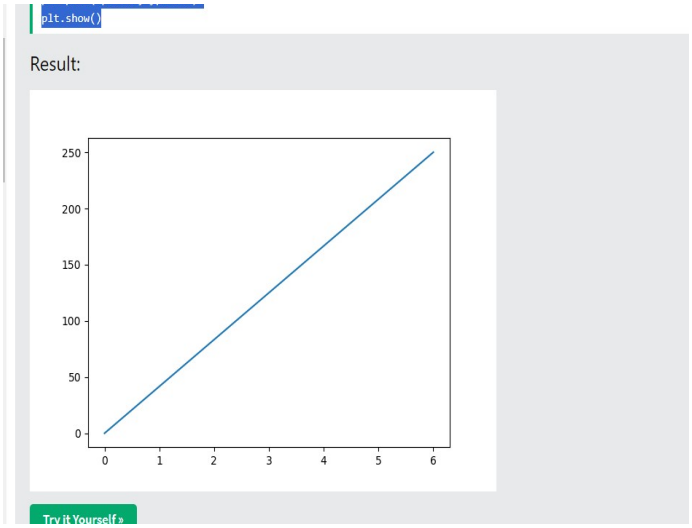
Project housing real estate code

1)import pandas as pd
Now the Pandas package can be referred to as `pd` instead of `pandas`
Pandas DataFrame is a 2 dimensional data structure, like a 2 dimensional array, or a table with rows and columns
`df = pd.DataFrame(data)`

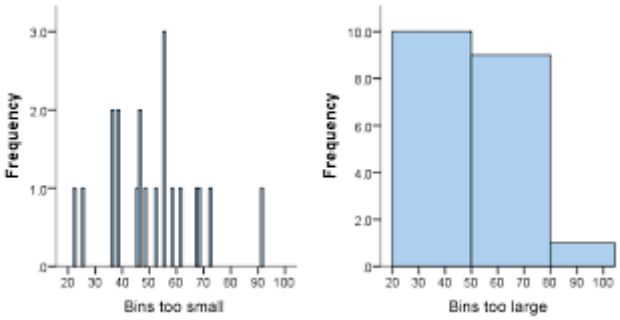
2) hosuing =pd.read_csv("data.csv")
here housing is a dataframe

3)housing.head()
now dataframe named housing has data of data.csv and .head will show a quick glance at first 5 rows of this data
4)housing.info()
shows all the information about data like how many entries are there this data set we are using has 506 enteries
5) housing['CHAS'].value_counts()
tells in 471 rows data is 0
and in 35 rows data is 1
note chas is in capital
6)housing.describe()
told us min mean max std counts 25% 50% means 25% values are lower than given data in table and all in data.csv
7)import matplotlib.pyplot as plt
imports pyblot

```
import matplotlib.pyplot as plt
import numpy as np

xpoints = np.array([0, 6])  //shows x asix  o to 6
ypoints = np.array([0, 250])  //shows y axis 0 to 250

plt.plot(xpoints, ypoints)
plt.show()
```



6)%matplotlib inline
This command is used to enable the inline rendering of matplotlib plots directly in the notebook interface.
9)housing.hist(bins=50,figsize(20,15))
get plots of every term like age crm b lstat and all
`housing.hist`: This part of the code suggests that you have a variable or DataFrame named `housing`
`bins=50`: This argument specifies the number of bins (or bars) that the histogram will be divided into. Bins represent intervals or ranges of data values. In this case, you're specifying 50 bins, meaning the data will be divided into 50 intervals for the histogram.



`figsize=(20, 15)`: This argument sets the size of the figure (the entire plot) in which the histogram will be displayed. It's a tuple representing the width and height of the figure in inches. In this case, the figure will be 20 inches wide and 15 inches tall.

Now we will do train test splitting
evaluate the performance of a model on unseen data. The process involves dividing a dataset into two distinct subsets: a training set and a testing (or validation) set. The main purpose of this splitting is to assess how well a trained model generalizes to new, unseen data

1)import numpy as np

2 def split_train_test(data,test_ratio):
created a function to split test and train the data test ratio will be like 20% to 50%
because we want to test data so we wont give majority of data so its value will be llike 0.2 or 0.3 somthing

2.5) np.random.seed(42)
explained neeche ignore for now

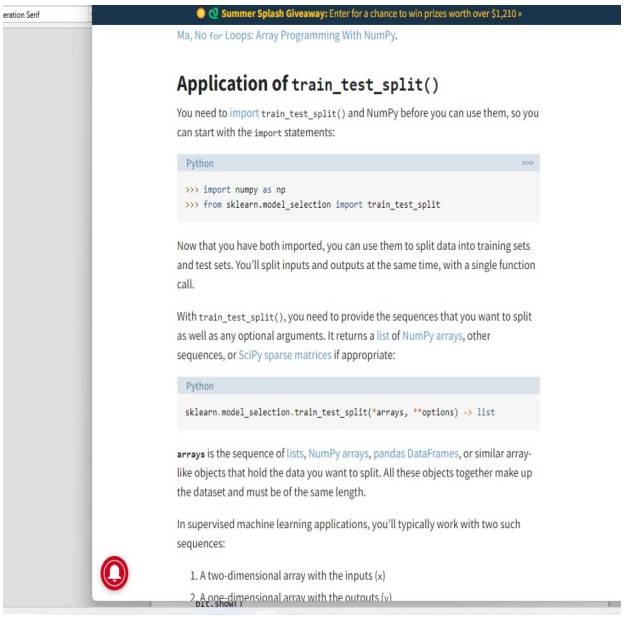
3)shuffled =np.random.permutation(len(data))
np.random.permutation(len(6)) lets say it was this output will be anything from 0 to 5
random and permutation are functions of numpy library and shuffled is a variable we defined here

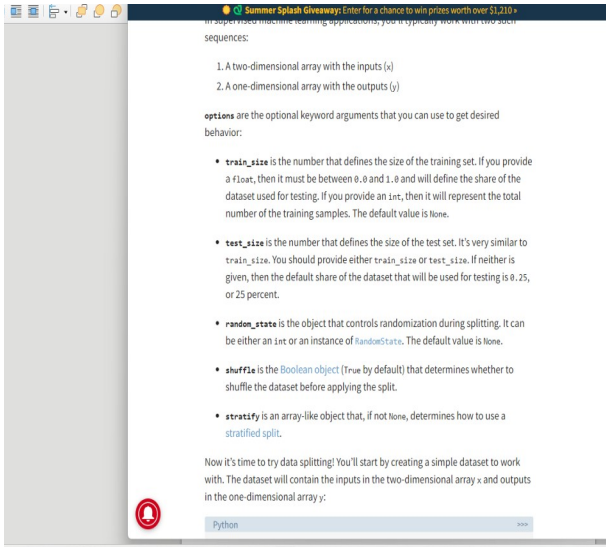
4)test_set_size=int(len(data) * test_ratio)
setting test size like 506 * test ratio which will be anything from 0 to 1

5)test_indices= shuffled [:test_set_size]
6)train_indices= shuffled [test_set_size:0]

lets say test_set_size comes to be 208 out of 506
so test indices will shuffled data from 0 to 208
while model will train data from 208 to remaining 506
indices in short term is like index

7)return data.iloc[train_indices], data.iloc[test_indicies]
when you combine `data.iloc`, you are using integer-based indexing to select specific rows and columns from the DataFrame or Series. Here's how you might use it:





8)train_set,test_set= split_train_test(housing,0.2)
calling out split train test function with housing vairiable which we happened to define as dataframe for this project containing data from data.csv and0.2 test ratio

9)print(len(train_set),len(test(set))
we get values 405 101 as output
if we go back to this code
shuffled =np.random.permutation(len(data))

and print (shuffled) it will give us random values everytime
and data in train can end up in test set or vice versa and model can overfit patterns
and that would be bad practice
to fix this we add np.random.seed(42)

`p.random.seed(42)` is used in the NumPy library to set the random seed for generating random numbers. This allows you to reproduce the same sequence of random numbers every time you run your code with the same seed value. It's often used in situations where you want to ensure reproducibility of your results, especially in scientific experiments or data analysis where randomness is involved.

All we did we could have directly done with sklearn.model.selection libraray and import train_stest_split as a functions

1)from sklearn.model_selection import train_test_split

2)train_set,test_set =train_test_split(housing,test_size=0.2,random_state=42)

42 for random seed you can pick any number 42 is like industry fav number

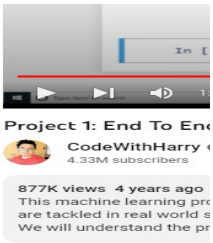
test_ratio 0.25% by defualt although can pick anything

housing is data frame we used

3)print(len(train_set))

4)print(len(test_set))

now lets say we earlier coded this



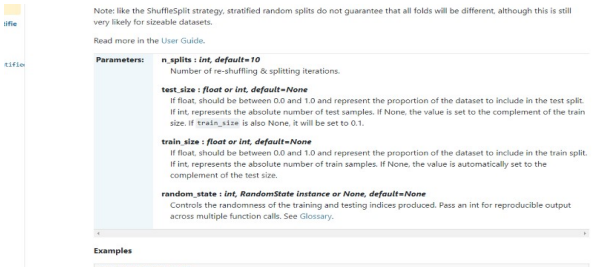
lets say if chas was a very important variable that company needed and 65% of times is 0 how wopuld model distribute test and train data the way where model has somwhat fair 0 and 1 ‘s in both test and train sets so we will use stratified sampling to eliminate this problem

1)from sklearn.model_selection import StratifiedShuffleSplit

importing stratified split function forom sklearn library

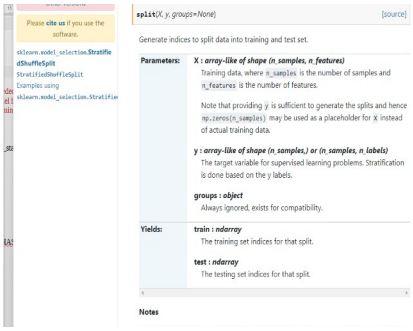
2)split=StratifiedShuffleSplit(n_splits=1,test_size=0.2,random_state=42)

default parameters



3)for train_index, test_index in split.split(housing,housing['CHAS']):

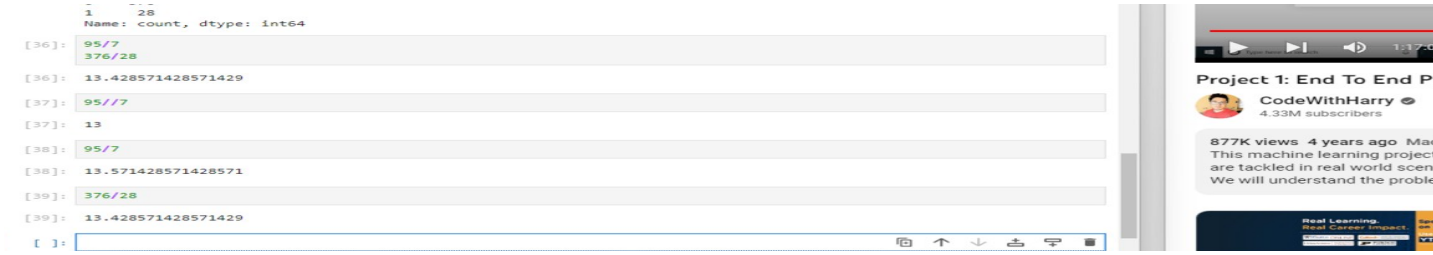
Generate indices to split data into training and test set.



4)strat_train_set=housing.loc[train_index]

he `.loc` accessor is used in pandas, a popular data manipulation library in Python, to select rows and columns from a DataFrame using labels. It provides a way to access data

based on the labels of rows
5)strat_train_set=housing.loc[test_index]
6) strat_test_set
shows test set
7)strat_test_set.info()
describes test set
8)strat_test_set['CHAS'].value_counts()
output:
CHAS
0 95
1 7
Name: count, dtype: int64
so 95 enteries has 0 as CHAS
7 enteries have 1 as CHAS
9)strat_train_set['CHAS'].value_counts()
output:CHAS
0 376
1 28
Name: count, dtype: int64
same for train set



same ratio for 0 and 1 ‘s in test set and training set

looking for corelations now
data set we have is very small so we will find out about corrrelation coffecient

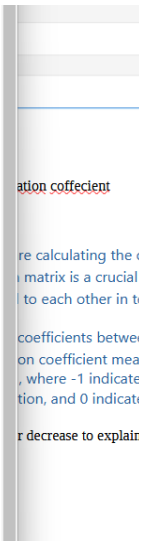
1)corr_matrix = housing.corr()

In the code `corr_matrix = housing.corr()`, you are calculating the correlation matrix for a DataFrame named `housing`. The correlation matrix is a crucial statistical tool used to understand how variables in a dataset are related to each other in terms of their linear relationships.
`corr()`: This method calculates the Pearson correlation coefficients between all pairs of numeric columns in the DataFrame. The Pearson correlation coefficient measures the linear relationship between two variables. It ranges from -1 to 1, where -1 indicates a perfect negative correlation, 1 indicates a perfect positive correlation, and 0 indicates no linear correlation.
In simple words if one variable increase other might increase or decrease to explain this we use pearson coffecient of correlation

2)corr_matrix['MEDV'].sort_values(ascending=False)

medv is median value of occupied homesas you can see in attribute info

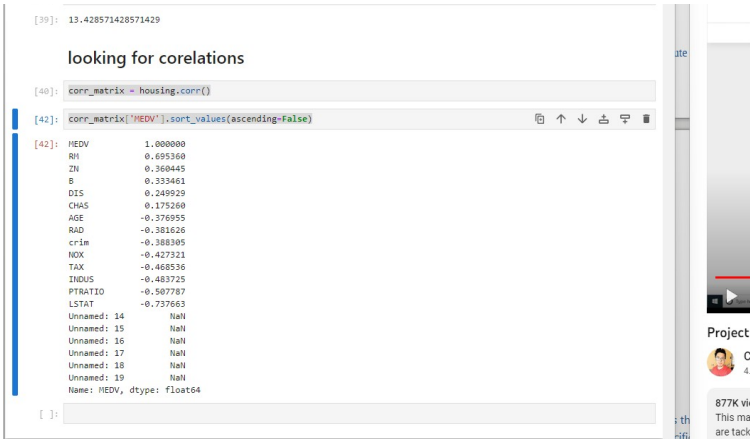
```
   Concerns housing values in suburbs of Boston.

5. Number of Instances: 506

6. Number of Attributes: 13 continuous attributes (including "class"
                         attribute "MEDV"), 1 binary-valued attribute.

7. Attribute Information:

    1. CRIM      per capita crime rate by town
    2. ZN        proportion of residential land zoned for lots over
                 25,000 sq.ft.
    3. INDUS     proportion of non-retail business acres per town
    4. CHAS      Charles River dummy variable (= 1 if tract bounds
                 river; 0 otherwise)
    5. NOX       nitric oxides concentration (parts per 10 million)
    6. RM        average number of rooms per dwelling
    7. AGE       proportion of owner-occupied units built prior to 1940
    8. DIS       weighted distances to five Boston employment centres
    9. RAD       index of accessibility to radial highways
    10. TAX      full-value property-tax rate per $10,000
    11. PTRATIO  pupil-teacher ratio by town
    12. B        1000(Bk - 0.63)^2 where Bk is the proportion of blacks
                 by town
    13. LSTAT    % lower status of the population
    14. MEDV     Median value of owner-occupied homes in $1000's

8. Missing Attribute Values:  None.
```



so we passed corr_matrix via MEDV

`.sort_values(ascending=False)`: This method sorts the correlation values in descending order. The `ascending=False` argument specifies that you want the values in descending order

output:



so if we increase medv attribute
rm is in positive correlations
means more the price more the rooms in rough words
AGE -0.376955
herr we see age has negetive correlation
so homes built prior to 1940 arent that expensive

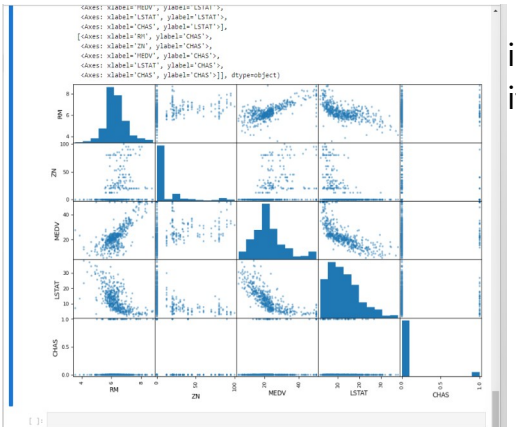
3)from pandas.plotting import scatter_matrix
for making graphs using pandas scatter plots
4}attributes=["RM","ZN","MEDV","LSTAT","CHAS"]
attributes graphs we want to see

5)scatter_matrix(housing[attributes],figsize=(12,8))

`scatter_matrix`: This function is used to create a scatter plot matrix. It's part of the pandas plotting library and is used to visualize relationships between multiple variables.

1. `housing[attributes]`: This selects specific columns (attributes) from the DataFrame `housing`. `attributes` is assumed to be a list of column names that you want to include in the scatter plot matrix.
2. `figsize=(12, 8)`: This parameter specifies the size of the figure (plot) in inches. It sets the width to 12 inches and the height to 8 inches.

Output:



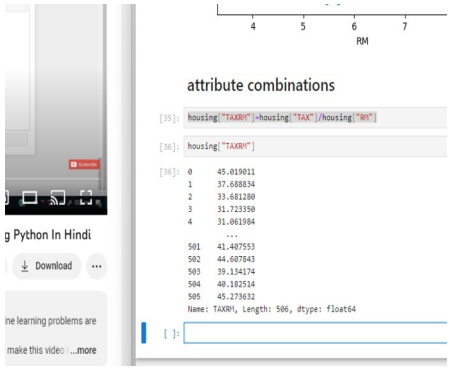
notice it doesent show straight graphs when medv is matched with medv and so on
it shows histograms to show variations in plotting

to plot one graph
6) housing.plot(kind= "scatter",x="RM",y="MEDV",alpha=0.8)
where kind = scatter
x and y are mapped to rm and medv
and alpha
seems like you're referring to setting the alpha (transparency) value to 0.8 in a scatter plot. In the context of data visualization, particularly with libraries like Matplotlib in Python, you can use the alpha parameter to control the transparency of data points in a scatter plot. An alpha value of 0.0 makes the points completely transparent, while an alpha value of 1.0 makes them fully opaque.

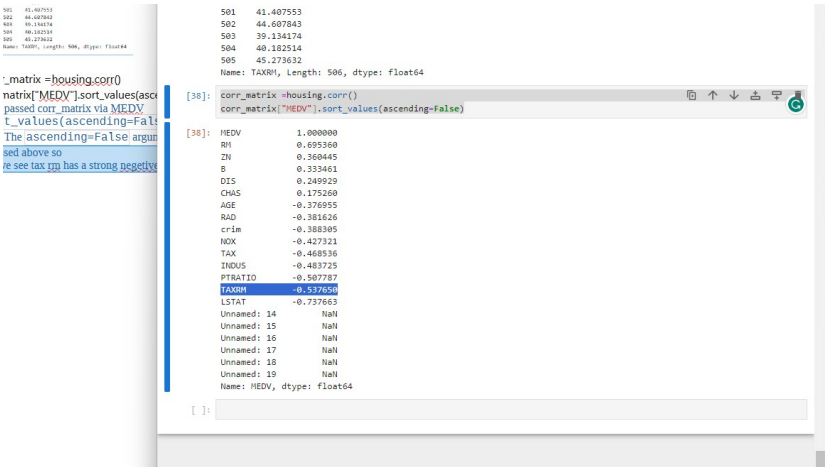
Attribute combinations

1)housing["TAXRM"]=housing["TAX"]/housing["RM"]
we created a new attribute taxrm which shows us tax per room
. RM average number of rooms per dwelling
TAX full-value property-tax rate per $10,000

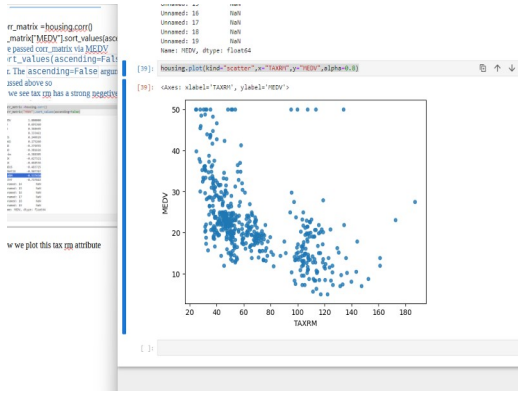
2)housing["TAXRM"]
to show values of taxrm attribute



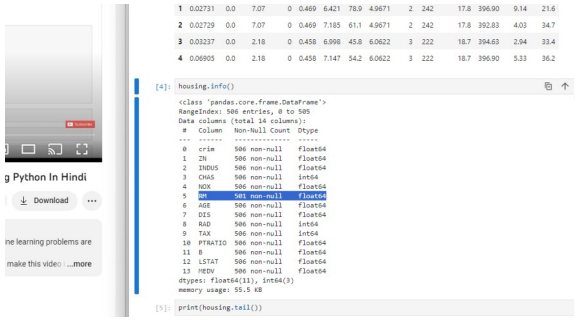
3)corr_matrix =housing.corr()
corr_matrix["MEDV"].sort_values(ascending=False)
so we passed corr_matrix via MEDV
`.sort_values(ascending=False)`: This method sorts the correlation values in descending order. The `ascending=False` argument specifies that you want the values in descending order discussed above so
here we see tax rm has a strong negetive correlation



4)now we plot this tax rm attribute
housing.plot(kind="scatter",x="TAXRM",y="MEDV",alpha=0.8)
these dots outside of dense cluster are called outliers part of data cleaining is to remove these out liers



if data had missing attributes
for this we deleted some RM enteries from data
now we will rerun or restart whole kernal because we changed data it might change past outputs we had



now see rm has 501 enteries now because we deleted 5 enteries

To take care of missing Attributes we have 3 options
1) get rid of missing data points
A=housing.dropna(subset=[“RM”])
a.shape
output:501,15
and original housing dataframe remains unchanged
2)get rid of whole attribute
housing.drop(“RM”,axis=1)
output :no RM column and original housing data fframe remains unchanged
3)set the value to some other value(o ,mean or median)
median=housing["RM"].median()
housing["RM"].fillna(median)
using pandas library creates a dataframe median
then fill Rm values having na with median values
cant choose first 2 options because it has positive strong correlation

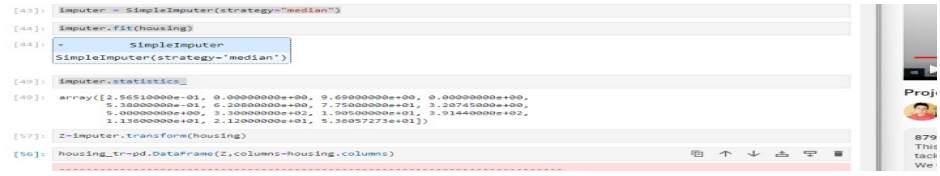
still original housing data frame is unchanged after 3rd option

now to do same stuff easily using library we impoet simpleimputer from sklearn library
1)from sklearn.impute import SimpleImputer

2)imputer = SimpleImputer(strategy="median")
simple imputer will use median to fill na spots

3)imputer.fit(housing)
to impouter to fit into housing dataframe

4)imputer.statistics_
the `statistics_` attribute is used to access the statistics that were used by the imputer to replace missing values in a dataset.



5) Q = imputer.transform(housing)

he `transform` method of the `SimpleImputer` instance (`imputer`) is applied to the `housing` dataset, and it returns a new dataset (`Q`) with the missing values replaced using the strategy specified (such as mean, median, etc.).

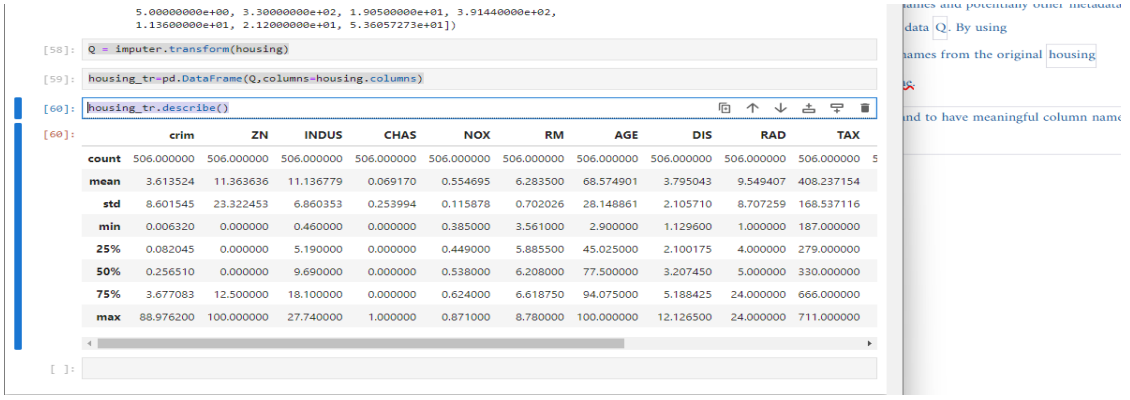
6) housing_tr=pd.DataFrame(Q,columns=housing.columns)

t looks like you're creating a new DataFrame called `housing_tr` by using the imputed data `Q` and assigning the columns from the original `housing` DataFrame. This is a common step after performing imputation, as it helps maintain the column names and potentially other metadata.

you create the `housing_tr` DataFrame using the imputed data `Q`. By using `columns=housing.columns`, you ensure that the column names from the original `housing` DataFrame are retained in the new `housing_tr` DataFrame.

> This is a common practice to keep your data consistent and to have meaningful column names in the imputed dataset.

7)housing_tr.describe()



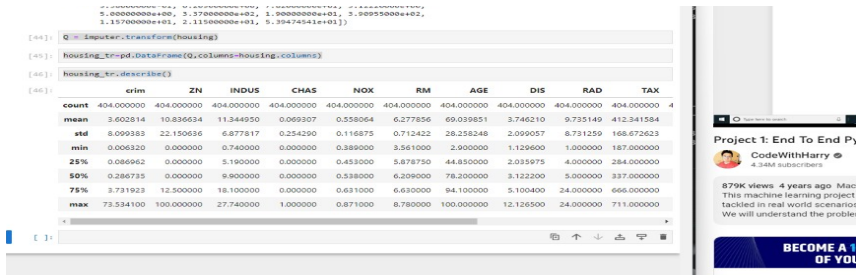
```
5.00000000e+00, 3.30000000e+02, 1.90500000e+01, 3.91440000e+02,
1.13600000e+01, 2.12000000e+01, 5.36057273e+01])

[58]: Q = imputer.transform(housing)

[59]: housing_tr=pd.DataFrame(Q,columns=housing.columns)

[60]: housing_tr.describe()
```

[60]:

	crim	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.283500	68.574901	3.795043	9.549407	408.237154
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.702026	28.148861	2.105710	8.707259	168.537116
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000	1.129600	1.000000	187.000000
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000	2.100175	4.000000	279.000000
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208000	77.500000	3.207450	5.000000	330.000000
75%	3.677083	12.500000	18.100000	0.000000	0.624000	6.618750	94.075000	5.188425	24.000000	666.000000
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	12.126500	24.000000	711.000000

```
[ ]:
```

we want to perform this on a train datasetfor real projects so before finding correlations
we type this
housing=strat_train_set.copy()
so now its copy of train set
404 enteries
so now output will be



```
5.00000000e+00, 3.37000000e+02, 1.90000000e+01, 3.90955000e+02,
1.15700000e+01, 2.11500000e+01, 5.39474541e+01])

[44]: Q = imputer.transform(housing)

[45]: housing_tr=pd.DataFrame(Q,columns=housing.columns)

[46]: housing_tr.describe()
```

[46]:

	crim	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX
count	404.000000	404.000000	404.000000	404.000000	404.000000	404.000000	404.000000	404.000000	404.000000	4
mean	3.602814	10.836634	11.344950	0.069307	0.558064	6.277856	69.039851	3.746210	9.735149	412.341584
std	8.099383	22.150636	6.877817	0.254290	0.116875	0.712422	28.258248	2.099057	8.731259	168.672623
min	0.006320	0.000000	0.740000	0.000000	0.389000	3.561000	2.900000	1.129600	1.000000	187.000000
25%	0.086962	0.000000	5.190000	0.000000	0.453000	5.878750	44.850000	2.035975	4.000000	284.000000
50%	0.286735	0.000000	9.900000	0.000000	0.538000	6.209000	78.200000	3.122200	5.000000	337.000000
75%	3.731923	12.500000	18.100000	0.000000	0.631000	6.630000	94.100000	5.100400	24.000000	666.000000
max	73.534100	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	12.126500	24.000000	711.000000

```
[ ]:
```

scikit learn has 3 objects
primarly three types of projects
1)ESTIMATORS
estimates some parameters based on datasets eg imputer we used above it has
fit method-fits in the datasets and caluclates internal parameters

2)TRANSFORMERS
transforms method takes input and returns output based on learnings from fit() it also has convieneince function called fit_transofrm()

3)PREDICTORS
linear regression modal is an example of predictor
fit() predict() are two common functions it also gives score() function which will evaluate the predictions

feature scaling
two types of feature scaling method
1) MIN-MAX SCALLING (NORMALIZATION)
(value-min)/(max-min) will come between 0 and 1
sklearn provides a class called min-max scaler for this

2)STANDERDIZATION
(value-mean)/std
sklearn provides a class called standerd scaler for this

creating a pipeline
1)from sklearn.pipeline import Pipeline
importing pipeline feature

2)from sklearn.preprocessing import StandardScaler
importing standerdisation lib

3)my_pipeline=Pipeline([
    ('imputer',SimpleImputer(strategy="median")),
    ('std_scaler',StandardScaler()),
    ])

Pipeline: A `Pipeline` is a way to streamline and automate a sequence of data preprocessing and modeling steps. It ensures that the steps are executed in a specified order, making it easier to manage and deploy machine learning workflows.

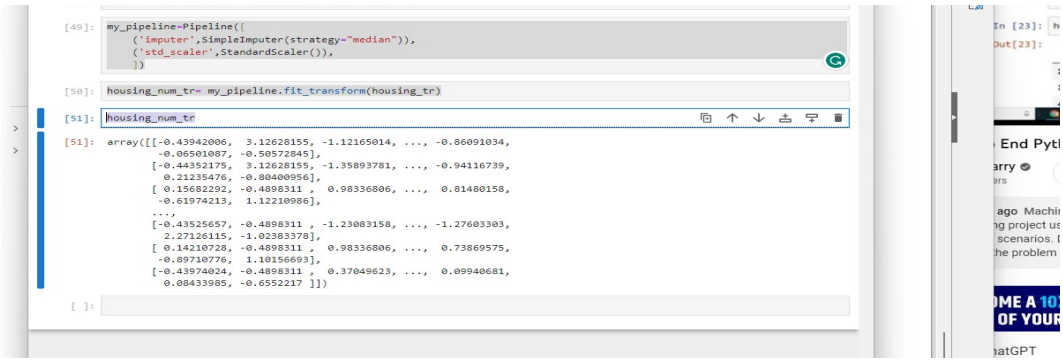
1. **('imputer', SimpleImputer(strategy="median"))**: In this part of the pipeline, the first step is to handle missing values in the dataset using the `SimpleImputer` class. The `SimpleImputer` is used to replace missing values with a specified strategy, and in this case, the strategy is set to "median". This means that missing values in each column will be replaced with the median value of that column.
2. **('std_scaler', StandardScaler())**: The second step in the pipeline involves standardizing the data using the `StandardScaler` class. Standardization (also called Z-score normalization) scales the features so that they have a mean of 0 and a standard deviation of

1. This preprocessing step is often used to ensure that all features are on a similar scale, which can help improve the performance of some machine learning algorithms.

To summarize, the given pipeline first replaces missing values with the median value for each column and then standardizes the features by scaling them to have a mean of 0 and a standard deviation of 1. This is a common preprocessing pipeline that is used to prepare data for various machine learning models. You can further extend this pipeline by adding more steps or customizing the existing ones based on the specific requirements of your machine learning task.

4)housing_num_tr= my_pipeline.fit_transform(housing_tr)
fitting pipeline into housing_tr(training set) and transofrming it

5)housing_num_tr
outputs a numpy array
we need numpy as input



selecting a desired model

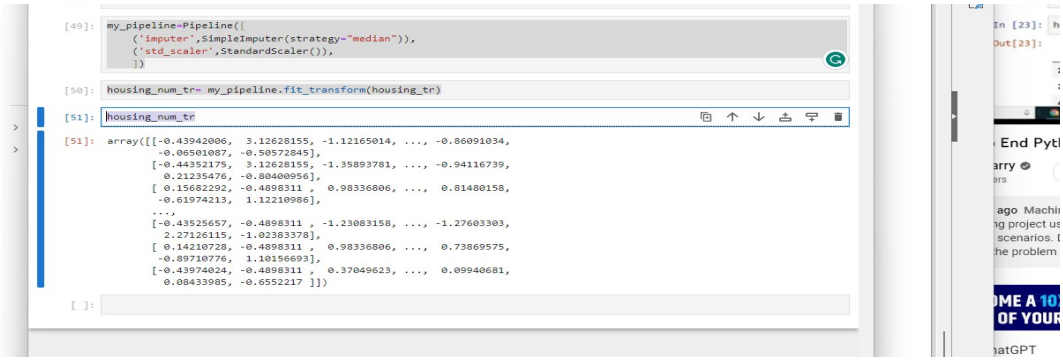
1)from sklearn.linear_model import LinearRegression
model=LinearRegression()

Import LinearRegression: You import the `LinearRegression` class from the `sklearn.linear_model` module. This class provides the functionality to create and train a linear regression model. **Create a LinearRegression Model**: You create an instance of the `LinearRegression` class by calling `LinearRegression()`. This initializes the linear regression model with its default settings.

2BEFORE THIS WE ADD THIS CODE BEFORE MISSING ATTRIBUTES

+)housing=strat_train_set.drop("MEDV",axis=1)
`strat_train_set.drop("MEDV", axis=1)`: This line of code is used to create a new DataFrame named `housing` by dropping the column labeled "MEDV" from the `strat_train_set`
+)housing_labels=strat_train_set["MEDV"].copy()
`strat_train_set["MEDV"].copy()`: This line of code is creating a new Series named `housing_labels` by extracting the values from the "MEDV" column of the `strat_train_set` DataFrame. The `.copy()` method ensures that you create a copy of the extracted values rather than just referencing them. This is important because modifications to `housing_labels` won't affect the original DataFrame.
SPERATING HOUSING AND HOUSING LABEL

resuming to linear regression
2)model.fit(housing_num_tr,housing_labels)
this is house num tr



housing labels we created above by extracting values from MEDV

3)some_data=housing.iloc[:5]
takes 5 enteries from hosuing dataframe
4)housing_labels=housing_labels.iloc[:5]
takes five enteries from MEDV from train set

5)prepared_data=my_pipeline.transform(some_data)
5 enteries from housing gets transormed into pipeline
5)model.predict(prepared_data)
gives ouput predictions
array([23.94921288, 27.21581435, 20.57575334, 25.06681527, 23.77281415])
now we compare it to our labels

6)some_labels
254 21.9
348 24.5
476 16.7
321 23.1
326 23.0
Name: MEDV, dtype: float64
gives output like this
compare it to predictions they arent that good last one good though

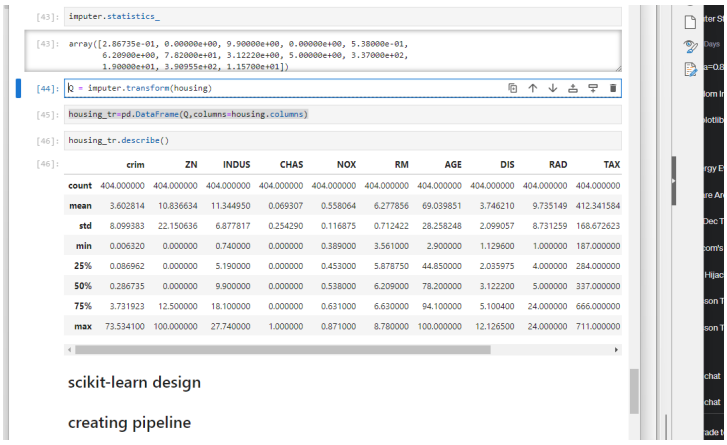
convert it ot list
7)list(some_labels)
we get output like
[21.9, 24.5, 16.7, 23.1, 23.0]

[ ]:

evaluating the model
1)from sklearn.metrics import mean_squared_error

This statement imports the `mean_squared_error` function from the `sklearn.metrics` module. This function is commonly used to calculate the mean squared error (MSE) between two sets of data, typically predictions and actual target values. The mean squared error is a common metric used to evaluate the performance of regression models.

2)housing_predictions= model.predict(housing_num_tr)
where model= linear regression()
and housing_num_tr= my_pipeline.fit_transform(housing_tr)
where housing tr is



3)lin_mse=mean_squared_error(housing_labels,housing_predictions)
mse is mean squared error finction of lib sk learn
where we declared housing predictions above and
housing_labels we know from this code we wrote before
housing=strat_train_set.drop("MEDV",axis=1)
housing_labels=strat_train_set["MEDV"].copy()
4)lin_rmse=np.sqrt(lin_mse)
we calculate root mean squareb error

```
[67]: lin_mse
[67]: 23.352925815303564
[68]: lin_rmse
[68]: 4.832486504409874
[ ]:
```

but caluclated housing_label values were
[21.9, 24.5, 16.7, 23.1, 23.0]
23 very less than these
so we will descard this model

now we will use decioson tree regressor
so we will alter above code
we converted code from liner regression to decision tree
commented out liner regressio n using #

selecting a desired model

```
[52]: #from sklearn.linear_model import LinearRegression
      from sklearn.tree import DecisionTreeRegressor

[55]: #model=LinearRegression()
      model=DecisionTreeRegressor()

[56]: model.fit(housing_num_tr,housing_labels)

[56]: DecisionTreeRegressor
      DecisionTreeRegressor()

[ ]: some_data=housing.iloc[:5]

[ ]: some_labels=housing_labels.iloc[:5]

[ ]: prepared_data=my_pipeline.transform(some_data)

[ ]: model.predict(prepared_data)

[ ]: list(some_labels)
```

but now our root mean square error goes 0 how?
We overfitted out traininbg daa and decison tree cleared all noise in training data
Amyways we rename them as rmse and mse because they are not part of lin_ (linear regression now)

```
[64]: lin_mse
[64]: 0.0
[65]: lin_rmse
[65]: 0.0
[ ]:
```

using cross validation
1) from sklearn.model_selection import cross_val_score
imports cross val func from sk leanr libraries

2)scores=cross_val_score(model,housing_num_tr,housing_labels,scoring="neg_mean_squared_error",cv=10)
`cross_val_score`: This function is part of scikit-learn's `model_selection` module and is used for performing cross-validation. It takes several parameters:

- `model`: This should be your trained machine learning model that you want to evaluate.
- `cv=10`: This parameter specifies the number of folds for cross-validation. In this case, you're using 10-fold cross-validation, meaning the data will be split into 10 parts, and the model will be trained and evaluated 10 times.
- **eg_mean_squared_error:** In scikit-learn's convention, scoring functions typically consider higher values to be better. Since lower values of MSE indicate better performance (as they indicate smaller prediction errors), the `neg_mean_squared_error` scoring function returns the negation of the mean squared error. This is done to align with the convention of higher scores being better.

3)rmse_scores=np.sqrt(-scores)
scores are already negetive so we have to -scores to root square it since
`neg_mean_squared_error` scoring function returns the negation of the mean squared error. This is done to align with the convention of higher scores being better.

4)rmse_scores
outputs :array([3.89223743, 5.73748904, 4.98666514, 3.84111264, 3.81562577,
2.93027302, 6.91642971, 3.58514295, 3.40899545, 3.88052831])

[ ]:

less error compared to linear regression
using linear regression

array([4.27001096, 4.26034932, 5.11177484, 3.83151625, 5.3435416 ,
4.38995448, 7.46842271, 5.49003038, 4.1392956 , 6.05405933])
so we choose decision tree regressor

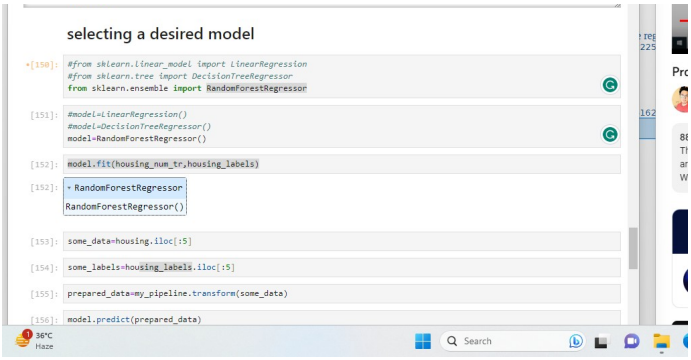
5)def print_scores(scores):
    print("scores:",scores)
    print("Mean:",scores.mean())
    print("standerdDeviation:",scores.std())

created a function to pint scores its mean and its std deviation
these functions included in numpy mean std mean and all

7)print_scores(rmse_scores)
output not we using rmse scores given by decision tree regressor
scores: [4.06276969 5.73236426 5.25725979 4.02922252 3.62287869 2.66622392
 6.82764601 3.82298967 3.51983664 4.29918597]
Mean: 4.384037716238515
standerdDeviation: 1.1567149701246144

using linear regression we get output:
scores: [4.27001096 4.26034932 5.11177484 3.83151625 5.3435416 4.38995448
 7.46842271 5.49003038 4.1392956 6.05405933]
Mean: 5.035895549223018
standerdDeviation: 1.0545235927112007

using random forest regressor



we get this output
scores: [3.02807391 2.76676348 4.36154577 2.67570716 3.29508102 2.65835356
 4.86175816 3.27670055 3.11379993 3.40758445]
Mean: 3.3445367979314087
standerdDeviation: 0.6893969355768271

randomforest regresor works better than deciosion tree regressor and linear regressor

note:Overfitting is a common problem in machine learning where a model learns the training data too well, to the point that it captures the noise and random fluctuations in the data rather than just the underlying patterns. As a result, an overfitted model may perform very well on the training data but fails to generalize well to new, unseen data. In essence, the model becomes too specialized to the training data and loses its ability to make accurate predictions on new data.
That was thing with decison tree regressor before we cross validated it

saving the model

1) from joblib import dump,load

The `joblib` library in Python provides tools for efficiently saving and loading Python objects, especially large numerical arrays, to and from disk. It is commonly used for saving machine learning models, trained parameters, and other data structures. The `dump` and `load` functions are two important functions from the `joblib` library.

2)dump(model, 'my.joblib')

`dump` function is used to save Python objects to disk in a binary format that is optimized for speed and efficiency. This function is often used to save trained machine learning models, so you can reuse them later without retraining.

`load` Function: The `load` function is used to load the previously saved objects back into memory. It allows you to retrieve and use the objects without retraining or reprocessing.

Testing model on test data

1)x_test=strat_test_set.drop("MEDV",axis=1)
2)y_test=strat_test_set["MEDV"].copy()
already done before attribute thing
3)x_test_prepared=my_pipeline.transform(x_test)
apply pipeline to transorm x test
my pipeline we defined before
4)final_predictons= model.predict(x_test_prepared)
predicting from randomforest regressor

5)final_mse=mean_squared_error(y_test,final_predictons)
applied room=t_mean_squared functions to final predictions using randomforestregressor this time on training set
6)final_rmse=np.sqrt(final_mse)
applied same as before
7)final_rmse
error reduces output:3.0425337560761325

8)print(final_predictons,list(y_test))

model working very good