**Import pandas as pd**

**Import numpy as np**

**Import seaborn as sns**

**Import matplotlib.pyplot as plt**

**Import warnings**

**%matplotlib inline**

**Warnings.filterwarnings('ignore')**

Pandas - used to perform data manipulation and analysis

Numpy - used to perform a wide variety of mathematical operations on arrays

Matplotlib - used for data visualization and graphical plotting

Seaborn - built on top of matplotlib with similar functionalities

%matplotlib - to enable the inline plotting.

Warnings - to manipulate warnings details

Filterwarnings('ignore') is to ignore the warnings thrown by the modules (gives clean results)

**Df = pd.read\_csv(r"c:\users\hp\downloads\boston dataset.csv")**

**Df.drop(columns=['unnamed: 0'], axis=0, inplace=true)**

**Df.head()**

1. `pd.read\_csv(r"c:\users\hp\downloads\boston dataset.csv")`: this part reads a csv file named "boston dataset.csv" located in the folder "c:\users\hp\downloads\". The data from the csv file is loaded into a special table-like structure called a dataframe.

2. `df.drop(columns=['unnamed: 0'], axis=0, inplace=true)`: the dataframe `df` has a column named 'unnamed: 0', and this line of code is designed to remove that column. However, there is an error in the code: the `axis` parameter is set to 0, which means it will try to drop rows instead of columns. To fix the code and drop the column correctly, you should set `axis=1`.

The correct version would be: `df.drop(columns=['unnamed: 0'], axis=1, inplace=true)`

3. `df.head()`: after dropping the 'unnamed: 0' column from the dataframe, this line shows the first few rows of the modified dataframe using the `head()` function. This is a quick way to check what the dataframe looks like after the changes.

To summarize, the code reads a csv file, creates a dataframe, removes the 'unnamed: 0' column, and then displays the first few rows of the updated dataframe.

**# statistical info**

**Df.describe()**

The df.describe() function is used to get statistical information about the dataframe df. It calculates and summarizes various statistical measures for each numerical column in the dataframe

**# datatype info**

**Df.info()**

The df.info() function provides concise information about the dataframe df, including the number of non-null values and the data types of each column.

**Df.isnull().sum()**

Df.isnull().sum() returns a series (a one-dimensional labeled array) that shows the count of missing values for each column in the dataframe. This information is useful for identifying columns that have missing data and helps you decide how to handle those missing values, such as filling them with appropriate values or removing the rows or columns with missing data altogether.

**# create box plots**

**Fig, ax = plt.subplots(ncols=7, nrows=2, figsize=(20, 10))**

**Index = 0**

**Ax = ax.flatten()**

**For col, value in df.items():**

**sns.boxplot(y=col, data=df, ax=ax[index])**

**index += 1**

**Plt.tight\_layout(pad=0.5, w\_pad=0.7, h\_pad=5.0)**

The code generates a grid of box plots, where each box plot represents the distribution of data in a numerical column of the dataframe df. The box plot provides information about the median, quartiles, and any potential outliers in each column's data distribution. The 2x7 grid allows you to visualize multiple columns' data distributions side by side, making it easier to compare them.

**# create dist plot**

**Fig, ax = plt.subplots(ncols=7, nrows=2, figsize=(20, 10))**

**Index = 0**

**Ax = ax.flatten()**

**For col, value in df.items():**

**sns.histplot(value, ax=ax[index])**

**index += 1**

**Plt.tight\_layout(pad=0.5, w\_pad=0.7, h\_pad=5.0)**

The code generates a grid of histograms, where each histogram represents the distribution of data in a numerical column of the dataframe df. Histograms are used to visualize how data is distributed across different values or bins. The 2x7 grid allows you to visualize multiple columns' data distributions side by side, making it easier to compare them. It helps you understand the spread and frequency of data values within each column and gives insights into the data's distribution.

**Cols = ['crim', 'zn', 'tax', 'black']**

**For col in cols:**

**# find minimum and maximum of that column**

**minimum = min(df[col])**

**maximum = max(df[col])**

**df[col] = (df[col] - minimum) / (maximum - minimum)**

Min-max normalization is used to transform the values of each column so that they are all on the same scale, ranging from 0 to 1. This normalization is useful when you want to compare or analyze data with different scales or ranges, making sure that no column dominates the analysis due to its larger values. It brings all the columns to a common scale and preserves the relative relationships **between the data points in each column.**

**Fig, ax = plt.subplots(ncols=7, nrows=2, figsize=(20, 10))**

**Index = 0**

**Ax = ax.flatten()**

**For col, value in df.items():**

**sns.histplot(value, ax=ax[index])**

**index += 1**

**Plt.tight\_layout(pad=0.5, w\_pad=0.7, h\_pad=5.0)**

The code generates a 2x7 grid of histogram plots, where each histogram represents the distribution of data in a numerical column of the dataframe df. Histograms are used to visualize how data is distributed across different values or bins. The 2x7 grid allows you to visualize multiple columns' data distributions side by side, making it easier to compare them. It helps you understand the spread and frequency of data values within each column and gives insights into the data's distribution. The sns.histplot() function from seaborn library is used to create these histogram plots.

**# standardization**

**From sklearn import preprocessing**

**Scalar = preprocessing.standardscaler()**

**# fit our data**

**Scaled\_cols = scalar.fit\_transform(df[cols])**

**Scaled\_cols = pd.dataframe(scaled\_cols, columns=cols)**

**Scaled\_cols.head()**

Standardization scales the data so that each feature (column) has a mean of 0 and a standard deviation of 1. It is useful when working with algorithms that assume the input data to be standardized or when the features have different measurement scales. Standardization helps to bring all the columns to a common scale and prevents any single feature from dominating the analysis based on its magnitude. It ensures that all the features contribute equally to the model training and helps improve the performance of certain algorithms that are sensitive to feature scales

**For col in cols:**

**df[col] = scaled\_cols[col]**

The code is copying the standardized values from scaled\_cols back into the original dataframe df. This step ensures that the columns in df are now standardized, meaning they have a mean of 0 and a standard deviation of 1, which is useful for various data analysis and modeling purposes. The original data in df has been replaced with its standardized version in the specified columns (cols).

**Fig, ax = plt.subplots(ncols=7, nrows=2, figsize=(20, 10))**

**Index = 0**

**Ax = ax.flatten()**

**For col, value in df.items():**

**sns.histplot(value, ax=ax[index])**

**index += 1**

**Plt.tight\_layout(pad=0.5, w\_pad=0.7, h\_pad=5.0)**

The code generates a 2x7 grid of histogram plots, where each histogram represents the distribution of data in a numerical column of the dataframe df. Histograms are used to visualize how data is distributed across different values or bins. The 2x7 grid allows you to visualize multiple columns' data distributions side by side, making it easier to compare them. It helps you understand the spread and frequency of data values within each column and gives insights into the data's distribution. The sns.histplot() function from seaborn library is used to create these histogram plots.

**Corr = df.corr()**

**Plt.figure(figsize=(20,10))**

**Sns.heatmap(corr, annot=true, cmap='rainbow')**

The code calculates the correlation between all numerical columns in the dataframe df and creates a heatmap to visualize these correlation values. The heatmap helps you quickly identify patterns and relationships between different variables in the data. Brighter colors represent stronger positive correlations, darker colors represent stronger negative correlations, and the absence of color (white cells) indicates no significant correlation. The heatmap is a useful tool for identifying which variables are positively or negatively associated with each other, providing insights into the relationships within your data.

**Sns.regplot(y=df['medv'], x=df['lstat'])**

Here, the price of houses decreases with the increase in the 'lstat'. Hence it is negatively correlated.

**Sns.regplot(y=df['medv'], x=df['rm'])**

Here, the prices of houses increase with the increase in 'rm'. Hence it is positively correlated.

Input split

Let us split the data for training and testing.

**X = df.drop(columns=['medv', 'rad'], axis=1)**

**Y = df['medv']**

Model training

Now let's import functions to train models.

Instead of training the whole model, we will split the dataset for estimating the model performance.

If you train and test the dataset completely, the results will be inaccurate. Hence, we

Will use 'train\_test\_split'.

We will add random\_state with the attribute 42 to get the same split upon re-running.

If you don't specify a random state, it will randomly split the data upon re-running giving inconsistent results

**From sklearn.model\_selection import cross\_val\_score, train\_test\_split**

**From sklearn.metrics import mean\_squared\_error**

**Def train(model, x, y):**

**# train the model**

**x\_train, x\_test, y\_train, y\_test = train\_test\_split(x, y, random\_state=42)**

**model.fit(x\_train, y\_train)**

**# predict the training set**

**pred = model.predict(x\_test)**

**# perform cross-validation**

**cv\_score = cross\_val\_score(model, x, y, scoring='neg\_mean\_squared\_error', cv=5)**

**cv\_score = np.abs(np.mean(cv\_score))**

**print("model report")**

**print("mse:",mean\_squared\_error(y\_test, pred))**

**print('cv score:', cv\_score)**

X contains input attributes and y contains the output attribute.

We use 'cross val score' for better validation of the model.

Here, cv=5 means that the cross-validation will split the data into 5 parts.

Np.abs will convert the negative score to positive and np.mean will give the average value of 5 scores.

Linear regression:

**From sklearn.linear\_model import linearregression**

**Model = linearregression(normalize=true)**

**Train(model, x, y)**

**Coef = pd.series(model.coef\_, x.columns).sort\_values()**

**Coef.plot(kind='bar', title='model coefficients')**

Mean squared error is around 23 and cross-validation score is around 35.

Since normalization is important for basic models like linear regression, we can state it as normalize=true.

Rm shows a high positive coefficient and nox shows a high negative coefficient.

Decision tree:

**From sklearn.tree import decisiontreeregressor**

**Model = decisiontreeregressor()**

**Train(model, x, y)**

**Coef = pd.series(model.feature\_importances\_, x.columns).sort\_values(ascending=false)**

**Coef.plot(kind='bar', title='feature importance')**

Here cv score is higher than linear regression.

Random forest:

**From sklearn.ensemble import randomforestregressor**

**Model = randomforestregressor()**

**Train(model, x, y)**

**Coef = pd.series(model.feature\_importances\_, x.columns).sort\_values(ascending=false)**

**Coef.plot(kind='bar', title='feature importance')**

Mse is around 10 and cv score is around 21.

We know that 'rm' and 'lstat' produces much correlation with the target variable. That is the reason behind its higher feature importance among other attributes.

**Extra trees:**

**From sklearn.ensemble import extratreesregressor**

**Model = extratreesregressor()**

**Train(model, x, y)**

**Coef = pd.series(model.feature\_importances\_, x.columns).sort\_values(ascending=false)**

**Coef.plot(kind='bar', title='feature importance')**

Here mse is similar to random forrest and the cross-validation score is decreased

Final thoughts

You can further improve the model by creating new attributes and performing hyperparameter tuning.

You can create a new categorical attribute with the help of an existing numerical attributes.

In this article, we discussed the dataset for boston house price prediction. We understood the difference between over-fitting and under-fitting and examined the methods to achieve the good-fit model.