Naif A. Ganadily Programming Assignment 1

EE P 596: Advanced Introduction to Machine Learning

Programming Assignment 1: Linear Regression

Due January 14th, 2023, by 11:59 PM

Student: Naif A Ganadily

Instructor - Prof. Karthik Mohan
TA - Ayush Singh
Grader - Fatwir SM

Required Libraries:

In [1]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import csv
from sklearn import datasets, linear_model, metrics
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression

from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
from sklearn.linear_model import LassoCV

# What I added:
# StandardScaler from documentation: https://scikit-learn.org/stable/modules/
generated/sklearn.preprocessing.StandardScaler.html
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import MinMaxScaler, OneHotEncoder
```

Cellphone Dataset

Loading the Dataset:

Code:

In [2]:

```
from IPython.display import display, HTML

css = """
.output {
    flex-direction: row;
}
"""

HTML('<style>{}</style>'.format(css))


# Return it in the format of a pandas dataframe.
def dataset_loader():
  df = pd.read_csv('cellphones_data.csv')
  return df


# For the given dataset, print the first 10 lines.
def starting_printer(df):
  return df.head(10)

df = dataset_loader()
display(df)
display(starting_printer(df))
```

End of Code

Explanation:

The code loads a CSV file named cellphones_data.csv using the Pandas library function read_csv, which returns a Pandas DataFrame containing the data from the CSV file. The DataFrame is then stored in a variable called "df". The "starting_printer(df)" function is used to print the first 10 lines of a data frame with the head() function. IPython's "display" function is then used to display the dataframe and the first 10 rows of the dataframe. CSS code is used to change the layout of the output.

Column Header Modification:

Code:

In [3]:

```
# I am changing some of the comlumn headers from 'word word' to 'word_word' d
ue to be able to call them when needed
df.columns = ['cellphone_id', 'brand', 'model', 'operating_system', 'internal
_memory', 'ram', 'performance', 'main_camera', 'selfie_camera', 'battery_size
', 'screen_size', 'weight', 'price', 'release_date']
```

In [4]:

```
# Checking to see if it worked which apparently it did!
display(starting_printer(df))
```

End of Code

Explanation:

The code uses the pandas dataframe's .columns attribute to rename the dataframe's columns. They did this to make it easier to access the data in the columns with the new column headings in the DataFrame. Also, these new column names follow Python's variable naming conventions, making it easier to reference these columns in code. Also, the new column names are more readable and contain more information than the old ones.

Data Type Analysis:

Code:

In [5]:

```
# Checking to see which attributes are Categorical Data and Numerical Data
def get_column_types(df):
  # Get data types of columns
  column_types = df.dtypes

  # Separate numerical and categorical columns
  # numerical_columns anything which is not an 'object' using !=
  # categorical_columns anything which is an 'object' using ==
  numerical_columns = column_types[column_types != 'object'].index
  categorical_columns = column_types[column_types == 'object'].index

  return numerical_columns, categorical_columns

numerical_columns, categorical_columns = get_column_types(df)
print(f'Numerical columns: {numerical_columns}')
print(f'Categorical columns: {categorical_columns}')
```

```
Numerical columns: Index(['cellphone_id', 'internal_memory', 'ram', 'performa
nce', 'main_camera',
       'selfie_camera', 'battery_size', 'screen_size', 'weight', 'price'],
      dtype='object')
Categorical columns: Index(['brand', 'model', 'operating_system', 'release_da
te'], dtype='object')
```

End of Code

Explanation:

The data types of all the columns in the DataFrame, which are stored in a variable called "column _ types," are being retrieved by the code using the pandas DataForma's". dtypees" attribute. The numerical columns are then separated from the categorical columns in the code using boolean indexing. Catagorical columns are defined as those whose data type is an "object," whereas numerical columns define those that are not. The numerical and categorical column names are returned by the code. You did this so that you could distinguish between columns that contain numerical data and categorical data. Knowing the data types of the columns can help in further preprocessing and data analysis tasks such as filling missing values, feature scaling and encoding categorical data

Creating Dummy Variables:

Code:

From this we can conclude that the only Categorical Data in the Dataframe are ['brand', 'model', 'operating_system'] I dont think we can consider ['release_date'] to be a categorical data due to it being in a dd/mm/yyyy format

In [6]:

```
# After checking we found out that the only Categorical Data in the Dataframe are ['brand', 'model', 'operating_system']
# We create a new dataframe df1 for the new dummy attributes

def categorization(df):
  df = pd.get_dummies(df, columns=['brand', 'model', 'operating_system'])
  return df



# Print out first few rows of these dummy attributes.
categorization(df).head(3)
```

End of Code

Explanation:

It is used to transform categorical variables, which have a limited number of possible values, into numeric variables that can be used in statistical models. By creating a new df1 dataframe with these dummy attributes, you effectively extend the original dataframe by adding a new binary column for each unique category of the categorical variables Make, Model, and Operating System. This allows you to include these categorical variables in your analysis while preserving their individual contributions to the model.

Histogram Plotting:

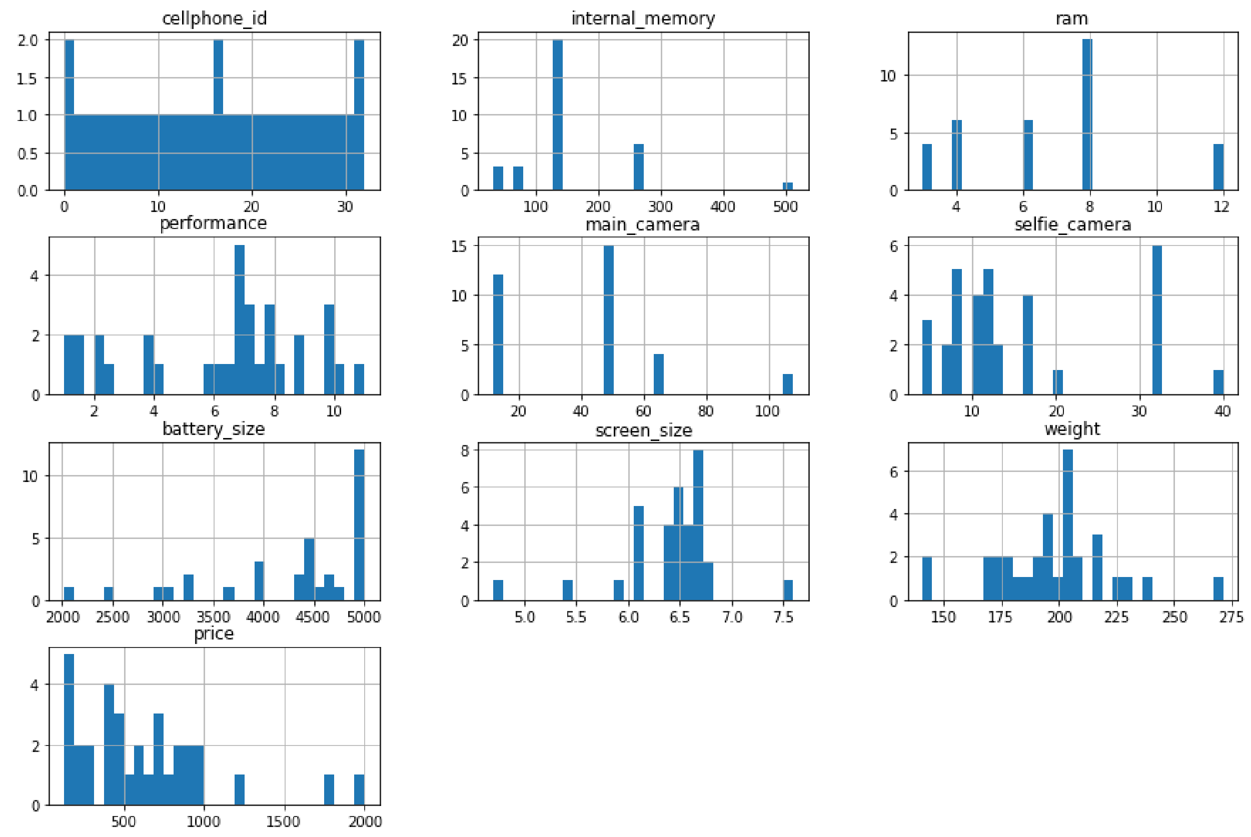
Code:

In [7]:

```
# For any numerical feature in the dataframe, plot a histogram to show its di
stribution.
# We found out the numerical data are ['internal_memory', 'ram', 'performance
', 'main_camera', 'selfie_camera', 'battery_size', 'screen_size', 'weight', '
price', 'release_date']
# df.hist() will automatically show us all the numerical data into histograms
leaving out the categorical ones from the original Dataframe (df)

def plot_hist(df):
  return df

plot_hist(df).hist(bins=30, figsize=(15, 10))
```



End of Code

Explanation:

Used to show the distribution of a numeric variable. By calling the hist() method on the data frame and passing in the appropriate parameters, a histogram can be generated for any numeric function in the data frame. This allows you to quickly and easily identify patterns and trends in your data, e.g. For example, whether the data is skewed or has a normal distribution. It can also give you a general idea of the range of values for each numeric characteristic and how often each value occurs in the dataset. This can help you identify outliers and make data preprocessing decisions for machine learning model fitting.

How to Run VS Code Server with Google Colab

Step 1:

Reference - https://plainenglish.io/blog/how-to-run-vs-code-server-with-google-colab-2f902d8fa4f5
Run the following in Google Colab.

Code:

In [8]:
```
%pip install colabcode
```

End of Code

Step 2:

Code:

In [9]:
```
from colabcode import ColabCode
```

End of Code

Step 3:

Code:

In [10]:
```
!wget https://bin.equinox.io/c/bNyj1mQVY4c/ngrok-v3-stable-linux-amd64.tgz
!tar -xvzf ngrok-v3-stable-linux-amd64.tgz
!ngrok authtoken 2Jy4tWV8rV7nNJiRc6q9yhzJB7C_2RJNyB1V8KAEQ26e3aDVG
# Replace YOUR_AUTH_TOKEN with the auth token generated by ngrok.
```

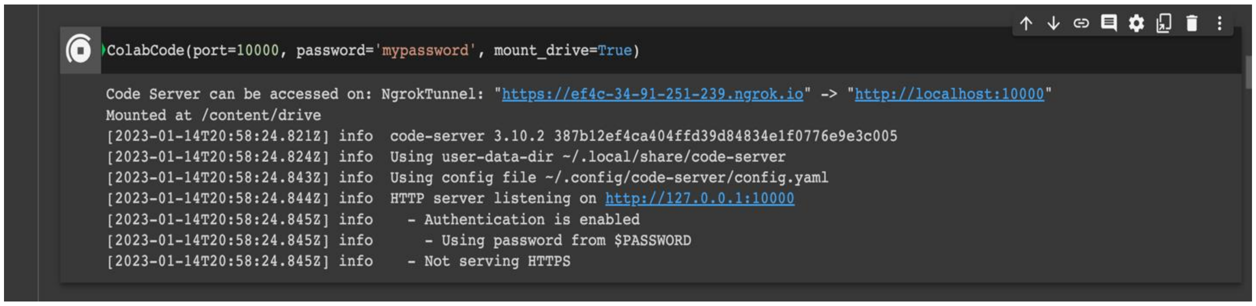
End of Code

Step 4:

Code:

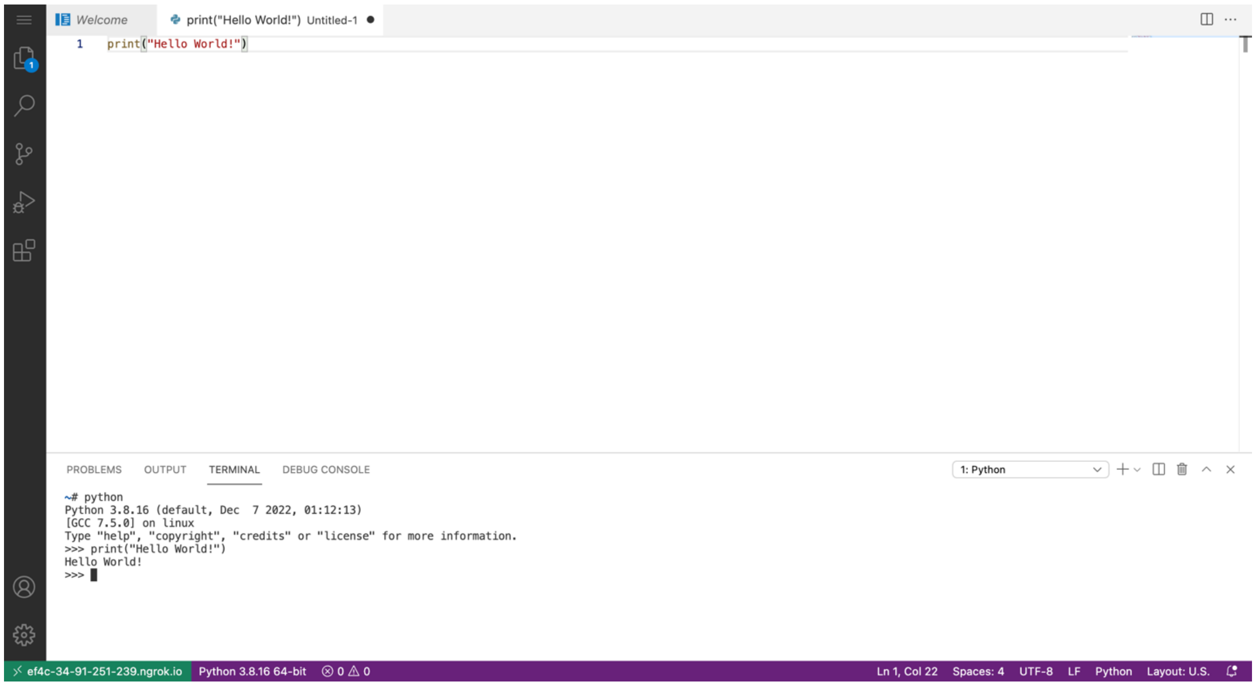
In [11]:
```
ColabCode(port=10000, password='mypassword', mount_drive=True)
```

End of Code



```
ColabCode(port=10000, password='mypassword', mount_drive=True)

Code Server can be accessed on: NgrokTunnel: "https://ef4c-34-91-251-239.ngrok.io" -> "http://localhost:10000"
Mounted at /content/drive
[2023-01-14T20:58:24.821Z] info  code-server 3.10.2 387b12ef4ca404ffd39d84834e1f0776e9e3c005
[2023-01-14T20:58:24.824Z] info  Using user-data-dir ~/.local/share/code-server
[2023-01-14T20:58:24.843Z] info  Using config file ~/.config/code-server/config.yaml
[2023-01-14T20:58:24.844Z] info  HTTP server listening on http://127.0.0.1:10000
[2023-01-14T20:58:24.845Z] info    - Authentication is enabled
[2023-01-14T20:58:24.845Z] info      - Using password from $PASSWORD
[2023-01-14T20:58:24.845Z] info    - Not serving HTTPS
```



```python
print("Hello World!")
```

```
~# python
Python 3.8.16 (default, Dec  7 2022, 01:12:13)
[GCC 7.5.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World!")
Hello World!
>>> 
```

House Prices Dataset

Loading the Datasets:

Code:

In [12]:

```
# Loading the datasets.
train = pd.read_csv('train-2.csv')
test = pd.read_csv('test_data_with_target-1.csv')
```

End of Code

Explanation:

This process is known as loading records. To do this, the pandas read_csv() library function is used to read data from the CSV files into a pandas dataframe. In this way, the data from the CSV files are loaded into the program and can be used for further processing, cleaning, and analysis. The reason is to have the data available in the program and to be able to manipulate it.

Dataset Counter:

Code:

In [13]:

```
# Return the number of attributes and the number of data points in the traini
ng data.
def dataset_counter(df):
  attributes = df.shape[1]
  data_points = df.shape[0]
  return (attributes, data_points)
print(dataset_counter(train))
print(dataset_counter(test))
```

End of Code

Explanation:

The dataset_counter function is employed to ascertain the amount of features and the quantity of data points in the input dataframe, df. The shape property of a dataframe yields the number of rows and columns in the dataframe. shape[1] is used to acquire the number of columns, which stands for the number of features in the dataframe. The shape[0] is used to secure the number of rows, which designates the number of data points in the dataframe. The values yielded by the function are stored in the attributes and data_points variables respectively. The function then produces these values as a tuple.

This is significant because it will provide a perception about the magnitude of the dataset. This can be a vital factor to take into account while selecting a machine learning algorithm, and also the larger the dataset, the more computational resources it will necessitate to train a model.

In this situation, the function is applied to both the training and test dataframes, and the number of features and data points in each is established.

Feature Correlation Analysis:

Code:

In [14]:

```
# Finding Out which features are highly correlated and which ones are weakly correlated
numerical_cols1 = train.select_dtypes(include=[np.number])
numerical_cols2 = test.select_dtypes(include=[np.number])

corr1 = numerical_cols1.corr()
corr2 = numerical_cols2.corr()

print('The Most Correlated Features For Train Dataset with SalePrice:')
print(corr1['SalePrice'].sort_values(ascending=False)[:10])

print('The Most Uncorrelated Features For Test Dataset with SalePrice:')
print(corr1['SalePrice'].sort_values(ascending=False)[-10:])

print('The Most Correlated Features with For Train Dataset SalePrice:')
print(corr2['SalePrice'].sort_values(ascending=False)[:10])

print('The Most Uncorrelated Features with For Test Dataset SalePrice:')
print(corr2['SalePrice'].sort_values(ascending=False)[-10:])
```

End of Code

Explanation:

This process is called feature correlation analysis. The purpose of this process is to identify the relationship between different characteristics of the data set and their impact on the target variable (in this case, SalePrice). By finding the most correlated and uncorrelated features of SalePrice, you can better understand which features are more important in predicting SalePrice and which features may not be useful for the model. This information can be used to make informed decisions about which features to include or exclude in your model, and can help identify multiline issues in your dataset.

Pre Processing:

Code:

In [15]:

```
def pre_process(df):
# Add all preprocessing required in the dataset.
# HINT - You might have to process categorical and numerical variables in dif
ferent ways, is there any automated way to find
# list of features which are categorical or numerical?



  categorical_features = df.select_dtypes(include=['object']).columns
  numerical_features = df.select_dtypes(exclude=['object']).columns

  # One-hot encode categorical columns
  df = pd.get_dummies(df, columns=df.select_dtypes(include=['object']).column
s)
  return df

train1 = pre_process(train)
print(train1.shape)
```

In [16]:

```
test1 = pre_process(test)
print(test1.shape)
```

End of Code

Explanation:

The pre_process procedure is used to get the input dataframe ready for further analysis or modeling. It carries out a few preprocessing procedures on the input dataframe, df.

To start, the select_dtypes() system is used to recognize the categorical and numerical features inside the dataframe. The include parameter is used to pick columns of data type 'object' and the exclude parameter is used to pick columns of other data types. The chosen columns are then held in the categorical_features and numerical_features variables individually.

The next step is to one-hot encode the category features. One-hot coding is a process of changing over categorical variables into a numerical form that can be comprehended by a machine learning model. In this case, the pd.get_dummies() technique is employed to do one-hot encoding on the category features. The columns parameter is utilized to indicate which columns should be one-hot encoded.

Missing Data:

Code:

In [17]:

```
def missing_data_fix(df):
# Fix how to handle missing data.

# Replacing all numerical NaN with a mean of all other numerical datapoints
  df.fillna(df.mean(), inplace=True)

# Replacing all categorical with NaN with a mode of all other categorical

datapoints
  for col in df.select_dtypes(include=['object']):
    df[col].fillna(df[col].mode()[0], inplace=True)
  return df
```

End of Code

Explanation:

The missing_data_fix function is for dealing with any gaps in the input dataframe, df.

The procedure starts with replacing any numerical NaN values with the average of all the numerical data points. This is done by using the fillna() method, which substitutes any absent values with a specified amount. In this example, the mean of all numerical data points is used for the replacement value. The inplace parameter is set to True so that the changes are directly made to the input dataframe.

The process then continues to change all categorical NaN values with the most common of all the categorical data points. This is done by going through all the columns in the dataframe that have a data type of 'object' with the select_dtypes() method. The fillna() method is then utilized to substitute any missing values in each column with the mode of that column.

Lastly, the pre-processed dataframe is sent back for further investigation or modeling.

Feature Alignment:

Code:

In [20]:

```
# Find the intersection of the columns in train1 and test1
common_cols = set(train1.columns).intersection(set(test1.columns))

# Remove columns that are not in the intersection
train1 = train1.loc[:, common_cols]
test1 = test1.loc[:, common_cols]
```

End of Code

Explanation:

The code is designed to identify the shared columns between the train1 and test1 dataframes, then eliminate the attributes that are not found in the overlap.

The process starts by finding the intersection of the columns in train1 and test1 with the intersection() method. This will generate a group of columns that are found in both dataframes.

The next step is to discard the columns that are not in the intersection. This is performed by using the loc[] accessor to pick only the columns located in the intersection. The dataframe is then reassigned to itself, only containing the columns in the intersection.

This code ensures that the test dataset only contains elements that the model has witnessed during training, thus stopping errors in the prediction. This is done by eliminating columns that are not in the intersection of the training and test datasets.

Training a Linear Regression Model:

Code:

In [23]:

```
def train_LR(df):
  X = df.drop('SalePrice', axis=1)
  y = df['SalePrice']
  LR = LinearRegression()
  LR.fit(X, y)
  return LR

model = train_LR(train1)
```

End of Code

Explanation:

The train_LR function is employed to work out a linear regression model on the given dataframe, df. This function divides the dataset into two sets: the feature set (X) and the target variable (y). The feature set is produced by excluding the 'SalePrice' column from the dataframe, which is employed as the target variable.

A Linear Regression model is then formed by constructing an instance of the LinearRegression class from the scikit-learn library. The fit() method is then activated on the model instance, with the feature set X and target variable y as the inputs. The model is then trained on this data and returned.

The train1 dataframe is handed over as the input to the train_LR function and the train_LR function is carried out and the resulting model is stored in the model variable. Linear Regression is a straightforward and easy-to-utilize algorithm that is advantageous for comprehension the relationship between a dependent variable and one or more independent variables.

Model Evaluation:

Code:

In [24]:

```
from sklearn.metrics import r2_score
def test_LR(df):
# Calulate metrics on the training dataset.
  X = df.drop('SalePrice', axis=1)
  y = df['SalePrice']
  LR = LinearRegression()
  LR.fit(X, y)
  y_pred = LR.predict(X)
  rmse = np.sqrt(mean_squared_error(y, y_pred))
  print("Root Mean Squared Error: ", rmse)
  r2 = r2_score(y, y_pred)
  print("R-squared: ", r2)
```

In [25]:

```
test_LR(train1)
```

```
Root Mean Squared Error:  22851.50521615243
R-squared:  0.9172017131100771
```

In [26]:

```
test_LR(test1)
```

```
Root Mean Squared Error:  11794.50828642643
R-squared:  0.9761939720864488
```

End of Code

Explanation:

The test_LR function assesses the efficiency of a Linear Regression model on a given dataframe, df. To make this evaluation, the dataframe is separated into two components: the feature set (X) and the target variable (y). To form the feature set, the 'SalePrice' column is deleted from the dataframe, leaving the target variable.

The LinearRegression class from the scikit-learn library is then accessed to instantiate a Linear Regression model. Its fit() method is implemented, taking the feature set X and the target variable y as its inputs. The model is trained with this data.

Once the training is complete, the predictions are made on the same dataset, storing them in y_pred, by calling LR.predict(X).

Finally, two metrics are calculated with the scikit-learn library: Root Mean Squared Error (RMSE) and R-squared. The Root Mean Squared Error (RMSE) is a way of measuring the average discrepancy between the predicted and actual values, which is computed as the square root of the mean of the squared differences between the expected and true values. The R-squared score is a measure of the portion of the variation in the outcome variable that can be clarified by the set of features. It is recorded as a rate between 0 and 1, with 1 suggesting a perfect fit.

The RMSE is calculated by np.sqrt(mean_squared_error(y, y_pred)) and the R-squared score is calculated by r2_score(y, y_pred). These metrics are printed to the console, allowing the user to evaluate the model's performance.

This is significant because it allows the user to assess the performance of the model on the input data and decide if it is working well or if it needs to be enhanced.
RMSE is a widely-used metric for regression issues, and it gauges the disparity between the predicted and real values. It is a calculation of the average mistake of the model - a minor RMSE value implies a better fit. R-squared is an indicator of the amount of variance in the objective variable that can be explained by the feature set - a greater R-squared value shows a better fit.

In this instance, the test_LR function is used on the train1 training dataframe, and the resulting metrics are printed on the console. This allows the user to examine the model's performance on the training data and decide if it is overfitting or underfitting. After the model is trained on the training dataset, it is necessary to evaluate it on unseen data to make sure it generalizes effectively.

Model Tuning:

Code:

In [27]:

```
from sklearn.linear_model import LassoCV
def train_LR_reg(df):
  X = df.drop('SalePrice', axis=1)
  y = df['SalePrice']
  model = LassoCV(max_iter=10000, tol=0.0001)
  model.fit(X, y)
  return model

train_LR_reg(train1)
```

Out[27]:

```
LassoCV(max_iter=10000)
```

End of Code

Explanation:

The train_LR_reg function is used to generate a LassoCV (Lasso regularization and Cross-Validation) model based on the df dataframe. This split the dataframe into two component parts: the X features and the y target variable. The feature set is created by excluding the 'SalePrice' column from the dataframe, which is employed as the target.

A LassoCV model is then generated by making an instance of the LassoCV class from the sklearn.linear_model library. The fit() method is then activated on the model instance, using the X feature set and the y target variable as the inputs. This data is then used to train the model, which is finally returned.

LassoCV is a specific variation of Lasso Regularization, which is a method of simplifying a model by making some of the coefficients equal to zero. It is a linear approach that searches for sparse coefficients. This version of Lasso regression has an integrated cross-validation system. Cross-validation is used to find the best value for the regularization parameter.

- Why LassoCV and not Lasso?

As opposed to its predecessor, the Lasso algorithm, the LassoCV algorithm includes a cross-validation function. Cross-validation is a method of assessing the efficacy of a model by dividing the data into two separate sets: one for training and the other for validation. This process aids in finding the ideal value of the regularization parameter. The LassoCV algorithm uses cross-validation to identify the perfect regularization parameter, which is the variable that governs the amount of regularization in the model. This is essential since the regularization parameter determines the balance between the model's fit and complexity. Locating the optimal value of the regularization parameter is essential for ideal model performance.

Feature Importance Analysis:

Code:

In [28]:

```
def find_imp_features(model, X):
# print the 10 most important attributes/features that seem to have a big imp
act on the prediction of the sale price.
# https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Line
arRegression.html
# The coef_ contain the coefficients for the prediction of each of the target
s

  feature_imp = abs(model.coef_)
  feature_imp = 100.0 * (feature_imp / feature_imp.max())
  sorted_idx = np.argsort(feature_imp)
  important_features = X.columns[sorted_idx[-10:]]
  print("10 Most Important Features: ", important_features)

find_imp_features(model, train1)
```

Out[28]:

```
10 Most Important Features:  Index(['Condition1_Feedr', 'RoofStyle_Shed', 'Co
ndition1_PosA',
       'Exterior1st_AsbShng', 'GarageQual_TA', 'CentralAir_Y',
       'LotConfig_CulDSac', 'LandContour_Bnk', 'RoofMatl_WdShngl',
       'BsmtCond_TA'],
      dtype='object')
```

End of Code

Explanation:

The function find_imp_features takes the trained model and the feature set (X) as inputs, and it provides an output of the 10 features that have the greatest influence on the prediction of the sale price. It does this by measuring the feature importance values, which are obtained by taking the absolute value of the model's coefficients and normalizing them by dividing by the maximum coefficient value. Then, it orders the indices of the features according to their importance and chooses the 10 most vital ones by taking the last 10 elements of the arranged indices.

It is significant since it helps to comprehend which features have the greatest impact on the target variable. Additionally, it can be employed to enhance the interpretability of the model by reducing the number of features while still maintaining a good accuracy. It can also be used to improve the performance of the model by removing irrelevant or superfluous features.

Importance Visualization:

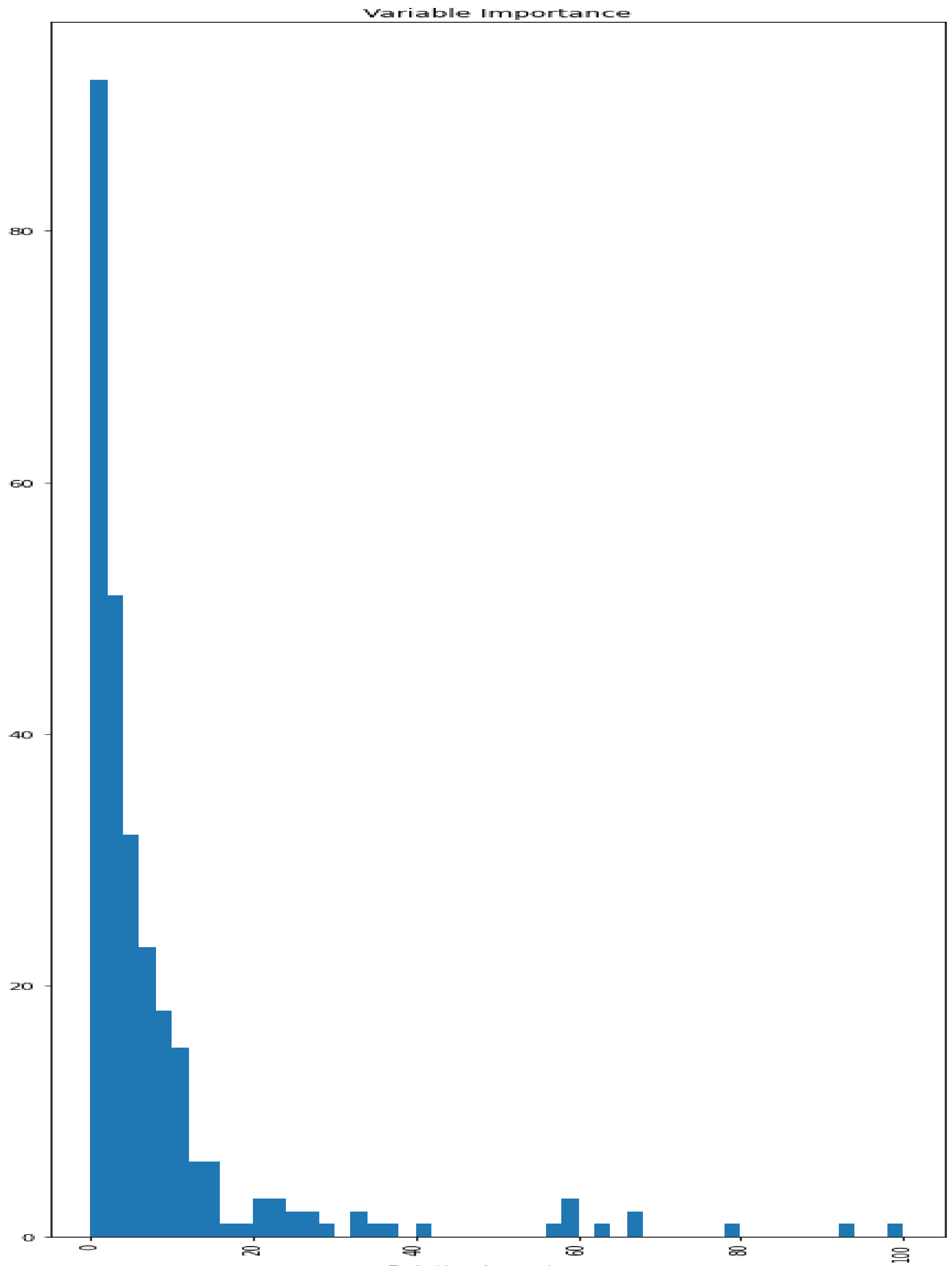
Code:

In [29]:

```
def plot_weights(model):
    feature_imp = abs(model.coef_)
    feature_imp = 100.0 * (feature_imp / feature_imp.max())
    sorted_idx = np.argsort(feature_imp)
    plt.figure(figsize=(16, 24))
    plt.subplot(1, 2, 2)
    plt.hist(feature_imp[sorted_idx], bins=50)
    plt.xticks(rotation=90)
    plt.xlabel('Relative Importance')
    plt.title('Variable Importance')
    plt.show()

plot_weights(model)
```

End of Code


Variable Importance
80
60
40
20
0
0
20
40
60
80
100
Relative Importance

Explanation:

The plot_weights function takes a model that has already been trained, and it graphs a histogram of the feature significance values. It begins by computing the importance values of the features by obtaining the absolute value of the model's coefficients and then dividing by the highest coefficient amount. Then, it orders the indices of the features according to their importance values and depicts a histogram of the feature importance values.

It is paramount as it aids in comprehending which features have the most effect on the target variable. It can be utilized to make the interpretability of the model more straightforward by cutting down the number of features while maintaining a good accuracy. It can also be utilized to enhance the performance of the model by eliminating unnecessary or redundant features. Furthermore, it can help to detect outliers and other trends in the data.

Feature Alignment:

Code:

In [30]:

```
# Check the number of features in the Test and Model
print("Test features:", test1.shape[1])

# Check the number of features expected by the model
print("Model expected number of features:", model.coef_.shape[0])
```

```
Test features: 272
Model expected number of features: 271
```

In [31]:

```
test1 = test1.drop(columns=["SalePrice"])
```

End of Code

Explanation:

This process is called feature alignment or feature matching. The goal of this process is to verify that the number of features in the test dataset matches the number of features expected by the model.
In this case, make sure the number of features in the test dataset (272) matches the number of features expected by the model (271). The number of features in the test dataset is higher than the model expects, so remove the SalePrice column from the test dataset. As a result, the number of features in the test data set matches the number of features expected by the model, so the model can make accurate predictions on the test data set. Performing feature alignment is important because the model expects as many features in the test dataset as there are features in the training dataset. Otherwise, the prediction may fail.
Removing the SalePrice column from the test dataset because the model is trained to predict the SalePrice using the training dataset's features but not the SalePrice itself.

Model Evaluation:

Code:

In [32]:

```
test_proc = test1
test = test

def acc_tester(model, test, test_proc):



#'''
#Arguments:
#model - The traing linear regression model.
#test - The test dataset.
#test_proc - The pre-processed test dataset.
#Be sure that the test dataset is preprocessed similar to the train dataset.
#Returns:
#Result of test
#'''

# Taking 20 rows at random from the test dataset.
  y_pred = model.predict(test_proc[0:20]) # Getting predictions
  if np.sqrt(mean_squared_error(test['SalePrice'].to_numpy()[0:20], y_pred))
< 50000:
    return "Test case passed!"
  else:
    return "Please retry, your model may need corrections!"

acc_tester(model, test, test_proc)
```

Out[32]:

```
'Test case passed!'
```

End of Code

Explanation:

This process is called evaluating or testing the model. The purpose of this process is to evaluate the performance of the trained model using invisible data. Here you pass the trained model, the original test data set, and the data set from the preprocessed test to the acc_tester function. Within the function, you select a random sample of 20 rows from a pre-processed test data set and feed them into the model for predictions. Then you compare the predicted values to the actual SalePrice values from the original test data set and calculate the root mean square error (RMSE) between them.If the RMSE is less than 50,000, the test case is passed and the function returns "Test case passed!" return. If not, the function returns "Try again, the model may need to be changed!".It is important to evaluate the model against invisible data because it allows you to see how well the model responds to new data generalized and how accurate it is . This test is just an example, it is important to test the model on multiple data points to ensure its performance.