

CYBR 486 - Lab #3 - Time to make, train, and evaluate a linear regression model!!

This lab will have you reading a dataset in as memory, determining whether that dataset needs some pre-processing in order for it to be valid to train a model on (This one won't but some in the future likely will). You will also be taking the data frame of the dataset and splitting it into train and test subsets, creating the model, training it, making predictions with it, and then evaluating its performance.

1.) Import statements and setup

First, our import statements for this lab:

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score,
mean_absolute_error
import numpy as np
import pandas as pd
```

We have some new ones this time, and all of the new ones serve as a callable method or “constructable object” (in the case of the LinearRegression model).

You can browse the scikit-learn documentation if you want to learn more about them but as a brief summary:

Train_test_split → relatively simple function used to split our X and y variables and split them into a training set and testing set.

LinearRegression → our linear regression model object. Has a lot of functionality but for this lab we're mainly using it to train on a training set, then have the model make predictions on the test set.

sklearn.metrics → The various functions imported on this line are tools used to evaluate the performance of the model by providing a score or quantifying the error calculated after making predictions on the test data

Lastly, we're reading the BonsttonHousing.csv file into a data frame like we've done with the iris.csv with the small change that we need to manually separate the data frame into our X, and y variables. When loading a curated dataset from sklearn or some other sources, there is often a Boolean argument that can be used when creating the data frame object (return_X_y=), supplying a True after the '=' would return the X and y parts already separated but I want you to have practice manually creating subsets for data frames.

Question 1: Write the code you would use to manually separate a pandas dataframe of the provided housing dataset into the X, and y subsets and show it below.

```
# Assuming the target variable is in the last column of the dataset
X = df.iloc[:, :-1] # All columns except the last one as features
y = df.iloc[:, -1] # The last column as the target
```

```
# Displaying the first few rows of X and y for confirmation
print("X (features) subset:")
print(X.head())


print("\ny (target) subset:")
print(y.head())
```

2.) Data types and Nulls

There are a number of things that aren't allowed when training and evaluating a linear regression model on a dataset. By default, columns that contain strings, or columns that have null or empty values are problematic. Model training requires numerical values, and strings aren't numerical values. There are ways around this, like encoding strings to numeric values, mapping a known number of strings sequentially to numbers (e.g. like {"apple", "peach", "pear"} = {1, 2, 3}), or even extreme measures like removing a string column from the dataset (which should only be used when you can determine that that column has little effect on predicted values). Similarly for null values, you can remove the rows entirely that have null values to avoid this problem, but this only works when you have few nulls. If there are a lot of nulls and you end up removing a large portion of the dataset, you have a problem. There are options for this as well, like filling in null values with a value that is estimated to be within the average of other values of the column. But before you consider any of these options you need to determine what data types the columns contain, and whether there are any null values.

Question 2a.) There is a method call that will show all the columns and their data types. What is this method? Also print the output of this method call and put a screenshot of that output below.

```
# Display the data types of all columns  
print("Columns and their data types:")  
print(df.dtypes)
```

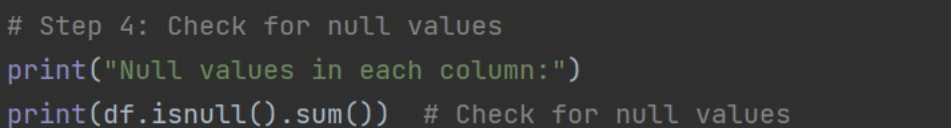


```
Columns and their data types:  
crim      float64  
zn        float64  
indus     float64  
chas      int64  
nox       float64  
rm        float64  
age       float64  
dis       float64  
rad       int64  
tax       int64  
ptratio   float64  
b         float64  
lstat     float64  
medv      float64  
dtype: object
```

Regarding determining if there are null values, this particular dataset doesn't have nulls in it, mainly because this is the first time we're training a model. But we still need to verify that there are no nulls because you will need to be able to do this in the future. Browse the documentation for pandas dataframes:

<https://pandas.pydata.org/docs/reference/frame.html>

Question 2b.) Looking at the documentation above, look for a method involving null or nulls. Try out the various methods that you think might be useful for determining if the dataset has any null values. Take a screenshot of at least 1 example of the code you used and the printed output that shows if any columns have null values for our housing dataset and post it below.



```
# Step 4: Check for null values  
print("Null values in each column:")  
print(df.isnull().sum()) # Check for null values
```

```
Checking for null values in each column:
```

```
crim      0
zn        0
indus     0
chas      0
nox       0
rm        0
age       0
dis       0
rad       0
tax       0
ptratio   0
b         0
lstat     0
medv      0
dtype: int64
```

3.) Split the dataset into training and test portions

At this point, we should have a dataset, read into memory as a data frame, and separated into X and y variables. This dataset should also be suitable to build a model and train as it shouldn't have any non-numeric values or any null values. The last step we need to perform before creating and training the model is to split our dataset into training and test subsets.

To achieve this, we have imported a handy function → `train_test_split()`. This function takes various arguments, the main ones being the “arraylike” objects of our X and y data subsets, and a value denoting the percentage of samples to be used for training or testing.

```
X_train, X_test, y_train, y_test = train_test_split(data_X, data_y,
test_size=0.2)
# X shape outputs (number of data entries, number of features)
print("Training data shape:", X_train.shape)
print("Testing data shape:", X_test.shape)
# y shape outputs the same
print("Training data shape:", y_train.shape)
print("Testing data shape:", y_test.shape)
```

As shown above, we can pass both our X and y subsets as arguments, and the size is determined by pre-defined “test_size=” or “train_size=” arguments. In this case

we want to use 80% of the data samples to train, and 20% to test. There are additional possible arguments for this function such as "random_state=" which takes an int that serves as the random seed if you want to be able to reproduce the results multiple times while randomly selecting samples, and "shuffle=" which takes a Boolean to determine whether the data samples should be randomly shuffled before splitting.

Question 3.) Why might you want to shuffle the data, or randomly select which data samples goes into the test or train subsets? Is there a reason or situation where that would be a bad thing?

Shuffling the data before splitting it into training and test sets is crucial for preventing bias and ensuring that the model learns meaningful patterns rather than memorizing any inherent order in the dataset. In many datasets, especially those collected over time or sorted by specific criteria (such as increasing house prices or customer age), the order of the data may not be random. If the model trains data in this predefined order without shuffling, it could learn trends related to the sequence instead of the actual relationships between features and the target variable. This could result in a model that performs well on similarly ordered data but poorly on real-world, unordered data.

By shuffling the data, you ensure that both the training and test sets have a representative mix of all data points, helping the model generalize better to unseen cases. For instance, in a housing dataset, shuffling ensures that houses from various price ranges are included in both the training and test sets, preventing the model from overfitting to a specific segment of the data.

However, there are scenarios where shuffling is inappropriate, particularly in time series data. In time series, the sequence of data points is important because past events influence future outcomes. Shuffling would disrupt this temporal order, making it impossible for the model to capture time-based patterns. In such cases, data is typically split chronologically, using earlier data for training and later data for testing, to maintain the sequence integrity.

4.) Make the model, Train the model, Predict the test set

Below is example code that creates a linear regression model object, trains that model on the separated training data while providing the actual labels as well, then makes predictions on the test data subset.

```
def buildModel(X_train, X_test, y_train):
```

```
# create the model object
model = LinearRegression()

# train the model, the line below shows the basic way to do this but
the dataset we're using, has some columns with string values and by
default we can't use those to train as we need floating point values
model.fit(X_train, y_train)

# generate predictions
y_predictions = model.predict(X_test)
return y_predictions
```

Yes, it's actually that simple to train a model and start making predictions. Granted this doesn't cover how we would take error evaluations and use those to update and improve the model, it's a single training pass but a good starting step. Using `.fit()` or `.predict()` while there are any columns that contain Strings, or any null, na, or NaN entries would cause the program to error which is why its important to be able to determine the data types of columns and to locate null values.

The code is quite simple for this, and there aren't really any observations I could ask you to make at this point, so just try this code out yourself to make sure it works and that you have an example to refer back to for future labs if need be.

5.) Evaluate the model

We've now imported our data set, we've checked whether we need to do any pre-processing of the data to make sure we can properly train and use the model, and we've created trained and generated predictions with that model. Now we have to come up with some type of value that shows how well or poorly the model performed on its first try.

In the lectures we've discussed a variety of methods for evaluating error in a model, ranging from mean absolute error, mean square error, and root mean square error. We also add a new one which is a score evaluation of how well the model did where the others are a measure of calculated error.

MAE:

https://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_absolute_error.html

MSE:

https://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_squared_error.html

R2: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.r2_score.html

We should have a pretty good idea of how MAE, MSE, and RMSE work, R2 is just a tiny bit different, in that it generates a score typically between 0.0 and 1.0. R2 is calculated as →

$$R2 = 1 - (\text{sum of squared residuals}) / (\text{total sum of squares})$$

We don't need to get too deep into the math of this, but the general idea is that an R2 score closer to 1.0 means a perfect score where predictions have low error and are accurately determined based on the actual inputs. A low score or 0.0 means that predictions are no more accurate than just spitting out the mean of the inputs. So long story short, 1.0 = good, lower numbers = bad.... HOWEVER..... R2 doesn't take into account overfit. Meaning you could have a high r2 score but also be overfit and therefore not a good final model.

Question 5.) Checkout the documentation of the various evaluation methods listed above, then I want you to take a screenshot of your code finding these evaluation metrics and printing them as output. You need to include the metrics for mean absolute error, mean squared error, root mean squared error, and r2. HINT: ROOT MEAN SQUARED ERROR IS LITERALLY THE SQUARE ROOT OF MEAN SQUARED ERROR. Look closely at the documentation to determine which arguments you should be using for these functions and in which order. You're comparing actual true labels of the test set, to the predicted labels our model predicted based on the input from the test set. Take a screenshot of the code you used and the outputs.

```
# Step 1: Import necessary libraries
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
import numpy as np
import pandas as pd

# Step 2: Load the dataset
df = pd.read_csv('BostonHousing.csv')

# Step 3: Split the dataset into X (features) and y (target variable)
X = df.iloc[:, :-1] # All columns except the last one
y = df.iloc[:, -1] # The last column

# Step 4: Check for null values
print("Null values in each column:")
print(df.isnull().sum()) # Check for null values

# Step 5: Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Step 6: Train the Linear Regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Step 7: Make predictions on the test data
y_predictions = model.predict(X_test)

# Step 8: Evaluate the model's performance
mae = mean_absolute_error(y_test, y_predictions)
mse = mean_squared_error(y_test, y_predictions)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, y_predictions)
```


Null values in each column:

crim	0
zn	0
indus	0
chas	0
nox	0
rm	0
age	0
dis	0
rad	0
tax	0
ptratio	0
b	0
lstat	0
medv	0

dtype: int64

Mean Absolute Error (MAE): 3.189091965887874

Mean Squared Error (MSE): 24.29111947497374

Root Mean Squared Error (RMSE): 4.928602182665359

R2 Score: 0.6687594935356289

Training data shape (X): (404, 13)

Testing data shape (X): (102, 13)

Training data shape (y): (404,)

Testing data shape (y): (102,)