r2_score(train_labels, train_predictions)

        # Calculate error metrics for validation set
        val_rmse = np.sqrt(mean_squared_error(val_labels, val_predictions))
        val_mae = mean_absolute_error(val_labels, val_predictions)
        val_r2e = r2_score(val_labels, val_predictions)

        # Append error values to lists
        train_rmse_list.append(train_rmse)
        train_mae_list.append(train_mae)
        train_r2e_list.append(train_r2e)

        val_rmse_list.append(val_rmse)
        val_mae_list.append(val_mae)
        val_r2e_list.append(val_r2e)

    # Compute mean and standard deviation of error metrics for training and validation sets
    train_rmse_mean = np.mean(train_rmse_list)
    train_rmse_std = np.std(train_rmse_list)
    train_mae_mean = np.mean(train_mae_list)
```

```
    train_mae_std = np.std(train_mae_list)
    train_r2e_mean = np.mean(train_r2e_list)
    train_r2e_std = np.std(train_r2e_list)

    val_rmse_mean = np.mean(val_rmse_list)
    val_rmse_std = np.std(val_rmse_list)
    val_mae_mean = np.mean(val_mae_list)
    val_mae_std = np.std(val_mae_list)
    val_r2e_mean = np.mean(val_r2e_list)
    val_r2e_std = np.std(val_r2e_list)

    # Print mean and std deviation of error metrics in rich tables
    console.print("[blue]Mean and Standard Deviation of Error Metrics over 5 Folds:[/blue]\n")
    console.print("[blue]Training Set:[/blue]")

    att_table = Table(title="Error Metrics", show_header=True, header_style="bold magenta")
    att_table.add_column("Metric", style="cyan", justify="center")
    att_table.add_column("Mean", style="cyan", justify="center")
    att_table.add_column("Std Dev", style="cyan", justify="center")

    att_table.add_row("RMSE", f"{train_rmse_mean:.4f}", f"{train_rmse_std:.4f}")
    att_table.add_row("MAE", f"{train_mae_mean:.4f}", f"{train_mae_std:.4f}")
    att_table.add_row("R-squared Error", f"{train_r2e_mean:.4f}", f"{train_r2e_std:.4f}")

    console.print(att_table)

    console.print("\n[blue]Validation Set:[/blue]")

    att_table = Table(title="Error Metrics", show_header=True, header_style="bold magenta")
    att_table.add_column("Metric", style="cyan", justify="center")
    att_table.add_column("Mean", style="cyan", justify="center")
    att_table.add_column("Std Dev", style="cyan", justify="center")

    att_table.add_row("RMSE", f"{val_rmse_mean:.4f}", f"{val_rmse_std:.4f}")
    att_table.add_row("MAE", f"{val_mae_mean:.4f}", f"{val_mae_std:.4f}")
    att_table.add_row("R-squared Error", f"{val_r2e_mean:.4f}", f"{val_r2e_std:.4f}")

    console.print(att_table)


knn_regression_system(train_data, test_data, k_neighbors=5)


# In[ ]:


knn_regression_system(standardized_train_data, standardized_test_data)


# In[ ]:


selected_features = sequential_feature_selection(standardized_train_data, 37)
train_data_selected = standardized_train_data[selected_features].copy()
test_data_selected = standardized_test_data[selected_features].copy()
knn_regression_system(train_data_selected, test_data_selected)


# In[ ]:


selected_features = ufs_feature_selection(standardized_train_data, 28)
train_data_selected = standardized_train_data[selected_features].copy()
test_data_selected = standardized_test_data[selected_features].copy()
knn_regression_system(train_data_selected, test_data_selected)


# In[ ]:




# <a style='text-decoration:none;line-height:16px;display:flex;color:#5B5B62;padding:10px;justify-content:end;' hre
# <img alt='Created in deepnote.com' style='display:inline;max-height:16px;margin:0px;margin-right:7.5px;' src='dat
# Created in <span style='font-weight:600;margin-left:4px;'>Deepnote</span></a>
```

```
#!/usr/bin/env python
# coding: utf-8

# # Imports and Instantiation

# In[ ]:


import numpy as np
import pandas as pd

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold
from sklearn.decomposition import PCA
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.feature_selection import SequentialFeatureSelector
from sklearn.neighbors import KNeighborsRegressor

from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score

from sklearn.tree import DecisionTreeRegressor
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor
from sklearn.linear_model import Ridge

import matplotlib.pyplot as plt
import re
from rich import print
from rich.table import Table
from rich.console import Console
import seaborn as sns

console = Console()


# # Loading the Data
#
#

# In[ ]:


# Defining the file paths of the train and test datasets
train_set = "data/final_train_data.csv"
test_set = "data/final_test_data.csv"

# Reading the Datasets into their respective Pandas Dataframes
train_data = pd.read_csv(train_set, header=0)
test_data = pd.read_csv(test_set, header=0)

# Dropping the 'Number' Column as it is surplus to requirements
train_data = train_data.drop('Number', axis=1)
test_data = test_data.drop('Number', axis=1)

# Storing all the features in a list of attributes
attributes = train_data.columns.tolist()

# Reorder the columns in test_data to match the column order of train_data
test_data = test_data[attributes]


# In[ ]:


def standardize_features(data, features_to_standardize):
    # Initialize the StandardScaler
    scaler = StandardScaler()

    # Fit the scaler to the selected features and transform them
    scaled_features = scaler.fit_transform(data[features_to_standardize])

    # Create a copy of the original data to retain non-standardized features
    scaled_data = data.copy()

    # Replace the selected features with the standardized values
```

```
    scaled_data.loc[:, features_to_standardize] = scaled_features

    return scaled_data

features_to_standardize = []
for column_name in train_data.columns:
    if 'one_hot' not in column_name and column_name != 'Price':
        features_to_standardize.append(column_name)

standardized_train_data = standardize_features(train_data, features_to_standardize)
standardized_test_data = standardize_features(test_data, features_to_standardize)


# In[ ]:


styled_df = train_data.style

# Apply various formatting options
styled_df = styled_df.set_properties(**{'text-align': 'center'})  # Align text to center
styled_df = styled_df.set_table_styles([{'selector': 'th', 'props': [('font-size', '11pt')]}])  # Set font size for

# Display the styled DataFrame

styled_df


# In[ ]:


def sequential_feature_selection(train_data, num_features):
    # Initialize the KNN model
    model = KNeighborsRegressor(n_neighbors=1)

    # Perform Sequential Forward Selection (SFS)
    sfs = SequentialFeatureSelector(model, n_features_to_select=num_features, scoring='neg_mean_squared_error', cv=
    sfs.fit(train_data.iloc[:, :-1], train_data.iloc[:, -1])

    # Get selected features
    selected_features = train_data.columns[sfs.get_support(indices=True)]

    # Append the last column of the original dataset to selected features
    selected_features = list(selected_features) + [train_data.columns[-1]]

    # Return the modified dataset with selected features
    return selected_features


# In[ ]:


def ufs_feature_selection(train_data, n_features):
    # Initialize UFS
    selector = SelectKBest(score_func=f_regression, k=n_features)

    # Fit UFS to the training data
    selector.fit(train_data.iloc[:, :-1], train_data.iloc[:, -1])

    # Get selected features based on UFS
    selected_features = train_data.columns[selector.get_support(indices=True)]

    # Append the last column of the original dataset to selected features
    selected_features = list(selected_features) + [train_data.columns[-1]]

    return selected_features


# # Error Calculation

# In[ ]:


def RMSE(labels, pred):
    mse_value = mean_squared_error(labels, pred)

    # Calculate RMSE
    rmse_value = np.sqrt(mse_value).round(4)
```

```
    return rmse_value

def MAE(labels, pred):
    mae = mean_absolute_error(labels, pred).round(4)

    return mae

def R2E(labels, pred):
    r_squared_value = r2_score(labels, pred).round(4)

    return r_squared_value


# ## Decision Tree Regression Model

# In[ ]:


def decision_tree_regression_system(train_data, test_data):

    # Define the number of folds for cross-validation
    k = 5  # You can choose any value of k

    # Initialize lists to store error values for each fold
    train_rmse_list, train_mae_list, train_r2e_list = [], [], []
    val_rmse_list, val_mae_list, val_r2e_list = [], [], []

    # Initialize the KFold splitter
    kf = KFold(n_splits=k, shuffle=True, random_state=42)

    # Iterate through each fold
    for train_index, val_index in kf.split(train_data):
        # Split data into training and testing sets
        train_df, val_df = train_data.iloc[train_index], train_data.iloc[val_index]

        # Optionally, reset the index of the new DataFrames
        train_df.reset_index(drop=True, inplace=True)
        val_df.reset_index(drop=True, inplace=True)

        train_inputs = train_df.iloc[:, :-1].copy()
        train_labels = train_df.iloc[:, -1].copy()

        val_inputs = val_df.iloc[:, :-1].copy()
        val_labels = val_df.iloc[:, -1].copy().round(2)

        # Initialize the decision tree regression model
        model = DecisionTreeRegressor(random_state=42)  # You can adjust parameters if needed
        # Train the model
        model.fit(train_inputs, train_labels)

        train_predictions = model.predict(train_inputs)
        val_predictions = model.predict(val_inputs)

        # Calculate error metrics for training set
        train_rmse = np.sqrt(mean_squared_error(train_labels, train_predictions))
        train_mae = mean_absolute_error(train_labels, train_predictions)
        train_r2e = r2_score(train_labels, train_predictions)

        # Calculate error metrics for validation set
        val_rmse = np.sqrt(mean_squared_error(val_labels, val_predictions))
        val_mae = mean_absolute_error(val_labels, val_predictions)
        val_r2e = r2_score(val_labels, val_predictions)

        # Append error values to lists
        train_rmse_list.append(train_rmse)
        train_mae_list.append(train_mae)
        train_r2e_list.append(train_r2e)

        val_rmse_list.append(val_rmse)
        val_mae_list.append(val_mae)
        val_r2e_list.append(val_r2e)

    # Compute mean and standard deviation of error metrics for training and validation sets
    train_rmse_mean = np.mean(train_rmse_list)
    train_rmse_std = np.std(train_rmse_list)
    train_mae_mean = np.mean(train_mae_list)
```

```
    train_mae_std = np.std(train_mae_list)
    train_r2e_mean = np.mean(train_r2e_list)
    train_r2e_std = np.std(train_r2e_list)

    val_rmse_mean = np.mean(val_rmse_list)
    val_rmse_std = np.std(val_rmse_list)
    val_mae_mean = np.mean(val_mae_list)
    val_mae_std = np.std(val_mae_list)
    val_r2e_mean = np.mean(val_r2e_list)
    val_r2e_std = np.std(val_r2e_list)

    # Print mean and std deviation of error metrics in rich tables
    console.print("[blue]Mean and Standard Deviation of Error Metrics over 5 Folds:[/blue]\n")
    console.print("[blue]Training Set:[/blue]")

    att_table = Table(title="Error Metrics", show_header=True, header_style="bold magenta")
    att_table.add_column("Metric", style="cyan", justify="center")
    att_table.add_column("Mean", style="cyan", justify="center")
    att_table.add_column("Std Dev", style="cyan", justify="center")

    att_table.add_row("RMSE", f"{train_rmse_mean:.4f}", f"{train_rmse_std:.4f}")
    att_table.add_row("MAE", f"{train_mae_mean:.4f}", f"{train_mae_std:.4f}")
    att_table.add_row("R-squared Error", f"{train_r2e_mean:.4f}", f"{train_r2e_std:.4f}")

    console.print(att_table)

    console.print("\n[blue]Validation Set:[/blue]")

    att_table = Table(title="Error Metrics", show_header=True, header_style="bold magenta")
    att_table.add_column("Metric", style="cyan", justify="center")
    att_table.add_column("Mean", style="cyan", justify="center")
    att_table.add_column("Std Dev", style="cyan", justify="center")

    att_table.add_row("RMSE", f"{val_rmse_mean:.4f}", f"{val_rmse_std:.4f}")
    att_table.add_row("MAE", f"{val_mae_mean:.4f}", f"{val_mae_std:.4f}")
    att_table.add_row("R-squared Error", f"{val_r2e_mean:.4f}", f"{val_r2e_std:.4f}")

    console.print(att_table)


decision_tree_regression_system(train_data, test_data)


# In[ ]:


decision_tree_regression_system(standardized_train_data, standardized_test_data)


# In[ ]:


selected_features = sequential_feature_selection(standardized_train_data, 40)
train_data_selected = standardized_train_data[selected_features].copy()
test_data_selected = standardized_test_data[selected_features].copy()
decision_tree_regression_system(train_data_selected, test_data_selected)


# In[ ]:


selected_features = ufs_feature_selection(standardized_train_data, 22)
train_data_selected = standardized_train_data[selected_features].copy()
test_data_selected = standardized_test_data[selected_features].copy()
decision_tree_regression_system(train_data_selected, test_data_selected)


# <a style='text-decoration:none;line-height:16px;display:flex;color:#5B5B62;padding:10px;justify-content:end;' hre
# <img alt='Created in deepnote.com' style='display:inline;max-height:16px;margin:0px;margin-right:7.5px;' src='dat
# Created in <span style='font-weight:600;margin-left:4px;'>Deepnote</span></a>
```

```python
#!/usr/bin/env python
# coding: utf-8

# # Imports and Instantiation

# In[ ]:


import numpy as np
import pandas as pd

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold
from sklearn.decomposition import PCA
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.feature_selection import SequentialFeatureSelector
from sklearn.neighbors import KNeighborsRegressor

from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score

from sklearn.neural_network import MLPRegressor
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor
from sklearn.linear_model import Ridge

import torch
import torch.nn as nn
import torch.optim as optim

import matplotlib.pyplot as plt
import re
from rich import print
from rich.table import Table
from rich.console import Console
import seaborn as sns

console = Console()


# # Loading the Data
#
#

# In[ ]:


# Defining the file paths of the train and test datasets
train_set = "data/final_train_data.csv"
test_set = "data/final_test_data.csv"

# Reading the Datasets into their respective Pandas Dataframes
train_data = pd.read_csv(train_set, header=0)
test_data = pd.read_csv(test_set, header=0)

# Dropping the 'Number' Column as it is surplus to requirements
train_data = train_data.drop('Number', axis=1)
test_data = test_data.drop('Number', axis=1)

# Storing all the features in a list of attributes
attributes = train_data.columns.tolist()

# Reorder the columns in test_data to match the column order of train_data
test_data = test_data[attributes]


# In[ ]:


def standardize_features(data, features_to_standardize):
    # Initialize the StandardScaler
    scaler = StandardScaler()

    # Fit the scaler to the selected features and transform them
    scaled_features = scaler.fit_transform(data[features_to_standardize])
```

```
    # Create a copy of the original data to retain non-standardized features
    scaled_data = data.copy()

    # Replace the selected features with the standardized values
    scaled_data.loc[:, features_to_standardize] = scaled_features

    return scaled_data

features_to_standardize = []
for column_name in train_data.columns:
    if 'one_hot' not in column_name and column_name != 'Price':
        features_to_standardize.append(column_name)

standardized_train_data = standardize_features(train_data, features_to_standardize)
standardized_test_data = standardize_features(test_data, features_to_standardize)


# In[ ]:


# styled_df = train_data.style
styled_df = train_data.style

# Apply various formatting options
styled_df = styled_df.set_properties(**{'text-align': 'center'})  # Align text to center
styled_df = styled_df.set_table_styles([{'selector': 'th', 'props': [('font-size', '11pt')]}])  # Set font size for

# Display the styled DataFrame

styled_df


# In[ ]:


def sequential_feature_selection(train_data, num_features):
    # Initialize the KNN model
    model = KNeighborsRegressor(n_neighbors=1)

    # Perform Sequential Forward Selection (SFS)
    sfs = SequentialFeatureSelector(model, n_features_to_select=num_features, scoring='neg_mean_squared_error', cv=
    sfs.fit(train_data.iloc[:, :-1], train_data.iloc[:, -1])

    # Get selected features
    selected_features = train_data.columns[sfs.get_support(indices=True)]

    # Append the last column of the original dataset to selected features
    selected_features = list(selected_features) + [train_data.columns[-1]]

    # Return the modified dataset with selected features
    return selected_features


# In[ ]:


def ufs_feature_selection(train_data, n_features):
    # Initialize UFS
    selector = SelectKBest(score_func=f_regression, k=n_features)

    # Fit UFS to the training data
    selector.fit(train_data.iloc[:, :-1], train_data.iloc[:, -1])

    # Get selected features based on UFS
    selected_features = train_data.columns[selector.get_support(indices=True)]

    # Append the last column of the original dataset to selected features
    selected_features = list(selected_features) + [train_data.columns[-1]]

    return selected_features


# # Error Calculation

# In[ ]:
```

```
def RMSE(labels, pred):
    mse_value = mean_squared_error(labels, pred)

    # Calculate RMSE
    rmse_value = np.sqrt(mse_value).round(4)

    return rmse_value

def MAE(labels, pred):
    mae = mean_absolute_error(labels, pred).round(4)

    return mae

def R2E(labels, pred):
    r_squared_value = r2_score(labels, pred).round(4)

    return abs(r_squared_value)


# ## Multi-Layer Perceptron Regression Model

# In[ ]:


class MLPRegressor(nn.Module):
    def __init__(self, input_size, hidden_sizes):
        super(MLPRegressor, self).__init__()
        self.input_size = input_size
        self.hidden_sizes = hidden_sizes

        self.layers = nn.ModuleList()
        for i in range(len(hidden_sizes)):
            if i == 0:
                self.layers.append(nn.Linear(input_size, hidden_sizes[i]))
            else:
                self.layers.append(nn.Linear(hidden_sizes[i-1], hidden_sizes[i]))
            self.layers.append(nn.ReLU())

        # Output layer
        self.output_layer = nn.Linear(hidden_sizes[-1], 1)

    def forward(self, x):
        for layer in self.layers:
            x = layer(x)
        return self.output_layer(x)


# In[ ]:


def mlp_regression_system(train_data, test_data):

    # Define the number of folds for cross-validation
    k = 5  # You can choose any value of k

    # Initialize lists to store error values for each fold
    train_rmse_list, train_mae_list, train_r2e_list = [], [], []
    val_rmse_list, val_mae_list, val_r2e_list = [], [], []

    # Initialize the KFold splitter
    kf = KFold(n_splits=k, shuffle=True, random_state=42)

    # Iterate through each fold
    for train_index, val_index in kf.split(train_data):
        # Split data into training and testing sets
        train_df, val_df = train_data.iloc[train_index], train_data.iloc[val_index]

        # Optionally, reset the index of the new DataFrames
        train_df.reset_index(drop=True, inplace=True)
        val_df.reset_index(drop=True, inplace=True)

        train_inputs = train_df.iloc[:, :-1].copy()
        train_labels = train_df.iloc[:, -1].copy()

        val_inputs = val_df.iloc[:, :-1].copy()
        val_labels = val_df.iloc[:, -1].copy().round(2)
```

```
    # Initialize the MLP model with dropout
    model = MLPRegressor(input_size=train_inputs.shape[1], hidden_sizes=[128, 256, 128])

    # Define loss function and optimizer
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=0.005)

    # Convert data to PyTorch tensors
    train_inputs_tensor = torch.tensor(train_inputs.values, dtype=torch.float32)
    train_labels_tensor = torch.tensor(train_labels.values, dtype=torch.float32)

    # Train the model
    num_epochs = 100
    for epoch in range(num_epochs):
        # Forward pass
        outputs = model(train_inputs_tensor)
        loss = criterion(outputs, train_labels_tensor.view(-1, 1))

        # Backward pass and optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    # Evaluate on training set
    train_predictions = model(train_inputs_tensor).detach().numpy()
    train_rmse = np.sqrt(mean_squared_error(train_labels, train_predictions))
    train_mae = mean_absolute_error(train_labels, train_predictions)
    train_r2e = r2_score(train_labels, train_predictions)

    # Evaluate on validation set
    val_inputs_tensor = torch.tensor(val_inputs.values, dtype=torch.float32)
    val_labels_tensor = torch.tensor(val_labels.values, dtype=torch.float32)
    val_predictions = model(val_inputs_tensor).detach().numpy()
    val_rmse = np.sqrt(mean_squared_error(val_labels, val_predictions))
    val_mae = mean_absolute_error(val_labels, val_predictions)
    val_r2e = r2_score(val_labels, val_predictions)

    # Append error values to lists
    train_rmse_list.append(train_rmse)
    train_mae_list.append(train_mae)
    train_r2e_list.append(train_r2e)

    val_rmse_list.append(val_rmse)
    val_mae_list.append(val_mae)
    val_r2e_list.append(val_r2e)

# Compute mean and standard deviation of error metrics for training and validation sets
train_rmse_mean = np.mean(train_rmse_list)
train_rmse_std = np.std(train_rmse_list)
train_mae_mean = np.mean(train_mae_list)
train_mae_std = np.std(train_mae_list)
train_r2e_mean = np.mean(train_r2e_list)
train_r2e_std = np.std(train_r2e_list)

val_rmse_mean = np.mean(val_rmse_list)
val_rmse_std = np.std(val_rmse_list)
val_mae_mean = np.mean(val_mae_list)
val_mae_std = np.std(val_mae_list)
val_r2e_mean = np.mean(val_r2e_list)
val_r2e_std = np.std(val_r2e_list)

# Print mean and std deviation of error metrics in rich tables
console.print("[blue]Mean and Standard Deviation of Error Metrics over 5 Folds:[/blue]\n")
console.print("[blue]Training Set:[/blue]")

att_table = Table(title="Error Metrics", show_header=True, header_style="bold magenta")
att_table.add_column("Metric", style="cyan", justify="center")
att_table.add_column("Mean", style="cyan", justify="center")
att_table.add_column("Std Dev", style="cyan", justify="center")

att_table.add_row("RMSE", f"{train_rmse_mean:.4f}", f"{train_rmse_std:.4f}")
att_table.add_row("MAE", f"{train_mae_mean:.4f}", f"{train_mae_std:.4f}")
att_table.add_row("R-squared Error", f"{train_r2e_mean:.4f}", f"{train_r2e_std:.4f}")

console.print(att_table)

console.print("\n[blue]Validation Set:[/blue]")
```

```
    att_table = Table(title="Error Metrics", show_header=True, header_style="bold magenta")
    att_table.add_column("Metric", style="cyan", justify="center")
    att_table.add_column("Mean", style="cyan", justify="center")
    att_table.add_column("Std Dev", style="cyan", justify="center")

    att_table.add_row("RMSE", f"{val_rmse_mean:.4f}", f"{val_rmse_std:.4f}")
    att_table.add_row("MAE", f"{val_mae_mean:.4f}", f"{val_mae_std:.4f}")
    att_table.add_row("R-squared Error", f"{val_r2e_mean:.4f}", f"{val_r2e_std:.4f}")

    console.print(att_table)

mlp_regression_system(train_data, test_data)


# In[ ]:


mlp_regression_system(standardized_train_data, standardized_test_data)


# In[ ]:


selected_features = sequential_feature_selection(standardized_train_data, 37)
train_data_selected = standardized_train_data[selected_features].copy()
test_data_selected = standardized_test_data[selected_features].copy()
mlp_regression_system(train_data_selected, test_data_selected)


# In[ ]:


selected_features = ufs_feature_selection(standardized_train_data, 28)
train_data_selected = standardized_train_data[selected_features].copy()
test_data_selected = standardized_test_data[selected_features].copy()
mlp_regression_system(train_data_selected, test_data_selected)


# <a style='text-decoration:none;line-height:16px;display:flex;color:#5B5B62;padding:10px;justify-content:end;' hre
# <img alt='Created in deepnote.com' style='display:inline;max-height:16px;margin:0px;margin-right:7.5px;' src='dat
# Created in <span style='font-weight:600;margin-left:4px;'>Deepnote</span></a>
```

```python
#!/usr/bin/env python
# coding: utf-8

# # Imports and Instantiation

# In[ ]:


import numpy as np
import pandas as pd

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold
from sklearn.decomposition import PCA
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.feature_selection import SequentialFeatureSelector
from sklearn.neighbors import KNeighborsRegressor

from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score

from sklearn.pipeline import make_pipeline
from sklearn.kernel_approximation import RBFSampler
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor
from sklearn.linear_model import Ridge

import matplotlib.pyplot as plt
import re
from rich import print
from rich.table import Table
from rich.console import Console
import seaborn as sns

console = Console()


# # Loading the Data
#
#

# In[ ]:


# Defining the file paths of the train and test datasets
train_set = "data/final_train_data.csv"
test_set = "data/final_test_data.csv"

# Reading the Datasets into their respective Pandas Dataframes
train_data = pd.read_csv(train_set, header=0)
test_data = pd.read_csv(test_set, header=0)

# Dropping the 'Number' Column as it is surplus to requirements
train_data = train_data.drop('Number', axis=1)
test_data = test_data.drop('Number', axis=1)

# Storing all the features in a list of attributes
attributes = train_data.columns.tolist()

# Reorder the columns in test_data to match the column order of train_data
test_data = test_data[attributes]


# In[ ]:


def standardize_features(data, features_to_standardize):
    # Initialize the StandardScaler
    scaler = StandardScaler()

    # Fit the scaler to the selected features and transform them
    scaled_features = scaler.fit_transform(data[features_to_standardize])

    # Create a copy of the original data to retain non-standardized features
    scaled_data = data.copy()
```

```
    # Replace the selected features with the standardized values
    scaled_data.loc[:, features_to_standardize] = scaled_features

    return scaled_data

features_to_standardize = []
for column_name in train_data.columns:
    if 'one_hot' not in column_name and column_name != 'Price':
        features_to_standardize.append(column_name)

standardized_train_data = standardize_features(train_data, features_to_standardize)
standardized_test_data = standardize_features(test_data, features_to_standardize)


# In[ ]:


# styled_df = train_data.style
styled_df = train_data.style

# Apply various formatting options
styled_df = styled_df.set_properties(**{'text-align': 'center'})  # Align text to center
styled_df = styled_df.set_table_styles([{'selector': 'th', 'props': [('font-size', '11pt')]}])  # Set font size for

# Display the styled DataFrame

styled_df


# In[ ]:


def sequential_feature_selection(train_data, num_features):
    # Initialize the KNN model
    model = KNeighborsRegressor(n_neighbors=1)

    # Perform Sequential Forward Selection (SFS)
    sfs = SequentialFeatureSelector(model, n_features_to_select=num_features, scoring='neg_mean_squared_error', cv=
    sfs.fit(train_data.iloc[:, :-1], train_data.iloc[:, -1])

    # Get selected features
    selected_features = train_data.columns[sfs.get_support(indices=True)]

    # Append the last column of the original dataset to selected features
    selected_features = list(selected_features) + [train_data.columns[-1]]

    # Return the modified dataset with selected features
    return selected_features


# In[ ]:


def ufs_feature_selection(train_data, n_features):
    # Initialize UFS
    selector = SelectKBest(score_func=f_regression, k=n_features)

    # Fit UFS to the training data
    selector.fit(train_data.iloc[:, :-1], train_data.iloc[:, -1])

    # Get selected features based on UFS
    selected_features = train_data.columns[selector.get_support(indices=True)]

    # Append the last column of the original dataset to selected features
    selected_features = list(selected_features) + [train_data.columns[-1]]

    return selected_features


# # Error Calculation

# In[ ]:


def RMSE(labels, pred):
    mse_value = mean_squared_error(labels, pred)
```

```
    # Calculate RMSE
    rmse_value = np.sqrt(mse_value).round(4)

    return rmse_value

def MAE(labels, pred):
    mae = mean_absolute_error(labels, pred).round(4)

    return mae

def R2E(labels, pred):
    r_squared_value = r2_score(labels, pred).round(4)

    return r_squared_value


# ## RBF Neural Network Model

# In[ ]:


def rbf_regression_system(train_data, test_data):

    # Define the number of folds for cross-validation
    k = 5  # You can choose any value of k

    # Initialize lists to store error values for each fold
    train_rmse_list, train_mae_list, train_r2e_list = [], [], []
    val_rmse_list, val_mae_list, val_r2e_list = [], [], []

    # Initialize the KFold splitter
    kf = KFold(n_splits=k, shuffle=True, random_state=42)

    # Iterate through each fold
    for train_index, val_index in kf.split(train_data):
        # Split data into training and testing sets
        train_df, val_df = train_data.iloc[train_index], train_data.iloc[val_index]

        # Optionally, reset the index of the new DataFrames
        train_df.reset_index(drop=True, inplace=True)
        val_df.reset_index(drop=True, inplace=True)

        train_inputs = train_df.iloc[:, :-1].copy()
        train_labels = train_df.iloc[:, -1].copy()

        val_inputs = val_df.iloc[:, :-1].copy()
        val_labels = val_df.iloc[:, -1].copy().round(2)

        # Create an RBF regression pipeline
        rbf_features = RBFSampler(gamma=0.1, n_components=100, random_state=42)  # Adjust gamma and n_components
        scaler = StandardScaler()
        ridge = Ridge(alpha=0.1)  # Adjust alpha (regularization strength)
        model = make_pipeline(rbf_features, scaler, ridge)

        # Train the model
        model.fit(train_inputs, train_labels)

        train_predictions = model.predict(train_inputs)
        val_predictions = model.predict(val_inputs)

        # Calculate error metrics for training set
        train_rmse = np.sqrt(mean_squared_error(train_labels, train_predictions))
        train_mae = mean_absolute_error(train_labels, train_predictions)
        train_r2e = r2_score(train_labels, train_predictions)

        # Calculate error metrics for validation set
        val_rmse = np.sqrt(mean_squared_error(val_labels, val_predictions))
        val_mae = mean_absolute_error(val_labels, val_predictions)
        val_r2e = r2_score(val_labels, val_predictions)

        # Append error values to lists
        train_rmse_list.append(train_rmse)
        train_mae_list.append(train_mae)
        train_r2e_list.append(train_r2e)

        val_rmse_list.append(val_rmse)
        val_mae_list.append(val_mae)
```

```
        val_r2e_list.append(val_r2e)

    # Compute mean and standard deviation of error metrics for training and validation sets
    train_rmse_mean = np.mean(train_rmse_list)
    train_rmse_std = np.std(train_rmse_list)
    train_mae_mean = np.mean(train_mae_list)
    train_mae_std = np.std(train_mae_list)
    train_r2e_mean = np.mean(train_r2e_list)
    train_r2e_std = np.std(train_r2e_list)

    val_rmse_mean = np.mean(val_rmse_list)
    val_rmse_std = np.std(val_rmse_list)
    val_mae_mean = np.mean(val_mae_list)
    val_mae_std = np.std(val_mae_list)
    val_r2e_mean = np.mean(val_r2e_list)
    val_r2e_std = np.std(val_r2e_list)

    # Print mean and std deviation of error metrics in rich tables
    console.print("[blue]Mean and Standard Deviation of Error Metrics over 5 Folds:[/blue]\n")
    console.print("[blue]Training Set:[/blue]")

    att_table = Table(title="Error Metrics", show_header=True, header_style="bold magenta")
    att_table.add_column("Metric", style="cyan", justify="center")
    att_table.add_column("Mean", style="cyan", justify="center")
    att_table.add_column("Std Dev", style="cyan", justify="center")

    att_table.add_row("RMSE", f"{train_rmse_mean:.4f}", f"{train_rmse_std:.4f}")
    att_table.add_row("MAE", f"{train_mae_mean:.4f}", f"{train_mae_std:.4f}")
    att_table.add_row("R-squared Error", f"{train_r2e_mean:.4f}", f"{train_r2e_std:.4f}")

    console.print(att_table)

    console.print("\n[blue]Validation Set:[/blue]")

    att_table = Table(title="Error Metrics", show_header=True, header_style="bold magenta")
    att_table.add_column("Metric", style="cyan", justify="center")
    att_table.add_column("Mean", style="cyan", justify="center")
    att_table.add_column("Std Dev", style="cyan", justify="center")

    att_table.add_row("RMSE", f"{val_rmse_mean:.4f}", f"{val_rmse_std:.4f}")
    att_table.add_row("MAE", f"{val_mae_mean:.4f}", f"{val_mae_std:.4f}")
    att_table.add_row("R-squared Error", f"{val_r2e_mean:.4f}", f"{val_r2e_std:.4f}")

    console.print(att_table)

rbf_regression_system(train_data, test_data)


# In[ ]:


rbf_regression_system(standardized_train_data, standardized_test_data)


# In[ ]:


selected_features = sequential_feature_selection(standardized_train_data, 35)
train_data_selected = standardized_train_data[selected_features].copy()
test_data_selected = standardized_test_data[selected_features].copy()
rbf_regression_system(train_data_selected, test_data_selected)


# In[ ]:


selected_features = ufs_feature_selection(standardized_train_data, 28)
train_data_selected = standardized_train_data[selected_features].copy()
test_data_selected = standardized_test_data[selected_features].copy()
rbf_regression_system(train_data_selected, test_data_selected)


# <a style='text-decoration:none;line-height:16px;display:flex;color:#5B5B62;padding:10px;justify-content:end;' hre
# <img alt='Created in deepnote.com' style='display:inline;max-height:16px;margin:0px;margin-right:7.5px;' src='dat
# Created in <span style='font-weight:600;margin-left:4px;'>Deepnote</span></a>
```

```python
#!/usr/bin/env python
# coding: utf-8

# # Imports and Instantiation

# In[ ]:


import numpy as np
import pandas as pd

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold
from sklearn.decomposition import PCA
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.feature_selection import SequentialFeatureSelector
from sklearn.neighbors import KNeighborsRegressor

from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score

from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor
from sklearn.linear_model import Ridge

import matplotlib.pyplot as plt
import re
from rich import print
from rich.table import Table
from rich.console import Console
import seaborn as sns

console = Console()


# # Loading the Data
#
#

# In[ ]:


# Defining the file paths of the train and test datasets
train_set = "data/final_train_data.csv"
test_set = "data/final_test_data.csv"

# Reading the Datasets into their respective Pandas Dataframes
train_data = pd.read_csv(train_set, header=0)
test_data = pd.read_csv(test_set, header=0)

# Dropping the 'Number' Column as it is surplus to requirements
train_data = train_data.drop('Number', axis=1)
test_data = test_data.drop('Number', axis=1)

# Storing all the features in a list of attributes
attributes = train_data.columns.tolist()

# Reorder the columns in test_data to match the column order of train_data
test_data = test_data[attributes]


# In[ ]:


def standardize_features(data, features_to_standardize):
    # Initialize the StandardScaler
    scaler = StandardScaler()

    # Fit the scaler to the selected features and transform them
    scaled_features = scaler.fit_transform(data[features_to_standardize])

    # Create a copy of the original data to retain non-standardized features
    scaled_data = data.copy()

    # Replace the selected features with the standardized values
    scaled_data.loc[:, features_to_standardize] = scaled_features
```

```
    return scaled_data

features_to_standardize = []
for column_name in train_data.columns:
    if 'one_hot' not in column_name and column_name != 'Price':
        features_to_standardize.append(column_name)

standardized_train_data = standardize_features(train_data, features_to_standardize)
standardized_test_data = standardize_features(test_data, features_to_standardize)


# In[ ]:


# styled_df = train_data.style
styled_df = train_data.style

# Apply various formatting options
styled_df = styled_df.set_properties(**{'text-align': 'center'})  # Align text to center
styled_df = styled_df.set_table_styles([{'selector': 'th', 'props': [('font-size', '11pt')]}])  # Set font size for

# Display the styled DataFrame

styled_df


# In[ ]:


def sequential_feature_selection(train_data, num_features):
    # Initialize the KNN model
    model = KNeighborsRegressor(n_neighbors=1)

    # Perform Sequential Forward Selection (SFS)
    sfs = SequentialFeatureSelector(model, n_features_to_select=num_features, scoring='neg_mean_squared_error', cv=
    sfs.fit(train_data.iloc[:, :-1], train_data.iloc[:, -1])

    # Get selected features
    selected_features = train_data.columns[sfs.get_support(indices=True)]

    # Append the last column of the original dataset to selected features
    selected_features = list(selected_features) + [train_data.columns[-1]]

    # Return the modified dataset with selected features
    return selected_features


# In[ ]:


def ufs_feature_selection(train_data, n_features):
    # Initialize UFS
    selector = SelectKBest(score_func=f_regression, k=n_features)

    # Fit UFS to the training data
    selector.fit(train_data.iloc[:, :-1], train_data.iloc[:, -1])

    # Get selected features based on UFS
    selected_features = train_data.columns[selector.get_support(indices=True)]

    # Append the last column of the original dataset to selected features
    selected_features = list(selected_features) + [train_data.columns[-1]]

    return selected_features


# # Error Calculation

# In[ ]:


def RMSE(labels, pred):
    mse_value = mean_squared_error(labels, pred)

    # Calculate RMSE
    rmse_value = np.sqrt(mse_value).round(4)
```

```
    return rmse_value

def MAE(labels, pred):
    mae = mean_absolute_error(labels, pred).round(4)

    return mae

def R2E(labels, pred):
    r_squared_value = r2_score(labels, pred).round(4)

    return r_squared_value


# ## Ridge Regression Model

# In[ ]:


def ridge_regression_system(train_data, test_data):

    # Define the number of folds for cross-validation
    k = 5  # You can choose any value of k

    # Initialize lists to store error values for each fold
    train_rmse_list, train_mae_list, train_r2e_list = [], [], []
    val_rmse_list, val_mae_list, val_r2e_list = [], [], []

    # Initialize the KFold splitter
    kf = KFold(n_splits=k, shuffle=True, random_state=42)

    # Iterate through each fold
    for train_index, val_index in kf.split(train_data):
        # Split data into training and testing sets
        train_df, val_df = train_data.iloc[train_index], train_data.iloc[val_index]

        # Optionally, reset the index of the new DataFrames
        train_df.reset_index(drop=True, inplace=True)
        val_df.reset_index(drop=True, inplace=True)

        train_inputs = train_df.iloc[:, :-1].copy()
        train_labels = train_df.iloc[:, -1].copy()

        val_inputs = val_df.iloc[:, :-1].copy()
        val_labels = val_df.iloc[:, -1].copy().round(2)

        # Initialize the linear regression model
        alpha = 1.0  # Regularization strength, you can adjust this value
        model = Ridge(alpha=alpha)
        # Train the model
        model.fit(train_inputs, train_labels)

        train_predictions = model.predict(train_inputs)
        val_predictions = model.predict(val_inputs)

        # Calculate error metrics for training set
        train_rmse = np.sqrt(mean_squared_error(train_labels, train_predictions))
        train_mae = mean_absolute_error(train_labels, train_predictions)
        train_r2e = r2_score(train_labels, train_predictions)

        # Calculate error metrics for validation set
        val_rmse = np.sqrt(mean_squared_error(val_labels, val_predictions))
        val_mae = mean_absolute_error(val_labels, val_predictions)
        val_r2e = r2_score(val_labels, val_predictions)

        # Append error values to lists
        train_rmse_list.append(train_rmse)
        train_mae_list.append(train_mae)
        train_r2e_list.append(train_r2e)

        val_rmse_list.append(val_rmse)
        val_mae_list.append(val_mae)
        val_r2e_list.append(val_r2e)

    # Compute mean and standard deviation of error metrics for training and validation sets
    train_rmse_mean = np.mean(train_rmse_list)
    train_rmse_std = np.std(train_rmse_list)
```

```
    train_mae_mean = np.mean(train_mae_list)
    train_mae_std = np.std(train_mae_list)
    train_r2e_mean = np.mean(train_r2e_list)
    train_r2e_std = np.std(train_r2e_list)

    val_rmse_mean = np.mean(val_rmse_list)
    val_rmse_std = np.std(val_rmse_list)
    val_mae_mean = np.mean(val_mae_list)
    val_mae_std = np.std(val_mae_list)
    val_r2e_mean = np.mean(val_r2e_list)
    val_r2e_std = np.std(val_r2e_list)

    # Print mean and std deviation of error metrics in rich tables
    console.print("[blue]Mean and Standard Deviation of Error Metrics over 5 Folds:[/blue]\n")
    console.print("[blue]Training Set:[/blue]")

    att_table = Table(title="Error Metrics", show_header=True, header_style="bold magenta")
    att_table.add_column("Metric", style="cyan", justify="center")
    att_table.add_column("Mean", style="cyan", justify="center")
    att_table.add_column("Std Dev", style="cyan", justify="center")

    att_table.add_row("RMSE", f"{train_rmse_mean:.4f}", f"{train_rmse_std:.4f}")
    att_table.add_row("MAE", f"{train_mae_mean:.4f}", f"{train_mae_std:.4f}")
    att_table.add_row("R-squared Error", f"{train_r2e_mean:.4f}", f"{train_r2e_std:.4f}")

    console.print(att_table)

    console.print("\n[blue]Validation Set:[/blue]")

    att_table = Table(title="Error Metrics", show_header=True, header_style="bold magenta")
    att_table.add_column("Metric", style="cyan", justify="center")
    att_table.add_column("Mean", style="cyan", justify="center")
    att_table.add_column("Std Dev", style="cyan", justify="center")

    att_table.add_row("RMSE", f"{val_rmse_mean:.4f}", f"{val_rmse_std:.4f}")
    att_table.add_row("MAE", f"{val_mae_mean:.4f}", f"{val_mae_std:.4f}")
    att_table.add_row("R-squared Error", f"{val_r2e_mean:.4f}", f"{val_r2e_std:.4f}")

    console.print(att_table)

ridge_regression_system(train_data, test_data)


# In[ ]:


ridge_regression_system(standardized_train_data, standardized_test_data)


# In[ ]:


selected_features = sequential_feature_selection(standardized_train_data, 37)
train_data_selected = standardized_train_data[selected_features].copy()
test_data_selected = standardized_test_data[selected_features].copy()
ridge_regression_system(train_data_selected, test_data_selected)


# In[ ]:


selected_features = ufs_feature_selection(standardized_train_data, 28)
train_data_selected = standardized_train_data[selected_features].copy()
test_data_selected = standardized_test_data[selected_features].copy()
ridge_regression_system(train_data_selected, test_data_selected)


# In[ ]:




# <a style='text-decoration:none;line-height:16px;display:flex;color:#5B5B62;padding:10px;justify-content:end;' hre
# <img alt='Created in deepnote.com' style='display:inline;max-height:16px;margin:0px;margin-right:7.5px;' src='dat
# Created in <span style='font-weight:600;margin-left:4px;'>Deepnote</span></a>
```

```
#!/usr/bin/env python
# coding: utf-8

# # Imports and Instantiation

# In[ ]:


import numpy as np
import pandas as pd

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold
from sklearn.decomposition import PCA
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.feature_selection import SequentialFeatureSelector
from sklearn.neighbors import KNeighborsRegressor

from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score

from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVR

import matplotlib.pyplot as plt
import re
from rich import print
from rich.table import Table
from rich.console import Console
import seaborn as sns

console = Console()


# # Loading the Data
#
#

# In[ ]:


# Defining the file paths of the train and test datasets
train_set = "data/final_train_data.csv"
test_set = "data/final_test_data.csv"

# Reading the Datasets into their respective Pandas Dataframes
train_data = pd.read_csv(train_set, header=0)
test_data = pd.read_csv(test_set, header=0)

# Dropping the 'Number' Column as it is surplus to requirements
train_data = train_data.drop('Number', axis=1)
test_data = test_data.drop('Number', axis=1)

# Storing all the features in a list of attributes
attributes = train_data.columns.tolist()

# Reorder the columns in test_data to match the column order of train_data
test_data = test_data[attributes]


# In[ ]:


def standardize_features(data, features_to_standardize):
    # Initialize the StandardScaler
    scaler = StandardScaler()

    # Fit the scaler to the selected features and transform them
    scaled_features = scaler.fit_transform(data[features_to_standardize])

    # Create a copy of the original data to retain non-standardized features
    scaled_data = data.copy()

    # Replace the selected features with the standardized values
```

```
    scaled_data.loc[:, features_to_standardize] = scaled_features

    return scaled_data

features_to_standardize = []
for column_name in train_data.columns:
    if 'one_hot' not in column_name and column_name != 'Price':
        features_to_standardize.append(column_name)

standardized_train_data = standardize_features(train_data, features_to_standardize)
standardized_test_data = standardize_features(test_data, features_to_standardize)


# In[ ]:


styled_df = train_data.style

# Apply various formatting options
styled_df = styled_df.set_properties(**{'text-align': 'center'})  # Align text to center
styled_df = styled_df.set_table_styles([{'selector': 'th', 'props': [('font-size', '11pt')]}])  # Set font size for

# Display the styled DataFrame

styled_df


# In[ ]:


def sequential_feature_selection(train_data, num_features):
    # Initialize the KNN model
    model = KNeighborsRegressor(n_neighbors=1)

    # Perform Sequential Forward Selection (SFS)
    sfs = SequentialFeatureSelector(model, n_features_to_select=num_features, scoring='neg_mean_squared_error', cv=
    sfs.fit(train_data.iloc[:, :-1], train_data.iloc[:, -1])

    # Get selected features
    selected_features = train_data.columns[sfs.get_support(indices=True)]

    # Append the last column of the original dataset to selected features
    selected_features = list(selected_features) + [train_data.columns[-1]]

    # Return the modified dataset with selected features
    return selected_features


# In[ ]:


def ufs_feature_selection(train_data, n_features):
    # Initialize UFS
    selector = SelectKBest(score_func=f_regression, k=n_features)

    # Fit UFS to the training data
    selector.fit(train_data.iloc[:, :-1], train_data.iloc[:, -1])

    # Get selected features based on UFS
    selected_features = train_data.columns[selector.get_support(indices=True)]

    # Append the last column of the original dataset to selected features
    selected_features = list(selected_features) + [train_data.columns[-1]]

    return selected_features


# # Error Calculation

# In[ ]:


def RMSE(labels, pred):
    mse_value = mean_squared_error(labels, pred)

    # Calculate RMSE
    rmse_value = np.sqrt(mse_value).round(4)
```

```
    return rmse_value

def MAE(labels, pred):
    mae = mean_absolute_error(labels, pred).round(4)

    return mae

def R2E(labels, pred):
    r_squared_value = r2_score(labels, pred).round(4)

    return r_squared_value


# ## Support Vector Regression Model

# In[ ]:


def svr_system(train_data, test_data):

    # Define the number of folds for cross-validation
    k = 5  # You can choose any value of k

    # Initialize lists to store error values for each fold
    train_rmse_list, train_mae_list, train_r2e_list = [], [], []
    val_rmse_list, val_mae_list, val_r2e_list = [], [], []

    # Initialize the KFold splitter
    kf = KFold(n_splits=k, shuffle=True, random_state=42)

    # Iterate through each fold
    for train_index, val_index in kf.split(train_data):
        # Split data into training and testing sets
        train_df, val_df = train_data.iloc[train_index], train_data.iloc[val_index]

        # Optionally, reset the index of the new DataFrames
        train_df.reset_index(drop=True, inplace=True)
        val_df.reset_index(drop=True, inplace=True)

        train_inputs = train_df.iloc[:, :-1].copy()
        train_labels = train_df.iloc[:, -1].copy()

        val_inputs = val_df.iloc[:, :-1].copy()
        val_labels = val_df.iloc[:, -1].copy().round(2)

        # Define the parameter grid for grid search
        param_grid = {
            'C': [0.1, 1, 10, 100],
            'gamma': [0.01, 0.1, 1, 10]
        }

        # Initialize SVR model with RBF kernel
        svr_rbf = SVR(kernel='rbf')

        # Initialize GridSearchCV
        grid_search = GridSearchCV(estimator=svr_rbf, param_grid=param_grid, cv=5, scoring='neg_mean_squared_error'

        # Train the model
        grid_search.fit(train_inputs, train_labels)

        # Get the best parameters
        best_params = grid_search.best_params_
        print("Best Parameters:", best_params)

        # Instantiate SVR model with best parameters
        model = SVR(kernel='rbf', C=best_params['C'], gamma=best_params['gamma'])

        # Train the model with the best parameters
        model.fit(train_inputs, train_labels)

        train_predictions = model.predict(train_inputs)
        val_predictions = model.predict(val_inputs)

        # Calculate error metrics for training set
        train_rmse = RMSE(train_labels, train_predictions)
        train_mae = MAE(train_labels, train_predictions)
```

```
        train_r2e = R2E(train_labels, train_predictions)

        # Calculate error metrics for validation set
        val_rmse = RMSE(val_labels, val_predictions)
        val_mae = MAE(val_labels, val_predictions)
        val_r2e = R2E(val_labels, val_predictions)

        # Append error values to lists
        train_rmse_list.append(abs(train_rmse))
        train_mae_list.append(abs(train_mae))
        train_r2e_list.append(abs(train_r2e))

        val_rmse_list.append(abs(val_rmse))
        val_mae_list.append(abs(val_mae))
        val_r2e_list.append(abs(val_r2e))

    # Compute mean and standard deviation of error metrics for training and validation sets
    train_rmse_mean = np.mean(train_rmse_list)
    train_rmse_std = np.std(train_rmse_list)
    train_mae_mean = np.mean(train_mae_list)
    train_mae_std = np.std(train_mae_list)
    train_r2e_mean = np.mean(train_r2e_list)
    train_r2e_std = np.std(train_r2e_list)

    val_rmse_mean = np.mean(val_rmse_list)
    val_rmse_std = np.std(val_rmse_list)
    val_mae_mean = np.mean(val_mae_list)
    val_mae_std = np.std(val_mae_list)
    val_r2e_mean = np.mean(val_r2e_list)
    val_r2e_std = np.std(val_r2e_list)

    # Print mean and std deviation of error metrics in rich tables
    console.print("[blue]Mean and Standard Deviation of Error Metrics over 5 Folds:[/blue]\n")
    console.print("[blue]Training Set:[/blue]")

    att_table = Table(title="Error Metrics", show_header=True, header_style="bold magenta")
    att_table.add_column("Metric", style="cyan", justify="center")
    att_table.add_column("Mean", style="cyan", justify="center")
    att_table.add_column("Std Dev", style="cyan", justify="center")

    att_table.add_row("RMSE", f"{train_rmse_mean:.4f}", f"{train_rmse_std:.4f}")
    att_table.add_row("MAE", f"{train_mae_mean:.4f}", f"{train_mae_std:.4f}")
    att_table.add_row("R-squared Error", f"{train_r2e_mean:.4f}", f"{train_r2e_std:.4f}")

    console.print(att_table)

    console.print("\n[blue]Validation Set:[/blue]")

    att_table = Table(title="Error Metrics", show_header=True, header_style="bold magenta")
    att_table.add_column("Metric", style="cyan", justify="center")
    att_table.add_column("Mean", style="cyan", justify="center")
    att_table.add_column("Std Dev", style="cyan", justify="center")

    att_table.add_row("RMSE", f"{val_rmse_mean:.4f}", f"{val_rmse_std:.4f}")
    att_table.add_row("MAE", f"{val_mae_mean:.4f}", f"{val_mae_std:.4f}")
    att_table.add_row("R-squared Error", f"{val_r2e_mean:.4f}", f"{val_r2e_std:.4f}")

    console.print(att_table)


svr_system(train_data, test_data)


# In[ ]:


selected_features = sequential_feature_selection(standardized_train_data, 37)
train_data_selected = standardized_train_data[selected_features].copy()
test_data_selected = standardized_test_data[selected_features].copy()
svr_system(train_data_selected, test_data_selected)


# In[ ]:


SVRselected_features = ufs_feature_selection(standardized_train_data, 35)
train_data_selected = standardized_train_data[selected_features].copy()
```

```
test_data_selected = standardized_test_data[selected_features].copy()
svr_system(train_data_selected, test_data_selected)


# <a style='text-decoration:none;line-height:16px;display:flex;color:#5B5B62;padding:10px;justify-content:end;' hre
# <img alt='Created in deepnote.com' style='display:inline;max-height:16px;margin:0px;margin-right:7.5px;' src='dat
# Created in <span style='font-weight:600;margin-left:4px;'>Deepnote</span></a>
```

```python
#!/usr/bin/env python
# coding: utf-8

# # Imports and Instantiation

# In[ ]:


import numpy as np
import pandas as pd

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold
from sklearn.decomposition import PCA
from sklearn.feature_selection import SelectKBest, f_regression

from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score

from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor

import matplotlib.pyplot as plt
import re
from rich import print
from rich.table import Table
from rich.console import Console
import seaborn as sns

console = Console()


# # Loading the Data
#
#

# In[ ]:


# Defining the file paths of the train and test datasets
train_set = "data/final_train_data.csv"
test_set = "data/final_test_data.csv"

# Reading the Datasets into their respective Pandas Dataframes
train_data = pd.read_csv(train_set, header=0)
test_data = pd.read_csv(test_set, header=0)

# Dropping the 'Number' Column as it is surplus to requirements
train_data = train_data.drop('Number', axis=1)
test_data = test_data.drop('Number', axis=1)

# Storing all the features in a list of attributes
attributes = train_data.columns.tolist()

# Reorder the columns in test_data to match the column order of train_data
test_data = test_data[attributes]


# In[ ]:


def standardize_features(data, features_to_standardize):
    # Initialize the StandardScaler
    scaler = StandardScaler()

    # Fit the scaler to the selected features and transform them
    scaled_features = scaler.fit_transform(data[features_to_standardize])

    # Create a copy of the original data to retain non-standardized features
    scaled_data = data.copy()

    # Replace the selected features with the standardized values
    scaled_data.loc[:, features_to_standardize] = scaled_features

    return scaled_data
```

```
features_to_standardize = []
for column_name in train_data.columns:
    if 'one_hot' not in column_name and column_name != 'Price':
        features_to_standardize.append(column_name)

standardized_train_data = standardize_features(train_data, features_to_standardize)
standardized_test_data = standardize_features(test_data, features_to_standardize)


# In[ ]:


# styled_df = train_data.style
styled_df = train_data.style

# Apply various formatting options
styled_df = styled_df.set_properties(**{'text-align': 'center'})  # Align text to center
styled_df = styled_df.set_table_styles([{'selector': 'th', 'props': [('font-size', '11pt')]}])  # Set font size for

# Display the styled DataFrame

styled_df


# ## Pearson Correlation Matrix

# In[ ]:


def pearson(train_data, num_features):
    # Fill missing values with the mean of each column
    train_data_filled = train_data.fillna(train_data.mean())

    # Compute Pearson correlation coefficients
    corr_matrix = train_data_filled.corr()
    corr_with_target = corr_matrix.iloc[:-1, -1]

    # Plot correlation matrix
    plt.figure(figsize=(10, 8))
    sns.heatmap(corr_matrix, cmap='coolwarm', linewidths=0.5)
    plt.title('Correlation Matrix')
    plt.show()


pearson(train_data, 15)


# In[ ]:


styled_df = test_data.style

# Apply various formatting options
styled_df = styled_df.set_properties(**{'text-align': 'center'})  # Align text to center
styled_df = styled_df.set_table_styles([{'selector': 'th', 'props': [('font-size', '11pt')]}])  # Set font size for

# Display the styled DataFrame

styled_df


# ## Trivial Model
# • A system that always outputs the mean output value y from the training set

# In[ ]:


class TrivialModel:
    def __init__(self, data):
        self.feature_data = data.iloc[:, :-1].copy()
        self.labels = data.iloc[:, -1].copy()
        self.labels = self.labels.round(2)

    def fit(self):
        self.output = self.labels.mean().round(2)

    def predict(self, user_input):
```

```
        return self.output

    def RMSE(self, labels, pred):
        mse_value = mean_squared_error(labels, pred)

        # Calculate RMSE
        rmse_value = np.sqrt(mse_value)

        return rmse_value

    def MAE(self, labels, pred):
        mae = mean_absolute_error(labels, pred)

        return mae

    def R2E(self, labels, pred):
        r_squared_value = r2_score(labels, pred).round(2)

        return r_squared_value


def trivial_system(train_data, test_data, split):
    # Shuffle the DataFrame (optional but recommended)
    train_data = train_data.sample(frac=1).reset_index(drop=True)

    # Determine the size of the training set (e.g., 80%)
    train_size = split

    # Split the DataFrame into train and validation sets
    train_df = train_data.iloc[:int(len(train_data) * train_size)]
    val_df = train_data.iloc[int(len(train_data) * train_size):]

    # Optionally, reset the index of the new DataFrames
    train_df.reset_index(drop=True, inplace=True)
    val_df.reset_index(drop=True, inplace=True)

    val_inputs = val_df.iloc[:, :-1].copy()
    val_labels = val_df.iloc[:, -1].copy().round(2)

    test_inputs = test_data.iloc[:, :-1].copy()
    test_labels = test_data.iloc[:, -1].copy().round(2)

    model = TrivialModel(train_df)
    model.fit()

    train_predictions = []
    val_predictions = []
    predictions = []

    for _, data_pt in model.feature_data.iterrows():
        output = model.predict(data_pt)
        train_predictions.append(output)

    console.print(f'Trivial Model\'s Laptop Price Prediction: {output}\n\n')

    console.print('Performance on Training Set: \n')

    rmse = model.RMSE(model.labels, train_predictions)
    mae = model.MAE(model.labels, train_predictions)
    r2e = abs(model.R2E(model.labels, train_predictions))

    att_table = Table()
    att_table.add_column(f'RMSE', style='blue')
    att_table.add_column(f'MAE', style='green')
    att_table.add_column(f'R-squared Error', style='red')

    att_table.add_row(str(rmse), str(mae), str(r2e))

    console.print(att_table)

    for _, data_pt in val_inputs.iterrows():
        output = model.predict(data_pt)
        val_predictions.append(output)

    console.print('Performance on Validation Set: \n')

    rmse = model.RMSE(val_labels, val_predictions)
```

```
    mae = model.MAE(val_labels, val_predictions)
    r2e = abs(model.R2E(val_labels, val_predictions))

    att_table = Table()
    att_table.add_column(f'RMSE', style='blue')
    att_table.add_column(f'MAE', style='green')
    att_table.add_column(f'R-squared Error', style='red')

    att_table.add_row(str(rmse), str(mae), str(r2e))

    console.print(att_table)


training_splits = [0.85, 0.8, 0.7, 0.6, 0.5]

for split in training_splits:
    print(f'Training-Validation Split: {split * 100}%')
    trivial_system(train_data, test_data, split)


# <a style='text-decoration:none;line-height:16px;display:flex;color:#5B5B62;padding:10px;justify-content:end;' hre
# <img alt='Created in deepnote.com' style='display:inline;max-height:16px;margin:0px;margin-right:7.5px;' src='dat
# Created in <span style='font-weight:600;margin-left:4px;'>Deepnote</span></a>
```

```
#!/usr/bin/env python
# coding: utf-8

# # Plots of Predicstions vs. True Prices

# In[ ]:


import numpy as np
import pandas as pd

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold
# from sklearn.decomposition import PCA
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.feature_selection import SequentialFeatureSelector
from sklearn.neighbors import KNeighborsRegressor

# from sklearn.metrics import mean_squared_error
# from sklearn.metrics import mean_absolute_error
# from sklearn.metrics import r2_score

from sklearn.neural_network import MLPRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor
from sklearn.pipeline import make_pipeline
from sklearn.kernel_approximation import RBFSampler
from sklearn.linear_model import Ridge
from sklearn.svm import SVR

import matplotlib.pyplot as plt
import random
# import re
# from rich import print
# from rich.table import Table
# from rich.console import Console
# import seaborn as sns

# console = Console()


# ## Loading Data

# In[ ]:


# Defining the file paths of the train and test datasets
train_set = "data/final_train_data.csv"
test_set = "data/final_test_data.csv"

# Reading the Datasets into their respective Pandas Dataframes
train_data = pd.read_csv(train_set, header=0)
test_data = pd.read_csv(test_set, header=0)

# Dropping the 'Number' Column as it is surplus to requirements
train_data = train_data.drop('Number', axis=1)
test_data = test_data.drop('Number', axis=1)

# Storing all the features in a list of attributes
attributes = train_data.columns.tolist()

# Reorder the columns in test_data to match the column order of train_data
test_data = test_data[attributes]
true_vs_pred = [test_data.iloc[:, -1].copy().round(2)]


# ## Getting Predictions

# In[ ]:


def export_predictions(test_pred, model_name):
    # pred_file = f'predictions/{model_name}_pred.csv'
    # test_pred.to_csv(pred_file, index=False)

    # pred_file = f'predictions/{model_name}_pred.csv'
```

```
    # Convert NumPy array to pandas DataFrame
    pred_df = pd.DataFrame(test_pred, columns=['Predictions'])

    # Define the file path for saving the predictions
    pred_file = f'predictions/{model_name}_pred.csv'

    # Save DataFrame to CSV file
    pred_df.to_csv(pred_file, index=False)


# ## Feature Selection

# In[ ]:


def ufs_feature_selection(train_data, n_features):
    # Initialize UFS
    selector = SelectKBest(score_func=f_regression, k=n_features)

    # Fit UFS to the training data
    selector.fit(train_data.iloc[:, :-1], train_data.iloc[:, -1])

    # Get selected features based on UFS
    selected_features = train_data.columns[selector.get_support(indices=True)]

    # Append the last column of the original dataset to selected features
    selected_features = list(selected_features) + [train_data.columns[-1]]

    return selected_features

selected_features = ufs_feature_selection(train_data, 28)
train_data = train_data[selected_features].copy()
test_data = test_data[selected_features].copy()


# ### 1-NN Base Model

# In[ ]:


def nearest_neighbour_system_pred(train_data, test_data, k_neighbors):

    # Define the number of folds for cross-validation
    k = 5  # You can choose any value of k

    # Initialize the KFold splitter
    kf = KFold(n_splits=k, shuffle=True, random_state=42)

    # Iterate through each fold
    for train_index, val_index in kf.split(train_data):
        # Split data into training and testing sets
        train_df = train_data.iloc[train_index]

        # Optionally, reset the index of the new DataFrames
        train_df.reset_index(drop=True, inplace=True)
        # val_df.reset_index(drop=True, inplace=True)

        train_inputs = train_df.iloc[:, :-1].copy()
        train_labels = train_df.iloc[:, -1].copy()

        # val_inputs = val_df.iloc[:, :-1].copy()
        # val_labels = val_df.iloc[:, -1].copy().round(2)

        # Initialize the linear regression model
        model = KNeighborsRegressor(n_neighbors=k_neighbors)
        # Train the model
        model.fit(train_inputs, train_labels)

        # train_predictions = model.predict(train_inputs)
        # val_predictions = model.predict(val_inputs)

    test_inputs = test_data.iloc[:, :-1].copy()
    test_labels = test_data.iloc[:, -1].copy().round(2)

    predictions = model.predict(test_inputs)
```

```
    return predictions


one_nn_BM_pred = nearest_neighbour_system_pred(train_data, test_data, 1)
export_predictions(one_nn_BM_pred, '1_nn_BM')
true_vs_pred.append(one_nn_BM_pred)


# ### Linear Regression Base Model

# In[ ]:


def linear_regression_system_pred(train_data, test_data):

    # Define the number of folds for cross-validation
    k = 5  # You can choose any value of k

    # Initialize the KFold splitter
    kf = KFold(n_splits=k, shuffle=True, random_state=42)

    # Iterate through each fold
    for train_index, val_index in kf.split(train_data):
        # Split data into training and testing sets
        train_df = train_data.iloc[train_index]

        # Optionally, reset the index of the new DataFrames
        train_df.reset_index(drop=True, inplace=True)
        # val_df.reset_index(drop=True, inplace=True)

        train_inputs = train_df.iloc[:, :-1].copy()
        train_labels = train_df.iloc[:, -1].copy()

        # val_inputs = val_df.iloc[:, :-1].copy()
        # val_labels = val_df.iloc[:, -1].copy().round(2)

        # Initialize the linear regression model
        model = LinearRegression()
        # Train the model
        model.fit(train_inputs, train_labels)


    test_inputs = test_data.iloc[:, :-1].copy()
    test_labels = test_data.iloc[:, -1].copy().round(2)

    predictions = model.predict(test_inputs)

    # Find the range of values in train_labels
    min_value = train_labels.min()
    max_value = train_labels.max()

    # Add tolerance of 20000 to the range
    min_value -= 5000
    max_value += 5000

    # Calculate the mean of all values in train_labels
    mean_value = train_labels.mean()

    # Replace values in predictions outside of the range (with tolerance) with the mean value
    predictions = np.where((predictions < min_value) | (predictions > max_value), mean_value, predictions)

    return predictions

lin_reg_BM_pred = linear_regression_system_pred(train_data, test_data)
export_predictions(lin_reg_BM_pred, 'lin_reg_BM')
true_vs_pred.append(lin_reg_BM_pred)


# ### Decision Tree Regression

# In[ ]:


def decision_tree_regression_system_pred(train_data, test_data):

    # Define the number of folds for cross-validation
    k = 5  # You can choose any value of k
```

```
    # Initialize the KFold splitter
    kf = KFold(n_splits=k, shuffle=True, random_state=42)

    # Iterate through each fold
    for train_index, val_index in kf.split(train_data):
        # Split data into training and testing sets
        train_df = train_data.iloc[train_index]
        # Optionally, reset the index of the new DataFrames
        train_df.reset_index(drop=True, inplace=True)

        train_inputs = train_df.iloc[:, :-1].copy()
        train_labels = train_df.iloc[:, -1].copy()

        # Initialize the decision tree regression model
        model = DecisionTreeRegressor(random_state=42)  # You can adjust parameters if needed
        # Train the model
        model.fit(train_inputs, train_labels)

    test_inputs = test_data.iloc[:, :-1].copy()
    test_labels = test_data.iloc[:, -1].copy().round(2)

    predictions = model.predict(test_inputs)

    return predictions


dec_tree_pred = decision_tree_regression_system_pred(train_data, test_data)
export_predictions(dec_tree_pred, 'dec_tree')
true_vs_pred.append(dec_tree_pred)


# ### k-Nearest Neighbors

# In[ ]:


knn_pred = nearest_neighbour_system_pred(train_data, test_data, 5)
export_predictions(knn_pred, 'kNN')
true_vs_pred.append(knn_pred)


# ### Multilayer Perceptron (MLP)

# In[ ]:


def mlp_regression_system_pred(train_data, test_data):

    # Define the number of folds for cross-validation
    k = 5  # You can choose any value of k

    # Initialize the KFold splitter
    kf = KFold(n_splits=k, shuffle=True, random_state=42)

    # Iterate through each fold
    for train_index, val_index in kf.split(train_data):
        # Split data into training and testing sets
        train_df = train_data.iloc[train_index]

        # Optionally, reset the index of the new DataFrames
        train_df.reset_index(drop=True, inplace=True)

        train_inputs = train_df.iloc[:, :-1].copy()
        train_labels = train_df.iloc[:, -1].copy()

        # Initialize the MLP model
        model = MLPRegressor(hidden_layer_sizes=(120, 40), activation='relu', random_state=42)
        # Train the model
        model.fit(train_inputs, train_labels)


    test_inputs = test_data.iloc[:, :-1].copy()
    test_labels = test_data.iloc[:, -1].copy().round(2)

    predictions = model.predict(test_inputs)
```

```
    return predictions


mlp_pred = mlp_regression_system_pred(train_data, test_data)
export_predictions(mlp_pred, 'MLP')
true_vs_pred.append(mlp_pred)


# ### Radial Basis Function (RBF) Network

# In[ ]:


def rbf_regression_system_pred(train_data, test_data):

    # Define the number of folds for cross-validation
    k = 5  # You can choose any value of k

    # Initialize the KFold splitter
    kf = KFold(n_splits=k, shuffle=True, random_state=42)

    # Iterate through each fold
    for train_index, val_index in kf.split(train_data):
        # Split data into training and testing sets
        train_df = train_data.iloc[train_index]

        # Optionally, reset the index of the new DataFrames
        train_df.reset_index(drop=True, inplace=True)


        train_inputs = train_df.iloc[:, :-1].copy()
        train_labels = train_df.iloc[:, -1].copy()

        # Create an RBF regression pipeline
        rbf_features = RBFSampler(gamma=0.1, n_components=100, random_state=42)  # Adjust gamma and n_components
        scaler = StandardScaler()
        ridge = Ridge(alpha=0.1)  # Adjust alpha (regularization strength)
        model = make_pipeline(rbf_features, scaler, ridge)

        # Train the model
        model.fit(train_inputs, train_labels)

    test_inputs = test_data.iloc[:, :-1].copy()
    test_labels = test_data.iloc[:, -1].copy().round(2)

    predictions = model.predict(test_inputs)

    return predictions


rbf_pred = rbf_regression_system_pred(train_data, test_data)
export_predictions(rbf_pred, 'RBF')
true_vs_pred.append(rbf_pred)


# ### Ridge Regression

# In[ ]:


def ridge_regression_system(train_data, test_data):

    # Define the number of folds for cross-validation
    k = 5  # You can choose any value of k

    # Initialize the KFold splitter
    kf = KFold(n_splits=k, shuffle=True, random_state=42)

    # Iterate through each fold
    for train_index, val_index in kf.split(train_data):
        # Split data into training and testing sets
        train_df = train_data.iloc[train_index]

        # Optionally, reset the index of the new DataFrames
        train_df.reset_index(drop=True, inplace=True)

        train_inputs = train_df.iloc[:, :-1].copy()
```

```
        train_labels = train_df.iloc[:, -1].copy()

        # Initialize the linear regression model
        alpha = 1.0  # Regularization strength, you can adjust this value
        model = Ridge(alpha=alpha)
        # Train the model
        model.fit(train_inputs, train_labels)


    test_inputs = test_data.iloc[:, :-1].copy()
    test_labels = test_data.iloc[:, -1].copy().round(2)

    predictions = model.predict(test_inputs)

    return predictions


ridge_reg_pred = ridge_regression_system(train_data, test_data)
export_predictions(ridge_reg_pred, 'ridge_reg')
true_vs_pred.append(ridge_reg_pred)


# ### Support Vector Regression (SVR)

# In[ ]:


def svr_system_pred(train_data, test_data):

    # Define the number of folds for cross-validation
    k = 5  # You can choose any value of k

    # Initialize the KFold splitter
    kf = KFold(n_splits=k, shuffle=True, random_state=42)

    # Iterate through each fold
    for train_index, val_index in kf.split(train_data):
        # Split data into training and testing sets
        train_df = train_data.iloc[train_index]

        # Optionally, reset the index of the new DataFrames
        train_df.reset_index(drop=True, inplace=True)
        # val_df.reset_index(drop=True, inplace=True)

        train_inputs = train_df.iloc[:, :-1].copy()
        train_labels = train_df.iloc[:, -1].copy()

        # Instantiate SVR model with best parameters
        model = SVR(kernel='rbf', C=10, gamma=0.01)

        # Train the model with the best parameters
        model.fit(train_inputs, train_labels)


    test_inputs = test_data.iloc[:, :-1].copy()
    test_labels = test_data.iloc[:, -1].copy().round(2)

    predictions = model.predict(test_inputs)

    return predictions

svr_pred = svr_system_pred(train_data, test_data)
export_predictions(svr_pred, 'SVR')
true_vs_pred.append(svr_pred)


# ### Trivial System

# In[ ]:


class TrivialModel:
    def __init__(self, data):
        self.feature_data = data.iloc[:, :-1].copy()
        self.labels = data.iloc[:, -1].copy()
        self.labels = self.labels.round(2)
```

```
    def fit(self):
        self.output = self.labels.mean().round(2)

    def predict(self, user_input):
        return self.output


def trivial_system_pred(train_data, test_data, split):
    # Shuffle the DataFrame (optional but recommended)
    train_data = train_data.sample(frac=1).reset_index(drop=True)

    # Determine the size of the training set (e.g., 80%)
    train_size = split

    # Split the DataFrame into train and validation sets
    train_df = train_data.iloc[:int(len(train_data) * train_size)]

    # Optionally, reset the index of the new DataFrames
    train_df.reset_index(drop=True, inplace=True)

    test_inputs = test_data.iloc[:, :-1].copy()
    test_labels = test_data.iloc[:, -1].copy().round(2)

    model = TrivialModel(train_df)
    model.fit()

    predictions = []


    for _, data_pt in test_inputs.iterrows():
        output = model.predict(data_pt)
        predictions.append(output)

    return predictions


training_splits = [0.85]

for split in training_splits:
    trivial_sys_pred = trivial_system_pred(train_data, test_data, split)
    split_str = str(int(split*100))
    export_predictions(trivial_sys_pred, f'trivial_sys_{split_str}')
    true_vs_pred.append(trivial_sys_pred)


# ## Load Predictions

# In[ ]:


filenames = ['1_nn_BM_pred.csv', 'lin_reg_BM_pred.csv', 'dec_tree_pred.csv', 'kNN_pred.csv',
             'MLP_pred.csv', 'RBF_pred.csv', 'ridge_reg_pred.csv', 'SVR_pred.csv', 'trivial_sys_50_pred.csv',
             'trivial_sys_60_pred.csv', 'trivial_sys_70_pred.csv', 'trivial_sys_80_pred.csv',
             'trivial_sys_85_pred.csv']

# List to store the data from each CSV file
data_list = [test_data.iloc[:, -1].copy().round(2)]

# Iterate over each file in the directory
for filename in filenames:
    data = pd.read_csv(f'predictions/{filename}', header=None, index_col=None, skiprows = 1)
    # Extract the column containing the data
    data_column = data.iloc[:, 0]

    # Convert the column to a list
    data_check = data_column.tolist()
    # Extract the values as a Series and then convert it to a list
    data_list.append(data_check)



# ## Plot

# In[ ]:


def plot_predictions_scatter(model_names, true_vs_pred):
```

```
    plt.figure(figsize=(16,12))

    x_vals = range(len(true_vs_pred[0]))
    # Plot predictions for each model
    for model_name, y_vals in zip(model_names, true_vs_pred):
        if model_name == 'True Price':
            plt.scatter(x_vals, y_vals, alpha=1, label=model_name, marker='o', color = 'red')
        else:
            plt.scatter(x_vals, y_vals, alpha=0.6, label=model_name, marker = 'x')


    plt.xlabel('Data Points')
    plt.ylabel('Prices')
    plt.title('Predictions and True Values')
    plt.legend()
    plt.show()


model_names = ['True Price', '1-NN BM', 'Linear Regression BM', 'Decision Tree Regression', 'k-NN', 'MLP', 'RBF', '

# Assuming models_predictions_dict is a dictionary where keys are model names and values are arrays or pandas Serie
plot_predictions_scatter(model_names, data_list)


# In[ ]:


def plot_predictions_scatter(model_names, true_vs_pred):
    num_models = len(model_names)-1
    fig, axes = plt.subplots(nrows=num_models, ncols=1, figsize=(8, 4*num_models))

    for i, (model_name, y_vals) in enumerate(zip(model_names[1:], true_vs_pred[1:])):
        color = [0.2, random.random(), random.random()]  # Random RGB color

        ax = axes[i]
        ax.scatter(range(len(y_vals)), true_vs_pred[0], alpha=1, label=model_names[0], marker = 'x', color = 'red')
        ax.scatter(range(len(y_vals)), y_vals, alpha=0.5, label=model_name, color=color)
        ax.set_xlabel('Data Points')
        ax.set_ylabel('Prices')
        ax.set_title(f'Predictions and True Values - {model_name}')
        ax.legend()

    plt.tight_layout()
    plt.show()

# Example usage
model_names = ['True Price', '1-NN BM', 'Linear Regression BM', 'Decision Tree Regression',
               'k-NN', 'MLP', 'RBF', 'Ridge Regression', 'Trivial System 85%']

plot_predictions_scatter(model_names, data_list)


# <a style='text-decoration:none;line-height:16px;display:flex;color:#5B5B62;padding:10px;justify-content:end;' hre
# <img alt='Created in deepnote.com' style='display:inline;max-height:16px;margin:0px;margin-right:7.5px;' src='dat
# Created in <span style='font-weight:600;margin-left:4px;'>Deepnote</span></a>
```

```python
#!/usr/bin/env python
# coding: utf-8

# # Imports and Instantiation

# In[ ]:


import numpy as np
import pandas as pd

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold
from sklearn.decomposition import PCA
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.feature_selection import SequentialFeatureSelector
from sklearn.neighbors import KNeighborsRegressor

from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score

from sklearn.pipeline import make_pipeline
from sklearn.kernel_approximation import RBFSampler
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor
from sklearn.linear_model import Ridge

import matplotlib.pyplot as plt
import re
from rich import print
from rich.table import Table
from rich.console import Console
import seaborn as sns
from tabulate import tabulate

console = Console()


# # Loading the Data
#
#

# In[ ]:


# Defining the file paths of the train and test datasets
train_set = "data/final_train_data.csv"
test_set = "data/final_test_data.csv"

# Reading the Datasets into their respective Pandas Dataframes
train_data = pd.read_csv(train_set, header=0)
test_data = pd.read_csv(test_set, header=0)

# Dropping the 'Number' Column as it is surplus to requirements
train_data = train_data.drop('Number', axis=1)
test_data = test_data.drop('Number', axis=1)

# Storing all the features in a list of attributes
attributes = train_data.columns.tolist()

# Reorder the columns in test_data to match the column order of train_data
test_data = test_data[attributes]


# # Data Standardization

# In[ ]:


def standardize_features(data, features_to_standardize):
    # Initialize the StandardScaler
    scaler = StandardScaler()

    # Fit the scaler to the selected features and transform them
    scaled_features = scaler.fit_transform(data[features_to_standardize])
```

```
    # Create a copy of the original data to retain non-standardized features
    scaled_data = data.copy()

    # Replace the selected features with the standardized values
    scaled_data.loc[:, features_to_standardize] = scaled_features

    return scaled_data

features_to_standardize = []
for column_name in train_data.columns:
    if 'one_hot' not in column_name and column_name != 'Price':
        features_to_standardize.append(column_name)

standardized_train_data = standardize_features(train_data, features_to_standardize)
standardized_test_data = standardize_features(test_data, features_to_standardize)


# # Feature Selection Method(s)

# In[ ]:


def ufs_feature_selection(train_data, n_features):
    # Initialize UFS
    selector = SelectKBest(score_func=f_regression, k=n_features)

    # Fit UFS to the training data
    selector.fit(train_data.iloc[:, :-1], train_data.iloc[:, -1])

    # Get selected features based on UFS
    selected_features = train_data.columns[selector.get_support(indices=True)]

    # Append the last column of the original dataset to selected features
    selected_features = list(selected_features) + [train_data.columns[-1]]

    return selected_features


# # Error Computation

# In[ ]:


def RMSE(labels, pred):
    mse_value = mean_squared_error(labels, pred)

    # Calculate RMSE
    rmse_value = np.sqrt(mse_value).round(4)

    return rmse_value

def MAE(labels, pred):
    mae = mean_absolute_error(labels, pred).round(4)

    return mae

def R2E(labels, pred):
    r_squared_value = r2_score(labels, pred).round(4)

    return r_squared_value


# # Model Performance on Test Data

# ## Trivial Model

# In[ ]:


class TrivialModel:
    def __init__(self, data):
        self.feature_data = data.iloc[:, :-1].copy()
        self.labels = data.iloc[:, -1].copy()
        self.labels = self.labels.round(2)

    def fit(self):
        self.output = self.labels.mean().round(2)
```

```
    def predict(self, user_input):
        return self.output

    def RMSE(self, labels, pred):
        mse_value = mean_squared_error(labels, pred)

        # Calculate RMSE
        rmse_value = np.sqrt(mse_value)

        return rmse_value

    def MAE(self, labels, pred):
        mae = mean_absolute_error(labels, pred)

        return mae

    def R2E(self, labels, pred):
        r_squared_value = r2_score(labels, pred).round(2)

        return r_squared_value


def trivial_system(train_data, test_data):
    # Shuffle the DataFrame (optional but recommended)
    train_data = train_data.sample(frac=1).reset_index(drop=True)

    test_inputs = test_data.iloc[:, :-1].copy()
    test_labels = test_data.iloc[:, -1].copy().round(2)

    model = TrivialModel(train_data)
    model.fit()

    predictions = []

    for _, data_pt in test_inputs.iterrows():
        output = model.predict(data_pt)
        predictions.append(output)

    console.print(f'Trivial Model\'s Laptop Price Prediction: {output}\n\n')

    console.print('Performance on Testing Set: \n')

    rmse = model.RMSE(test_labels, predictions)
    mae = model.MAE(test_labels, predictions)
    r2e = model.R2E(test_labels, predictions)

    att_table = Table()
    att_table.add_column(f'RMSE', style='blue')
    att_table.add_column(f'MAE', style='green')
    att_table.add_column(f'R-squared Error', style='red')

    att_table.add_row(str(rmse.round(4)), str(mae.round(4)), str(abs(r2e)))

    console.print(att_table)


trivial_system(train_data, test_data)


# ## 1-Nearest Neighbor (Baseline)

# In[ ]:


def nearest_neighbour_system(train_data, test_data):
    # Split data into training and testing sets
    train_inputs = train_data.iloc[:, :-1].copy()
    train_labels = train_data.iloc[:, -1].copy()

    test_inputs = test_data.iloc[:, :-1].copy()
    test_labels = test_data.iloc[:, -1].copy().round(2)

    # Initialize the nearest neighbor model
    model = KNeighborsRegressor(n_neighbors=1)

    # Train the model
```

```
    model.fit(train_inputs, train_labels)

    # Predict on test set
    test_predictions = model.predict(test_inputs)

    # Calculate error metrics for test set
    test_rmse = np.sqrt(mean_squared_error(test_labels, test_predictions))
    test_mae = mean_absolute_error(test_labels, test_predictions)
    test_r2e = r2_score(test_labels, test_predictions)

    console.print("\n[blue]Error Metrics for Test Set:[/blue]\n")
    print_error_metrics(test_rmse, test_mae, test_r2e)

def print_error_metrics(rmse, mae, r2e):
    att_table = Table(title="Error Metrics", show_header=True, header_style="bold magenta")
    att_table.add_column("Metric", style="cyan", justify="center")
    att_table.add_column("Value", style="cyan", justify="center")

    att_table.add_row("RMSE", f"{rmse:.4f}")
    att_table.add_row("MAE", f"{mae:.4f}")
    att_table.add_row("R-squared Error", f"{r2e:.4f}")

    console.print(att_table)

selected_features = ufs_feature_selection(standardized_train_data, 22)
train_data_selected = standardized_train_data[selected_features].copy()
test_data_selected = standardized_test_data[selected_features].copy()
nearest_neighbour_system(train_data_selected, test_data_selected)


# ## Linear Regression Model (Baseline)

# In[ ]:


def linear_regression_system(train_data, test_data):
    # Train the model on the entire training set
    train_inputs = train_data.iloc[:, :-1].copy()
    train_labels = train_data.iloc[:, -1].copy()

    model = LinearRegression()
    model.fit(train_inputs, train_labels)

    # Predictions on test set
    test_inputs = test_data.iloc[:, :-1].copy()
    test_labels = test_data.iloc[:, -1].copy().round(2)
    predictions = model.predict(test_inputs)

    # Find the range of values in train_labels
    min_value = train_labels.min()
    max_value = train_labels.max()

    # Add tolerance of 20000 to the range
    min_value -= 5000
    max_value += 5000

    # Calculate the mean of all values in train_labels
    mean_value = train_labels.mean()

    # Replace values in predictions outside of the range (with tolerance) with the mean value
    predictions = np.where((predictions < min_value) | (predictions > max_value), mean_value, predictions)

    # Calculate error metrics for the test set
    rmse = np.sqrt(mean_squared_error(test_labels, predictions))
    mae = mean_absolute_error(test_labels, predictions)
    r2e = r2_score(test_labels, predictions)

    # Print the results table for the test data
    console.print("\n[blue]Performance on Testing Set:[/blue]")

    att_table = Table(title="Error Metrics", show_header=True, header_style="bold magenta")
    att_table.add_column("Metric", style="cyan", justify="center")
    att_table.add_column("Value", style="cyan", justify="center")

    att_table.add_row("RMSE", f"{rmse:.4f}")
    att_table.add_row("MAE", f"{mae:.4f}")
    att_table.add_row("R-squared Error", f"{r2e:.4f}")
```

```
    console.print(att_table)

selected_features = ufs_feature_selection(standardized_train_data, 22)
train_data_selected = standardized_train_data[selected_features].copy()
test_data_selected = standardized_test_data[selected_features].copy()
linear_regression_system(train_data_selected, test_data_selected)


# ## Ridge Regression: Highest Performing Model on the Test Dataset

# In[ ]:


def ridge_regression_system(train_data, test_data):
    # Split data into training and testing sets
    train_inputs = train_data.iloc[:, :-1].copy()
    train_labels = train_data.iloc[:, -1].copy()

    test_inputs = test_data.iloc[:, :-1].copy()
    test_labels = test_data.iloc[:, -1].copy().round(2)

    # Initialize the linear regression model
    alpha = 0.75  # Regularization strength, you can adjust this value
    model = Ridge(alpha=alpha)

    # Train the model
    model.fit(train_inputs, train_labels)

    # Predict on test set
    test_predictions = model.predict(test_inputs)

    # Calculate error metrics for test set
    test_rmse = np.sqrt(mean_squared_error(test_labels, test_predictions))
    test_mae = mean_absolute_error(test_labels, test_predictions)
    test_r2e = r2_score(test_labels, test_predictions)

    console.print("\n[blue]Error Metrics for Test Set:[/blue]\n")
    print_error_metrics(test_rmse, test_mae, test_r2e)

def print_error_metrics(rmse, mae, r2e):
    att_table = Table(title="Error Metrics", show_header=True, header_style="bold magenta")
    att_table.add_column("Metric", style="cyan", justify="center")
    att_table.add_column("Value", style="cyan", justify="center")

    att_table.add_row("RMSE", f"{rmse:.4f}")
    att_table.add_row("MAE", f"{mae:.4f}")
    att_table.add_row("R-squared Error", f"{r2e:.4f}")

    console.print(att_table)

selected_features = ufs_feature_selection(standardized_train_data, 22)
train_data_selected = standardized_train_data[selected_features].copy()
test_data_selected = standardized_test_data[selected_features].copy()
ridge_regression_system(train_data_selected, test_data_selected)


# <a style='text-decoration:none;line-height:16px;display:flex;color:#5B5B62;padding:10px;justify-content:end;' hre
# <img alt='Created in deepnote.com' style='display:inline;max-height:16px;margin:0px;margin-right:7.5px;' src='dat
# Created in <span style='font-weight:600;margin-left:4px;'>Deepnote</span></a>
```