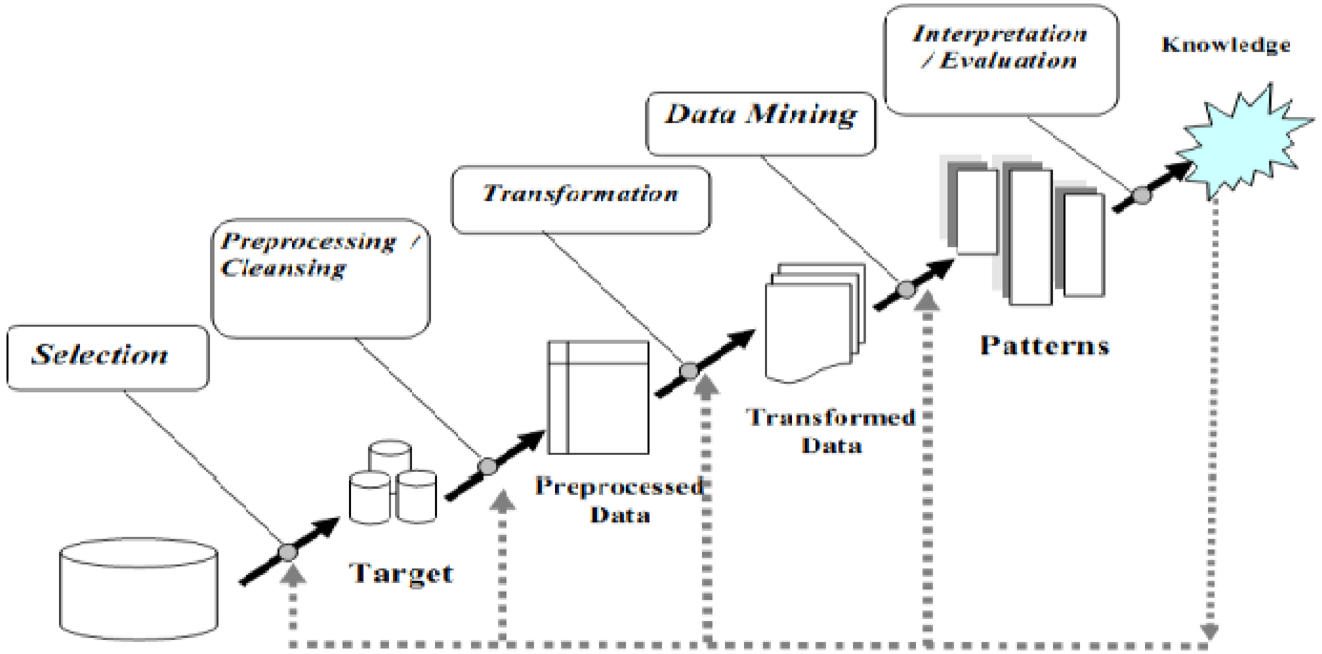
# Data Analytics Process Approaches

## Data Analytics Approaches

- Generic
  - KDD
  - CRISP-DM
- Vendor specific
  - Google
  - Microsoft
  - Cloudera

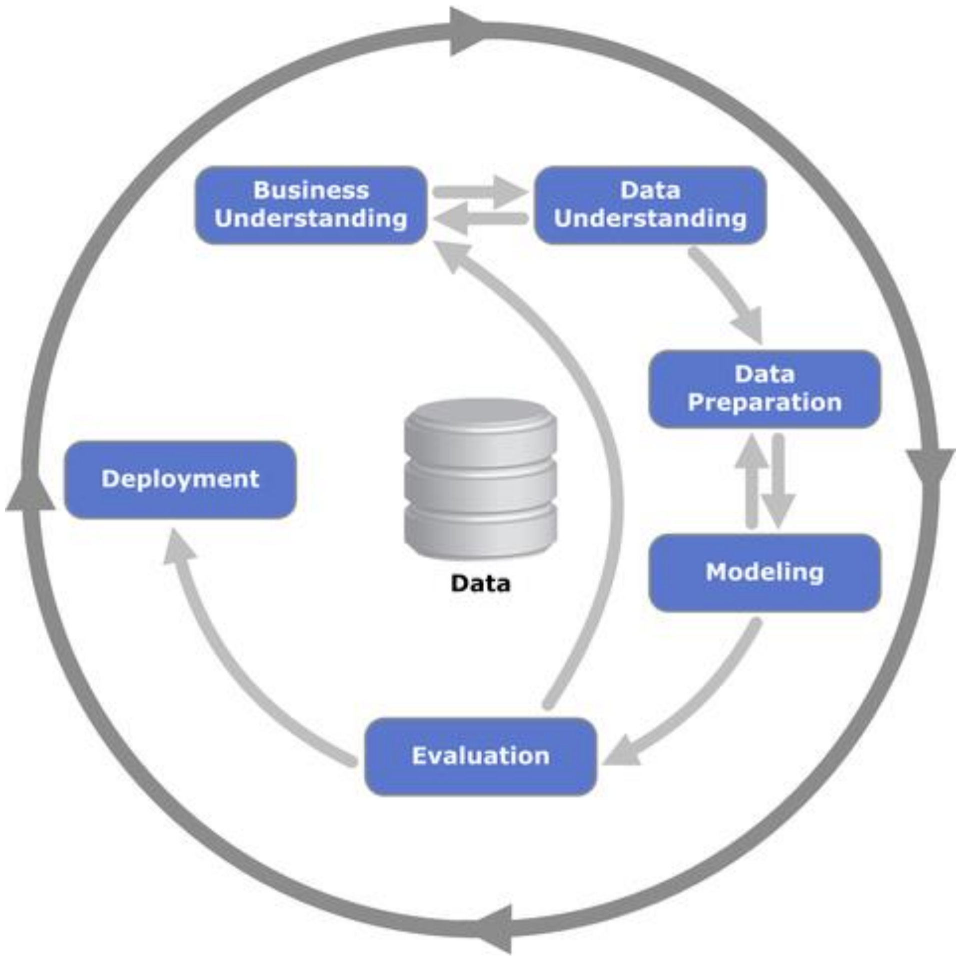
## Knowledge Discovery for Databases (KDD)

- First implemented in 1996
- Divides the process for finding knowledge in data



# CRISP-DM

- Published shortly after KDD
- Active EU project in 1997 with an effort to update the process in 2006–2008
- Major contributor was Daimler-Benz
- Described as an iterative approach and methodology to solve data-mining problems

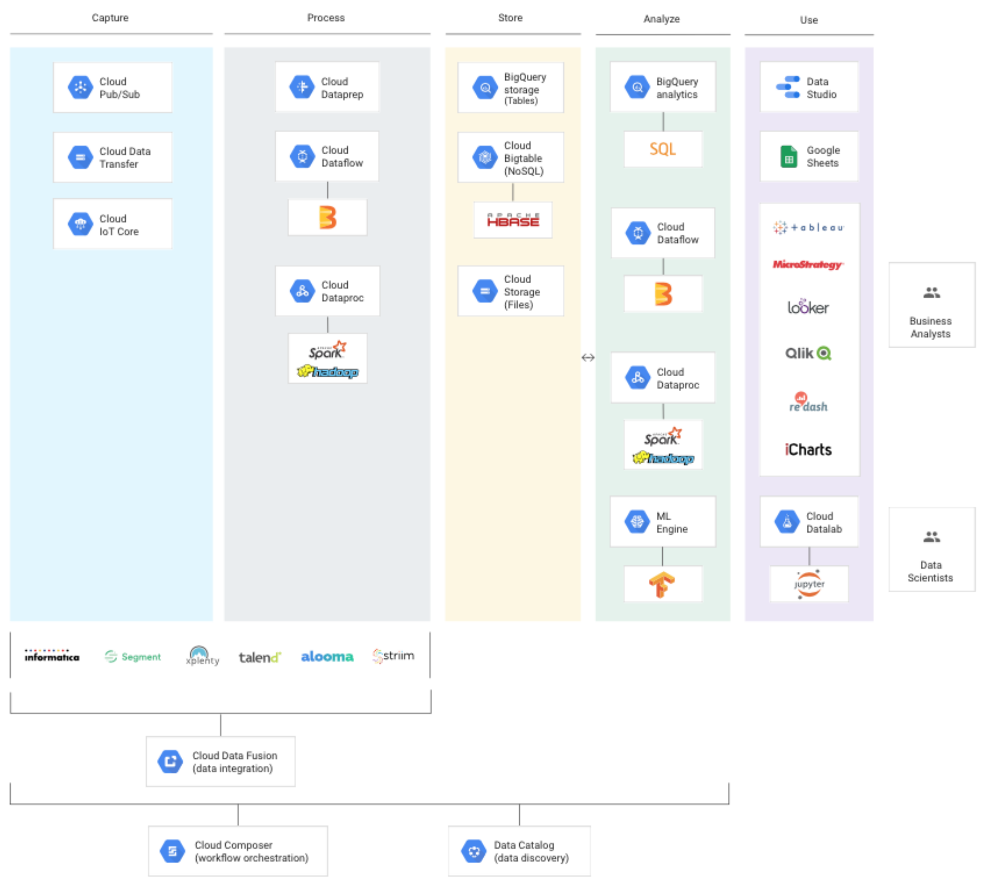


## Vendor Approaches

- Cloud ML players
  - Google
  - Amazon Web Services (AWS)
  - Microsoft
  - Cloudera

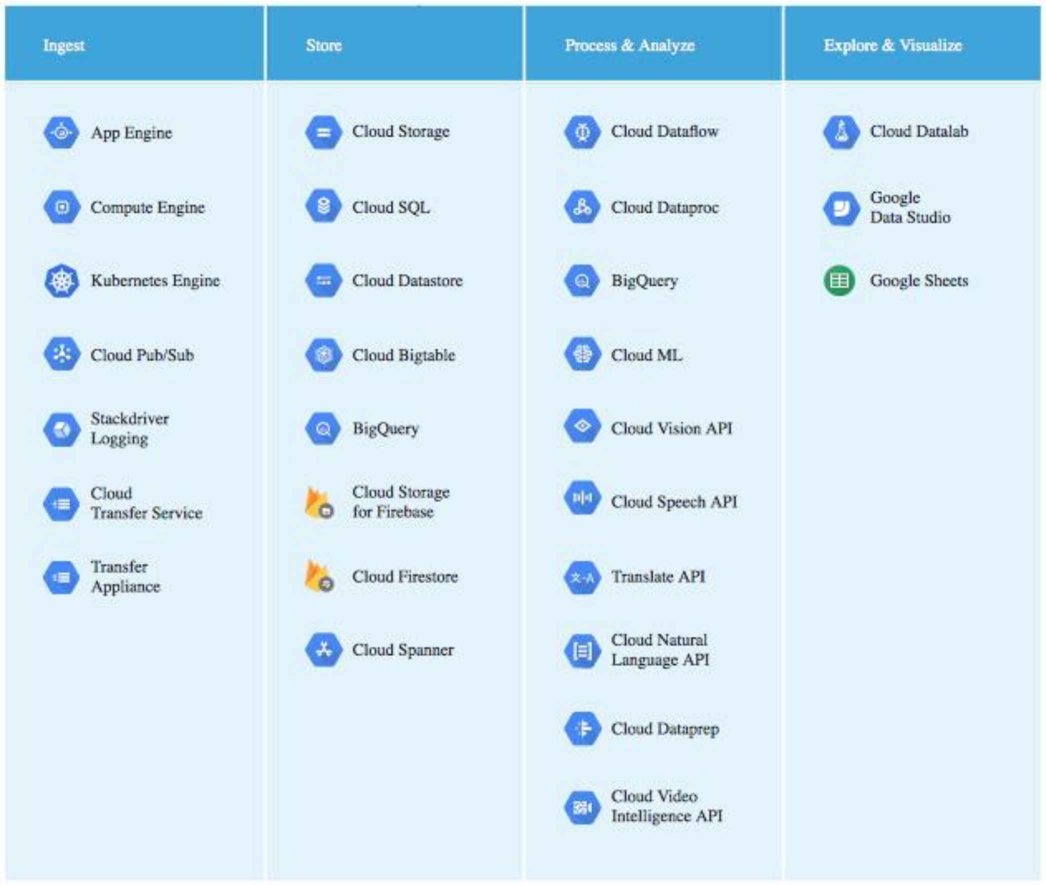
# Google Approaches

- Google Cloud’s big data solutions page (https://cloud.google.com/solutions/big-data)
- Steps
  - Capture
  - Process
  - Store
  - Analyze
  - Use



# Google’s Data Lifecycle

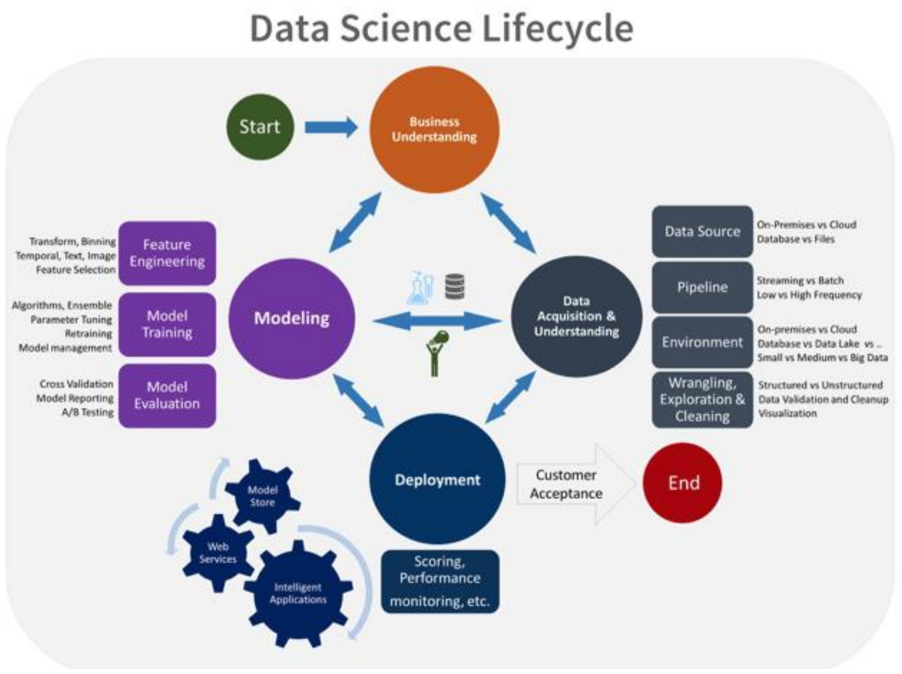
- https://cloud.google.com/solutions/data-lifecycle-cloud-platform
- Steps
  - Ingest
  - Store
  - Process and analyze
  - Explore and visualize



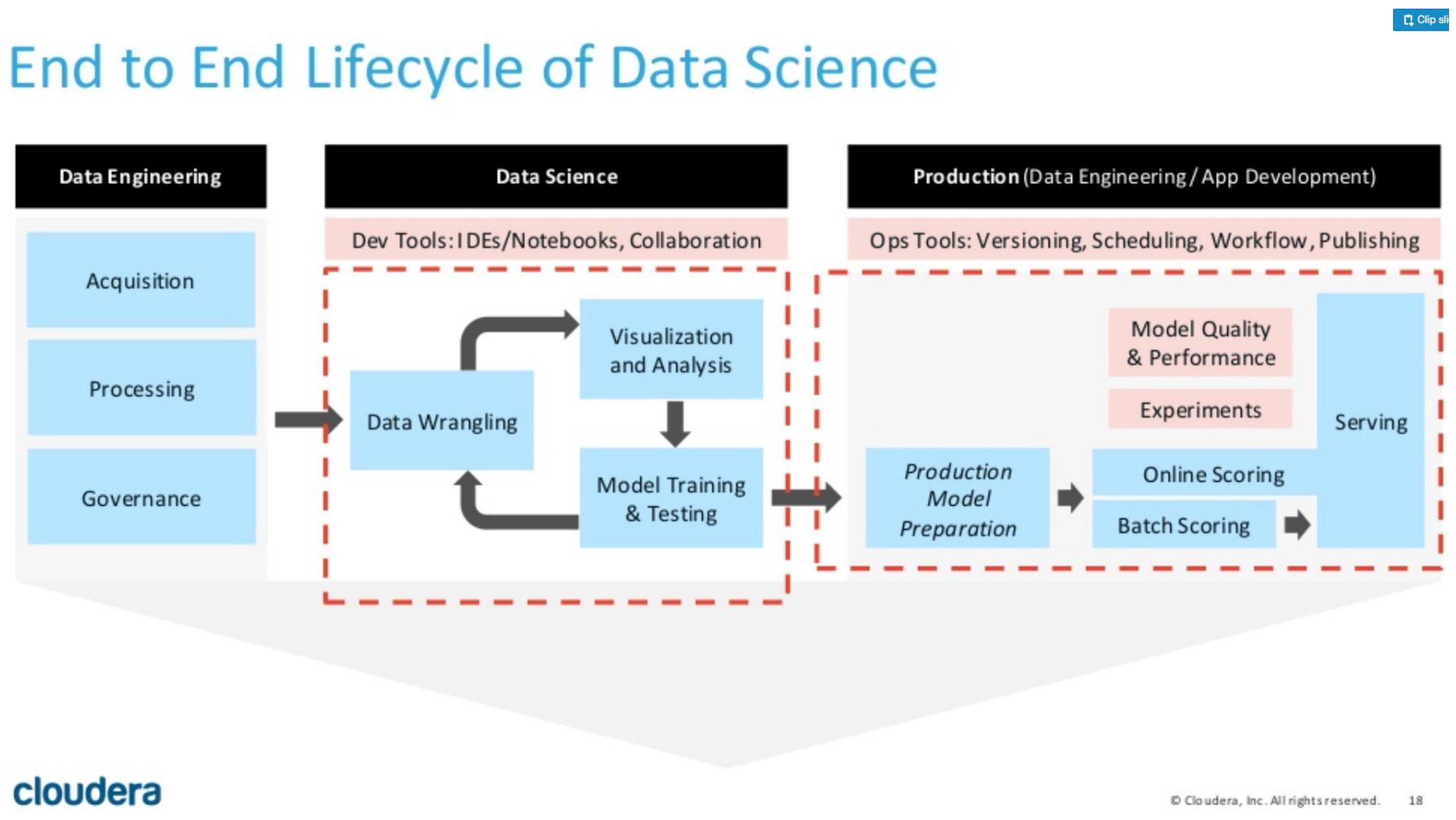
| Ingest | Store | Process & Analyze | Explore & Visualize |
|---|---|---|---|
| App Engine | Cloud Storage | Cloud Dataflow | Cloud Datalab |
| Compute Engine | Cloud SQL | Cloud Dataproc | Google Data Studio |
| Kubernetes Engine | Cloud Datastore | BigQuery | Google Sheets |
| Cloud Pub/Sub | Cloud Bigtable | Cloud ML | |
| Stackdriver Logging | BigQuery | Cloud Vision API | |
| Cloud Transfer Service | Cloud Storage for Firebase | Cloud Speech API | |
| Transfer Appliance | Cloud Firestore | Translate API | |
| | Cloud Spanner | Cloud Natural Language API | |
| | | Cloud Dataprep | |
| | | Cloud Video Intelligence API | |

# Microsoft Team Data Science Process

- https://docs.microsoft.com/en-us/azure/machine-learning/team-data-science-process/overview
- Steps
  - Business understanding
  - Data acquisition and understanding
  - Modeling
  - Deployment
  - Customer acceptance



# Cloudera



End to End Lifecycle of Data Science

Data Engineering
Data Science
Production (Data Engineering / App Development)
Acquisition
Processing
Governance
Dev Tools: IDEs/Notebooks, Collaboration
Data Wrangling
Visualization and Analysis
Model Training & Testing
Ops Tools: Versioning, Scheduling, Workflow, Publishing
Model Quality & Performance
Experiments
Serving
Production Model Preparation
Online Scoring
Batch Scoring
cloudera
© Cloudera, Inc. All rights reserved. 18

Data Analytics Process Approaches

## The End

## Defining the Problem

## Defining the Problem

- Questions to ask
  - What is the business problem you are trying to address?
  - How is it currently done?
  - What performance measure will you use to evaluate your model?
  - What performance criteria do you need to meet to be considered successful?
- Frame it in terms of an analytical solution
  - What kind of machine learning task is it (supervised, unsupervised, classification, regression)?
  - Is it batch or online learning?

## Example: The Boston Housing Dataset

- Our target is median housing price
- This is a regression problem
- The first example we did used only a single feature, but this time we are going to use more, so it is a multiple regression problem
- It is known as a univariate regression problem since we are only trying to predict a single value
- It is a fairly small dataset that can fit into memory, so this is a batch learning problem

## Possible Performance Measures

- A possible performance measure for regression problems is the root mean square error (RMSE)

$$\sqrt{\frac{1}{m}\sum_{i=1}^{m} (h(x^{(i)}) - y^{(i)})^{2}}$$

- Your hypothesis is h
- Your true label is y
- Your number of examples is m
- The ith example is $x^{(i)}$

# Mean Absolute Error

- Another possible performance measure for regression is mean absolute error (MAE)
- Might want to use it if there are a lot of outliers in your data
  - Think about it: what happens if you square an outlier

$$\frac{1}{m}\sum_{i=1}^{m}\left|h(x^{(i)}-y^{(i)}\right|$$

## What Is the Difference Between These Two?

- They are distance measures
  - Also called norms
- RMSE is the Euclidean norm, also called the l2 norm
- MAE is called l1 norm, the distance travelled along the x and y axis to get to the point of the vector
- If outliers aren’t an issue, RMSE is preferred

Defining the Problem

## The End

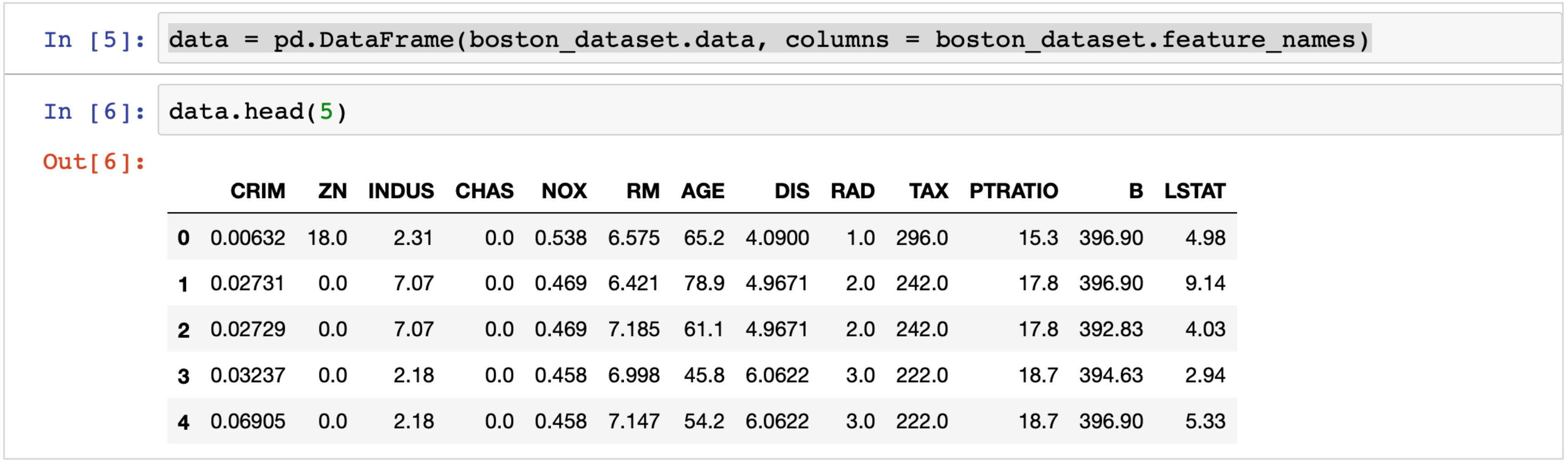
## Getting the Data

# Getting the Data

- Previously, we loaded the data via a Bunch object in scikit-learn
- Let’s now take that Bunch object and make it into a pandas DataFrame
  - This makes it easier to explore the data
- In the scikit-learn Bunch class, which is like a dictionary, here are the following classes:
  - Data
  - Target
  - Feature_names
  - DESCR
- Data, target, and feature_names are ndarrays

# Converting Data to a Pandas DataFrame

- Here is the code to convert the data to a NumPy array

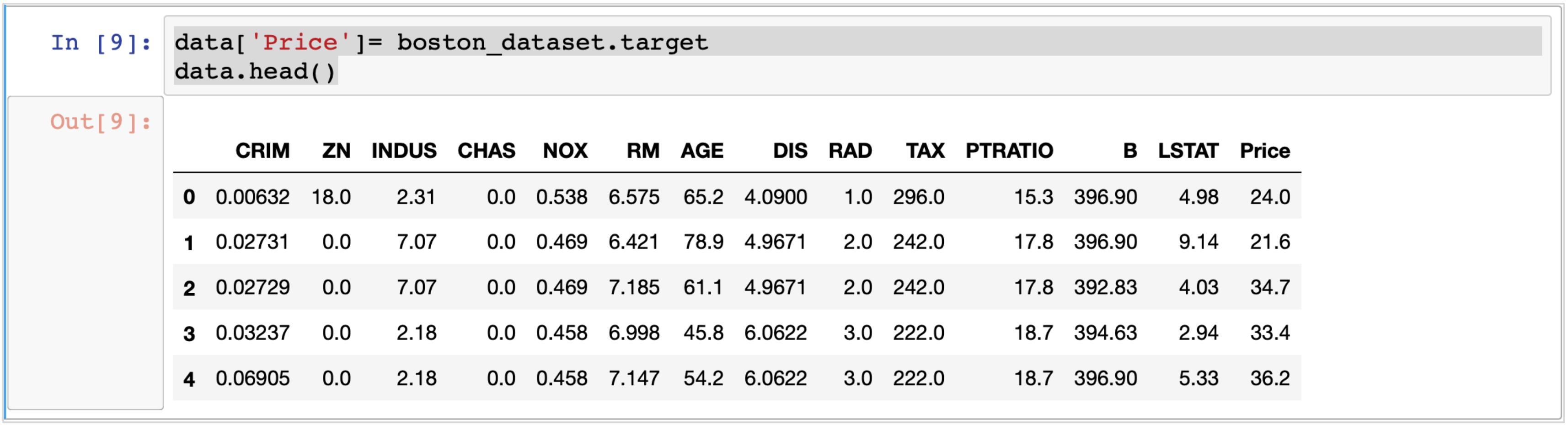


```
In [5]: data = pd.DataFrame(boston_dataset.data, columns = boston_dataset.feature_names)

In [6]: data.head(5)

Out[6]:
```

| | CRIM | ZN | INDUS | CHAS | NOX | RM | AGE | DIS | RAD | TAX | PTRATIO | B | LSTAT |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0.00632 | 18.0 | 2.31 | 0.0 | 0.538 | 6.575 | 65.2 | 4.0900 | 1.0 | 296.0 | 15.3 | 396.90 | 4.98 |
| **1** | 0.02731 | 0.0 | 7.07 | 0.0 | 0.469 | 6.421 | 78.9 | 4.9671 | 2.0 | 242.0 | 17.8 | 396.90 | 9.14 |
| **2** | 0.02729 | 0.0 | 7.07 | 0.0 | 0.469 | 7.185 | 61.1 | 4.9671 | 2.0 | 242.0 | 17.8 | 392.83 | 4.03 |
| **3** | 0.03237 | 0.0 | 2.18 | 0.0 | 0.458 | 6.998 | 45.8 | 6.0622 | 3.0 | 222.0 | 18.7 | 394.63 | 2.94 |
| **4** | 0.06905 | 0.0 | 2.18 | 0.0 | 0.458 | 7.147 | 54.2 | 6.0622 | 3.0 | 222.0 | 18.7 | 396.90 | 5.33 |

# Adding the Target to the Pandas DataFrame

- Now let's add the target to this NumPy array



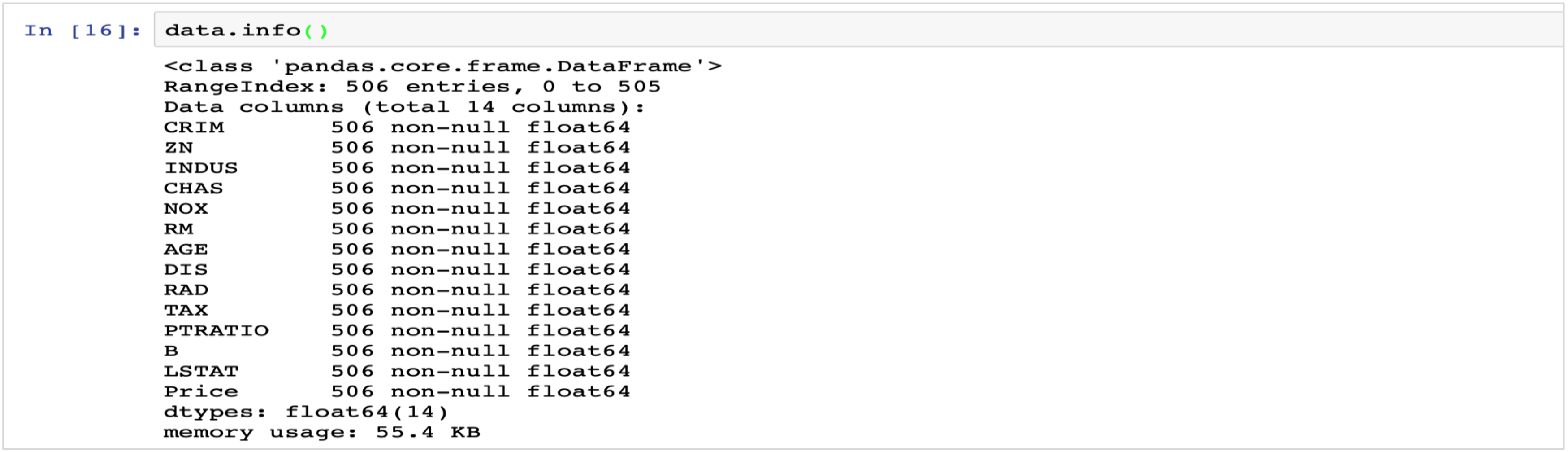
In [9]:
```
data['Price']= boston_dataset.target
data.head()
```

Out[9]:

| | CRIM | ZN | INDUS | CHAS | NOX | RM | AGE | DIS | RAD | TAX | PTRATIO | B | LSTAT | Price |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0.00632 | 18.0 | 2.31 | 0.0 | 0.538 | 6.575 | 65.2 | 4.0900 | 1.0 | 296.0 | 15.3 | 396.90 | 4.98 | 24.0 |
| **1** | 0.02731 | 0.0 | 7.07 | 0.0 | 0.469 | 6.421 | 78.9 | 4.9671 | 2.0 | 242.0 | 17.8 | 396.90 | 9.14 | 21.6 |
| **2** | 0.02729 | 0.0 | 7.07 | 0.0 | 0.469 | 7.185 | 61.1 | 4.9671 | 2.0 | 242.0 | 17.8 | 392.83 | 4.03 | 34.7 |
| **3** | 0.03237 | 0.0 | 2.18 | 0.0 | 0.458 | 6.998 | 45.8 | 6.0622 | 3.0 | 222.0 | 18.7 | 394.63 | 2.94 | 33.4 |
| **4** | 0.06905 | 0.0 | 2.18 | 0.0 | 0.458 | 7.147 | 54.2 | 6.0622 | 3.0 | 222.0 | 18.7 | 396.90 | 5.33 | 36.2 |

## Looking at the Data Structure

- Try data.info() to look at the total number of rows, the attribute's type and the number of non-null values



```
In [16]: data.info()

         <class 'pandas.core.frame.DataFrame'>
         RangeIndex: 506 entries, 0 to 505
         Data columns (total 14 columns):
         CRIM       506 non-null float64
         ZN         506 non-null float64
         INDUS      506 non-null float64
         CHAS       506 non-null float64
         NOX        506 non-null float64
         RM         506 non-null float64
         AGE        506 non-null float64
         DIS        506 non-null float64
         RAD        506 non-null float64
         TAX        506 non-null float64
         PTRATIO    506 non-null float64
         B          506 non-null float64
         LSTAT      506 non-null float64
         Price      506 non-null float64
         dtypes: float64(14)
         memory usage: 55.4 KB
```

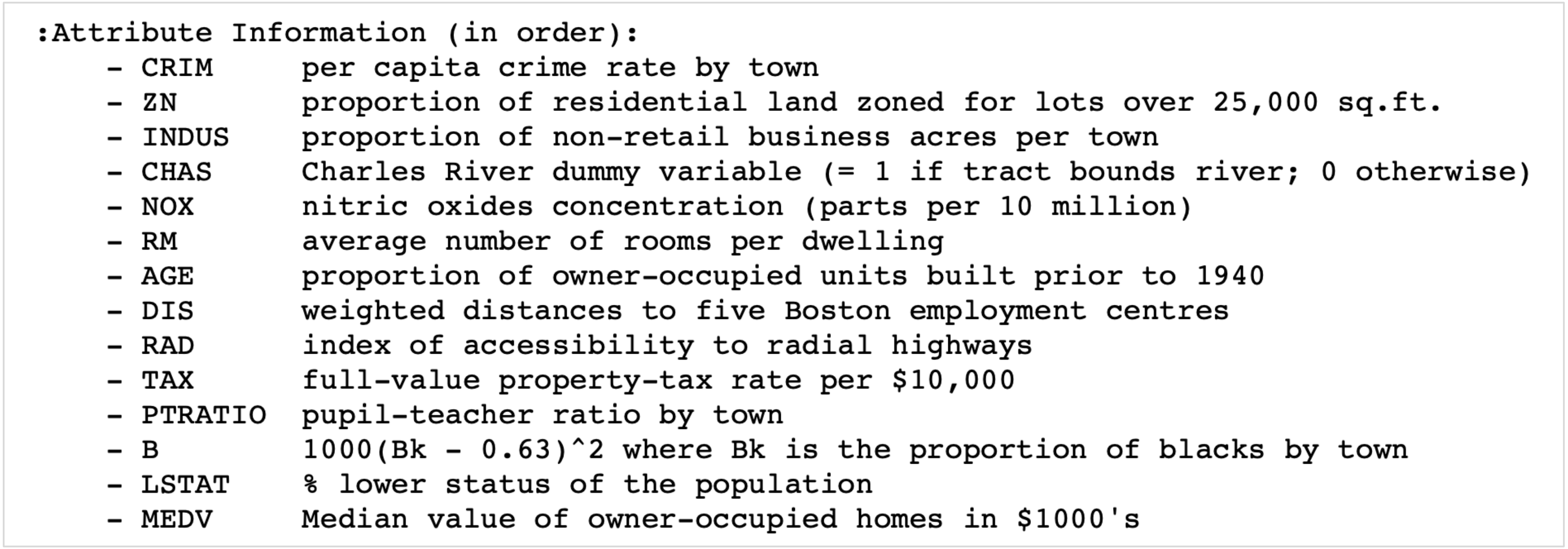
Getting the Data

## The End

## Exploring the Data

## Description of the Boston Dataset

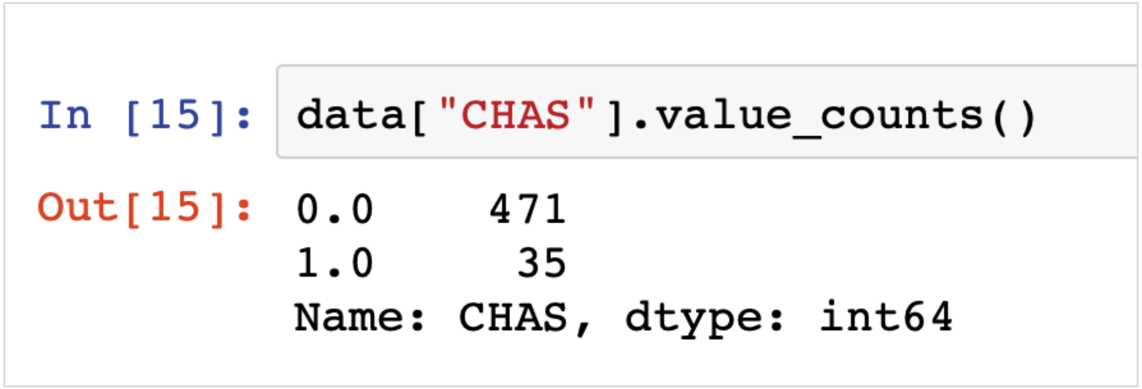
- The scikit-learn Bunch class has a DESCR element that tells you about the dataset. (Question: What is the categorical variable?)



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

# Which Feature Is Categorical?

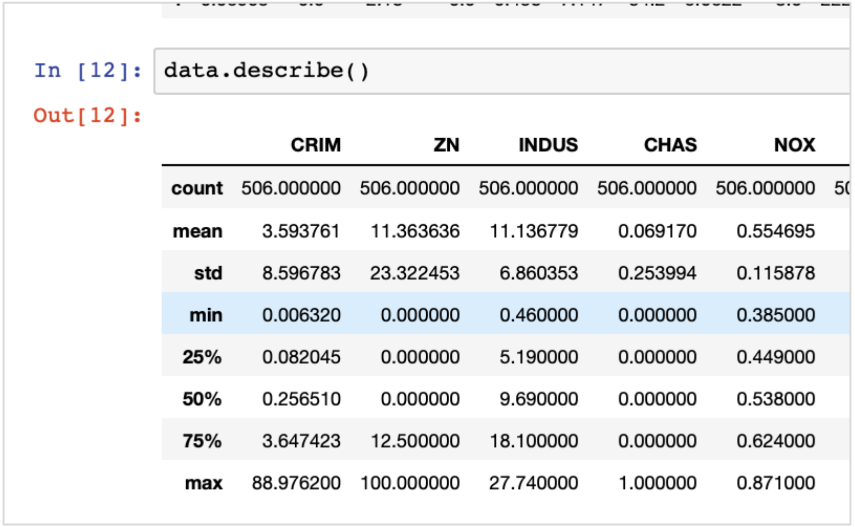
- From the description you should be able to see CHAS is a categorical variable—it is a 1 if it is on the Charles River and 0 otherwise
- Let’s see how many homes are on the Charles River:



```
In [15]: data["CHAS"].value_counts()

Out[15]: 0.0    471
         1.0     35
         Name: CHAS, dtype: int64
```

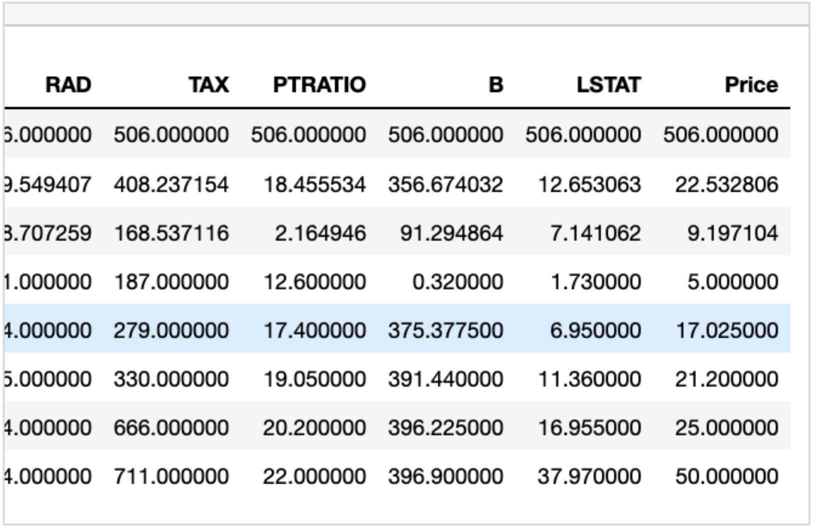
# Let's Get a Summary of the Numerical Attributes

- The pandas DataFrame describe() shows a summary of the numerical attributes. (Question: What do you notice about the max of the price?)



```
In [12]: data.describe()
Out[12]:
```

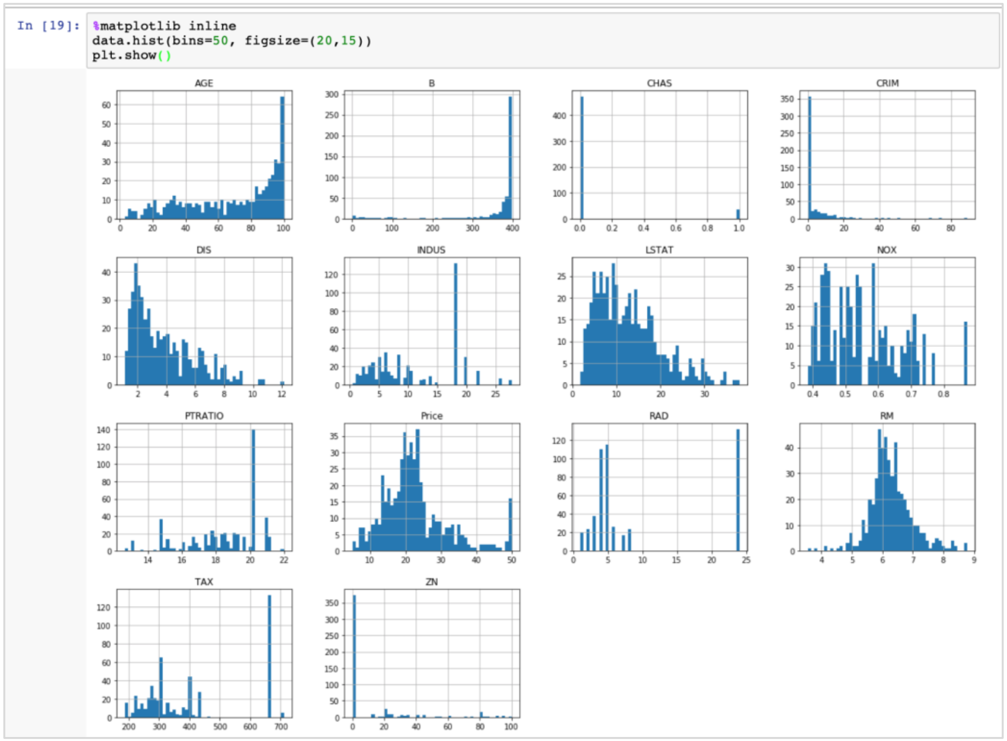
| | CRIM | ZN | INDUS | CHAS | NOX | |
|---|---|---|---|---|---|---|
| **count** | 506.000000 | 506.000000 | 506.000000 | 506.000000 | 506.000000 | 50 |
| **mean** | 3.593761 | 11.363636 | 11.136779 | 0.069170 | 0.554695 | |
| **std** | 8.596783 | 23.322453 | 6.860353 | 0.253994 | 0.115878 | |
| **min** | 0.006320 | 0.000000 | 0.460000 | 0.000000 | 0.385000 | |
| **25%** | 0.082045 | 0.000000 | 5.190000 | 0.000000 | 0.449000 | |
| **50%** | 0.256510 | 0.000000 | 9.690000 | 0.000000 | 0.538000 | |
| **75%** | 3.647423 | 12.500000 | 18.100000 | 0.000000 | 0.624000 | |
| **max** | 88.976200 | 100.000000 | 27.740000 | 1.000000 | 0.871000 | |



| RAD | TAX | PTRATIO | B | LSTAT | Price |
|---|---|---|---|---|---|
| .000000 | 506.000000 | 506.000000 | 506.000000 | 506.000000 | 506.000000 |
| .549407 | 408.237154 | 18.455534 | 356.674032 | 12.653063 | 22.532806 |
| .707259 | 168.537116 | 2.164946 | 91.294864 | 7.141062 | 9.197104 |
| 1.000000 | 187.000000 | 12.600000 | 0.320000 | 1.730000 | 5.000000 |
| 4.000000 | 279.000000 | 17.400000 | 375.377500 | 6.950000 | 17.025000 |
| 5.000000 | 330.000000 | 19.050000 | 391.440000 | 11.360000 | 21.200000 |
| 4.000000 | 666.000000 | 20.200000 | 396.225000 | 16.955000 | 25.000000 |
| 4.000000 | 711.000000 | 22.000000 | 396.900000 | 37.970000 | 50.000000 |

# Plot Histograms

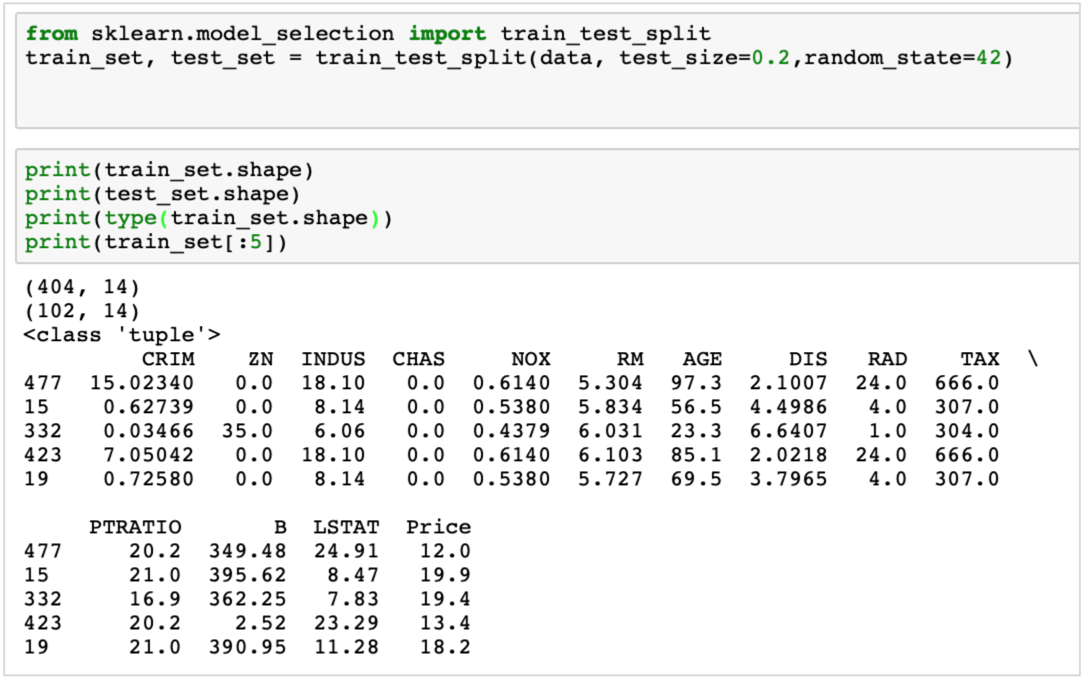
- A quick visual way to look at the characteristics of your data is to print a histogram of your numerical attributes
- You can see here that Price is capped
- You can also see CHAS is definitely a categorical variable



```
In [19]: %matplotlib inline
         data.hist(bins=50, figsize=(20,15))
         plt.show()
```

# Create and Hold Out a Test Set

- Use sklearn’s train_test_split to split data into training and test sets



```
from sklearn.model_selection import train_test_split
train_set, test_set = train_test_split(data, test_size=0.2,random_state=42)
```

```
print(train_set.shape)
print(test_set.shape)
print(type(train_set.shape))
print(train_set[:5])
```

```
(404, 14)
(102, 14)
<class 'tuple'>
         CRIM    ZN  INDUS  CHAS     NOX     RM   AGE     DIS   RAD    TAX  \
477  15.02340   0.0  18.10   0.0  0.6140  5.304  97.3  2.1007  24.0  666.0
15    0.62739   0.0   8.14   0.0  0.5380  5.834  56.5  4.4986   4.0  307.0
332   0.03466  35.0   6.06   0.0  0.4379  6.031  23.3  6.6407   1.0  304.0
423   7.05042   0.0  18.10   0.0  0.6140  6.103  85.1  2.0218  24.0  666.0
19    0.72580   0.0   8.14   0.0  0.5380  5.727  69.5  3.7965   4.0  307.0

     PTRATIO       B  LSTAT  Price
477     20.2  349.48  24.91   12.0
15      21.0  395.62   8.47   19.9
332     16.9  362.25   7.83   19.4
423     20.2    2.52  23.29   13.4
19      21.0  390.95  11.28   18.2
```

## Exploring the Data

- For regression problems it is important to look at correlation
- This is also called Pearson's r
- Coefficient range from −1 to 1
  - 1 means strong positive correlation
  - −1 means strong negative correlation
  - Close to zero means no linear correlation

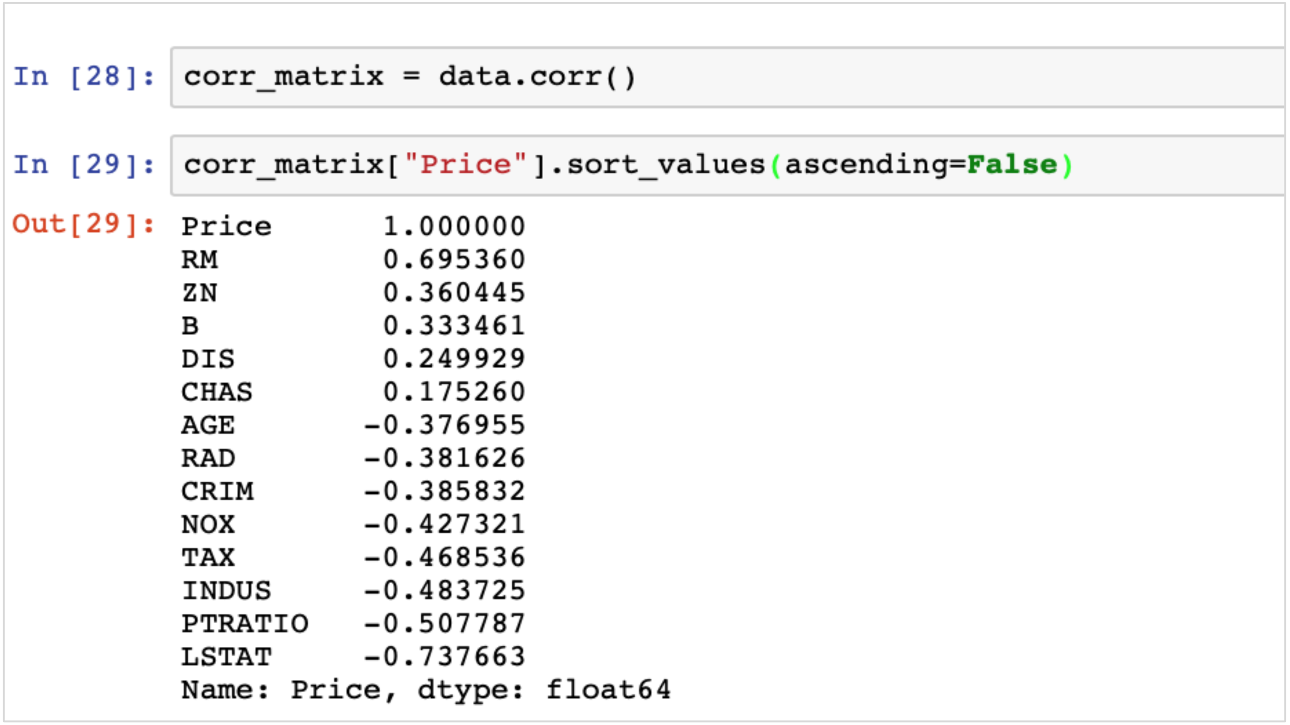
Exploring the Data

## The End

## Visualizing Correlation

# Panda’s Correlation Method

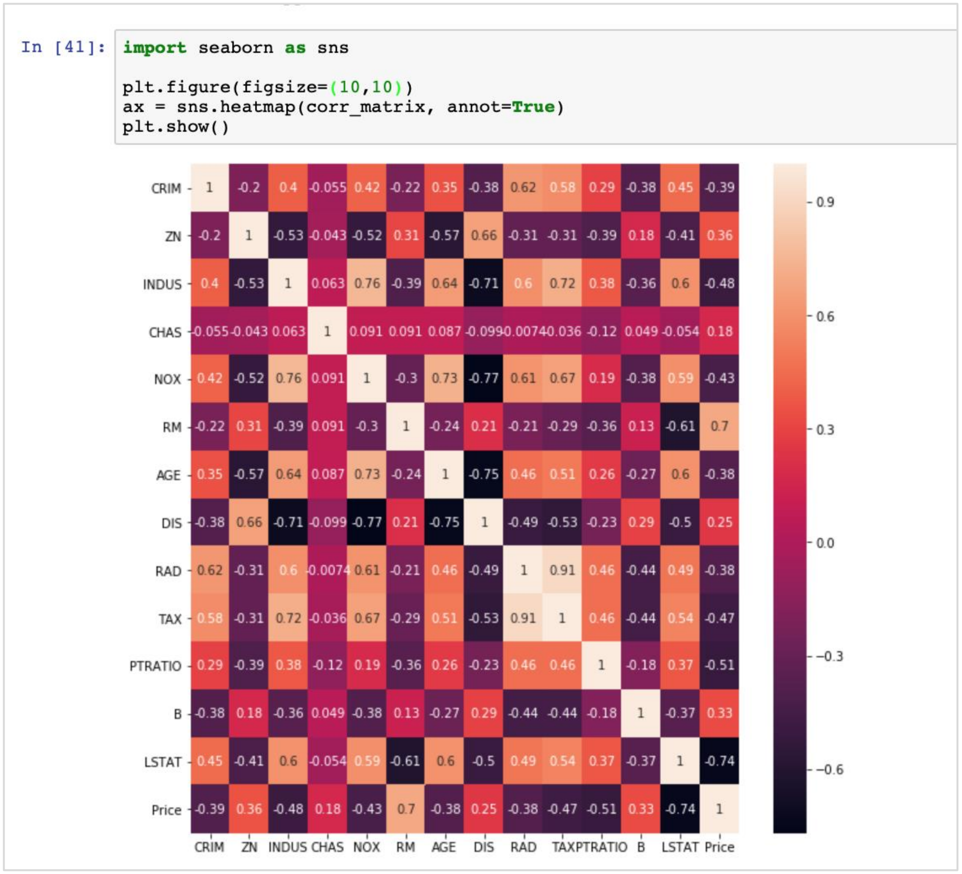
- Let’s look at correlation between our features and target (Price)
- Average number of rooms is highly correlated with price and LSTAT is negatively correlated with price



```
In [28]: corr_matrix = data.corr()

In [29]: corr_matrix["Price"].sort_values(ascending=False)

Out[29]: Price       1.000000
         RM          0.695360
         ZN          0.360445
         B           0.333461
         DIS         0.249929
         CHAS        0.175260
         AGE        -0.376955
         RAD        -0.381626
         CRIM       -0.385832
         NOX        -0.427321
         TAX        -0.468536
         INDUS      -0.483725
         PTRATIO    -0.507787
         LSTAT      -0.737663
         Name: Price, dtype: float64
```

## Visualize Correlations: sns.heatmap

- A nice way to visualize correlations is with a seaborn heatmap
- Lighter colors have higher correlation
- Darker colors have negative correlation

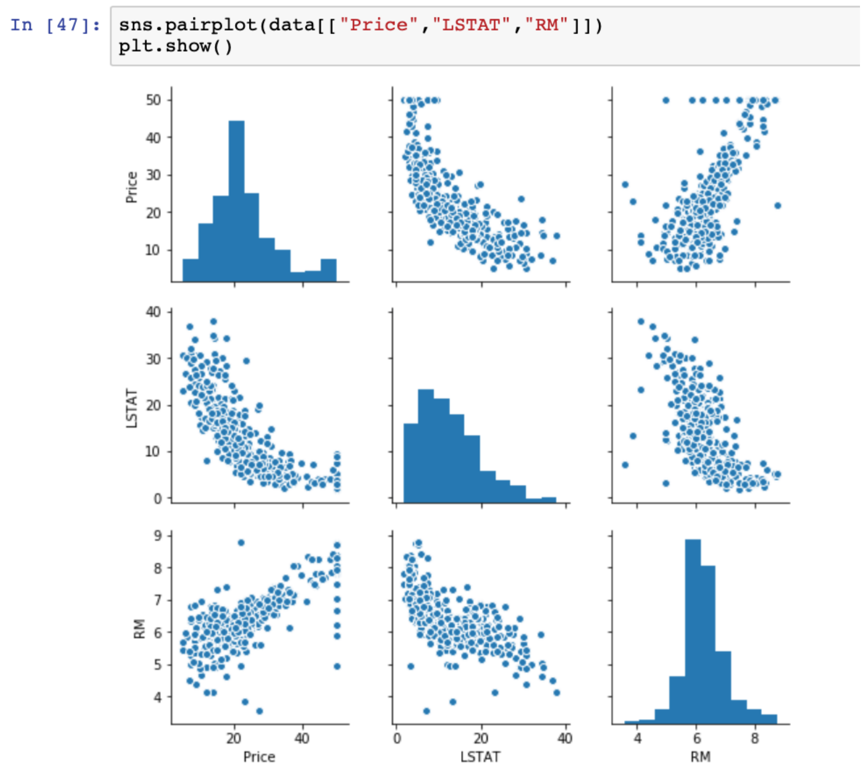


```
In [41]: import seaborn as sns

         plt.figure(figsize=(10,10))
         ax = sns.heatmap(corr_matrix, annot=True)
         plt.show()
```

| | CRIM | ZN | INDUS | CHAS | NOX | RM | AGE | DIS | RAD | TAX | PTRATIO | B | LSTAT | Price |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CRIM | 1 | -0.2 | 0.4 | -0.055 | 0.42 | -0.22 | 0.35 | -0.38 | 0.62 | 0.58 | 0.29 | -0.38 | 0.45 | -0.39 |
| ZN | -0.2 | 1 | -0.53 | -0.043 | -0.52 | 0.31 | -0.57 | 0.66 | -0.31 | -0.31 | -0.39 | 0.18 | -0.41 | 0.36 |
| INDUS | 0.4 | -0.53 | 1 | 0.063 | 0.76 | -0.39 | 0.64 | -0.71 | 0.6 | 0.72 | 0.38 | -0.36 | 0.6 | -0.48 |
| CHAS | -0.055 | -0.043 | 0.063 | 1 | 0.091 | 0.091 | 0.087 | -0.099 | -0.0074 | -0.036 | -0.12 | 0.049 | -0.054 | 0.18 |
| NOX | 0.42 | -0.52 | 0.76 | 0.091 | 1 | -0.3 | 0.73 | -0.77 | 0.61 | 0.67 | 0.19 | -0.38 | 0.59 | -0.43 |
| RM | -0.22 | 0.31 | -0.39 | 0.091 | -0.3 | 1 | -0.24 | 0.21 | -0.21 | -0.29 | -0.36 | 0.13 | -0.61 | 0.7 |
| AGE | 0.35 | -0.57 | 0.64 | 0.087 | 0.73 | -0.24 | 1 | -0.75 | 0.46 | 0.51 | 0.26 | -0.27 | 0.6 | -0.38 |
| DIS | -0.38 | 0.66 | -0.71 | -0.099 | -0.77 | 0.21 | -0.75 | 1 | -0.49 | -0.53 | -0.23 | 0.29 | -0.5 | 0.25 |
| RAD | 0.62 | -0.31 | 0.6 | -0.0074 | 0.61 | -0.21 | 0.46 | -0.49 | 1 | 0.91 | 0.46 | -0.44 | 0.49 | -0.38 |
| TAX | 0.58 | -0.31 | 0.72 | -0.036 | 0.67 | -0.29 | 0.51 | -0.53 | 0.91 | 1 | 0.46 | -0.44 | 0.54 | -0.47 |
| PTRATIO | 0.29 | -0.39 | 0.38 | -0.12 | 0.19 | -0.36 | 0.26 | -0.23 | 0.46 | 0.46 | 1 | -0.18 | 0.37 | -0.51 |
| B | -0.38 | 0.18 | -0.36 | 0.049 | -0.38 | 0.13 | -0.27 | 0.29 | -0.44 | -0.44 | -0.18 | 1 | -0.37 | 0.33 |
| LSTAT | 0.45 | -0.41 | 0.6 | -0.054 | 0.59 | -0.61 | 0.6 | -0.5 | 0.49 | 0.54 | 0.37 | -0.37 | 1 | -0.74 |
| Price | -0.39 | 0.36 | -0.48 | 0.18 | -0.43 | 0.7 | -0.38 | 0.25 | -0.38 | -0.47 | -0.51 | 0.33 | -0.74 | 1 |

# Let’s Look at Features Highly Correlated With Price

- Average number of rooms: 0.7
- Percentile lower status of the population (LSTAT): −0.74
- Pandas has a scatter_matrix or seaborn has a pairplot



```
In [47]: sns.pairplot(data[["Price","LSTAT","RM"]])
         plt.show()
```

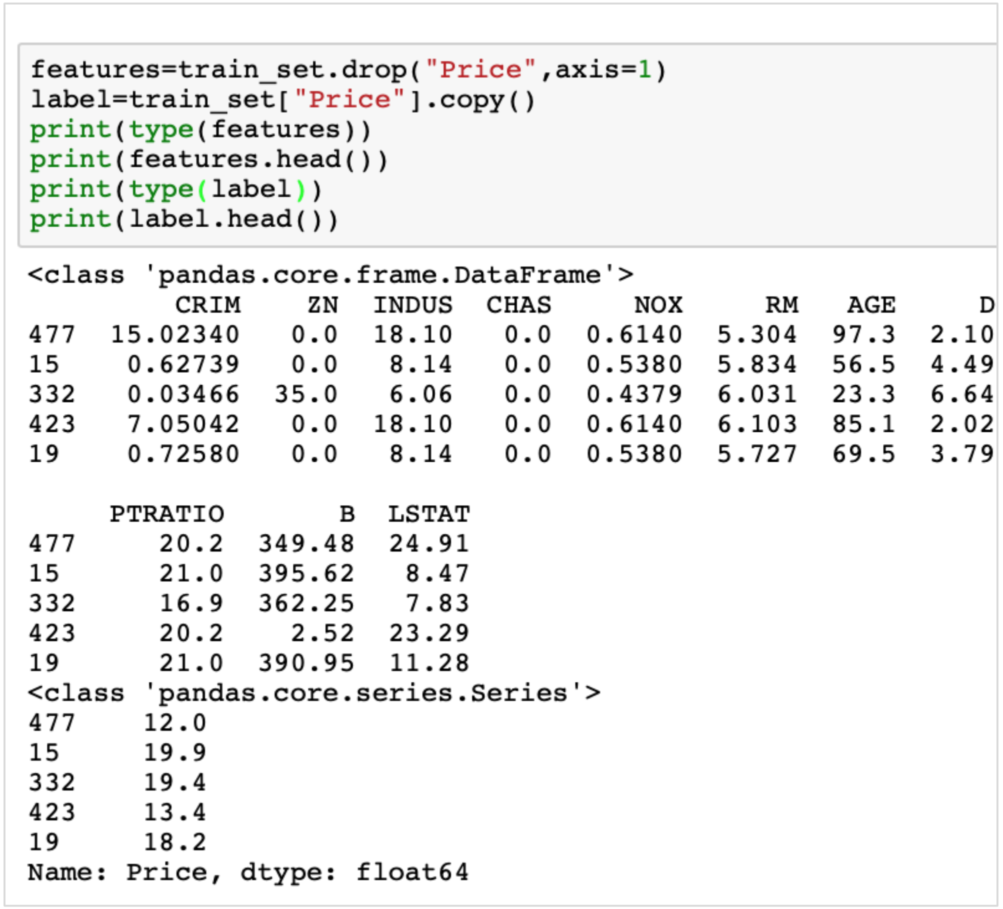
Visualizing Correlation

## The End

## Preparing the Data

# Preparing the Data

- Provide a feature set that does not include the label
- Include a label series to provide the model



```
features=train_set.drop("Price",axis=1)
label=train_set["Price"].copy()
print(type(features))
print(features.head())
print(type(label))
print(label.head())
```

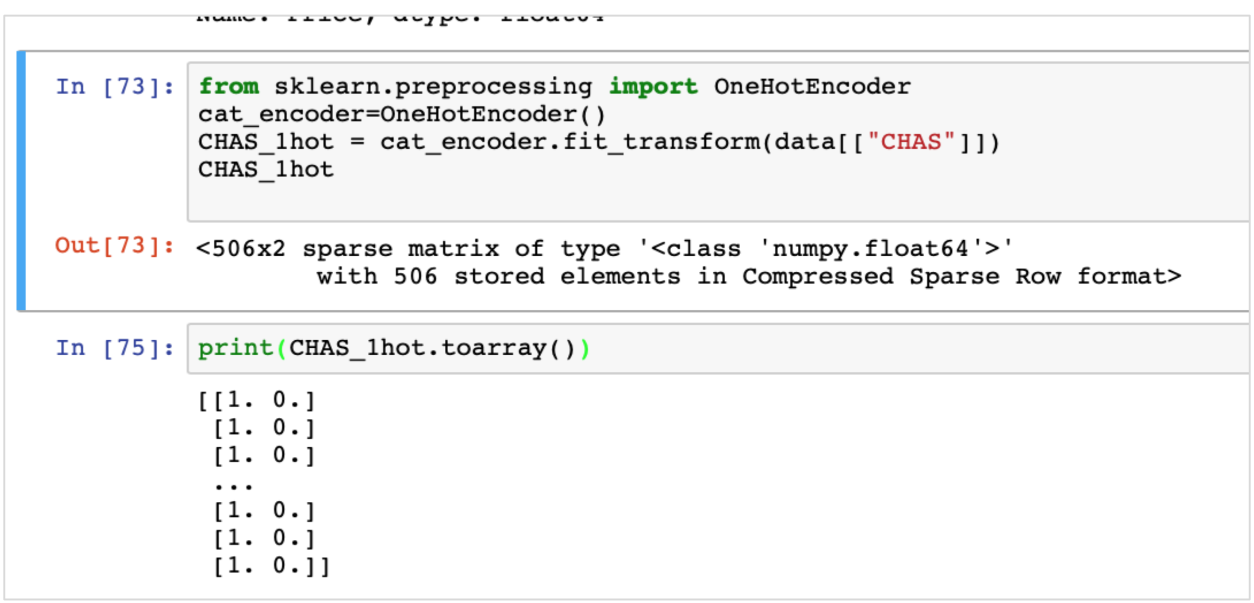
```
<class 'pandas.core.frame.DataFrame'>
         CRIM    ZN  INDUS  CHAS     NOX     RM   AGE     D
477  15.02340   0.0  18.10   0.0  0.6140  5.304  97.3  2.10
15    0.62739   0.0   8.14   0.0  0.5380  5.834  56.5  4.49
332   0.03466  35.0   6.06   0.0  0.4379  6.031  23.3  6.64
423   7.05042   0.0  18.10   0.0  0.6140  6.103  85.1  2.02
19    0.72580   0.0   8.14   0.0  0.5380  5.727  69.5  3.79

     PTRATIO       B  LSTAT
477     20.2  349.48  24.91
15      21.0  395.62   8.47
332     16.9  362.25   7.83
423     20.2    2.52  23.29
19      21.0  390.95  11.28
<class 'pandas.core.series.Series'>
477    12.0
15     19.9
332    19.4
423    13.4
19     18.2
Name: Price, dtype: float64
```

## Deal With Missing Data

- Get rid of attributes with missing data
- Get rid of examples with missing data
- Set missing data to some value
- If you chose option 3 (I’ll take the car, Drew) you will need to use an imputer
- A transformer in scikit-learn is an estimator with a fit and then a transform method
- An imputer is an example of a transformer

# Handling Categorical Variables

- Recall in our example, CHAS is 1 if it is Charles Riverfront and 0 otherwise
- So the machine learning model doesn't think these categories are ordered, we one-hot encode them so only one value will be equal to 1
- Notice this returns a sparse matrix
- You can turn it back into a NumPy array with the toArray method



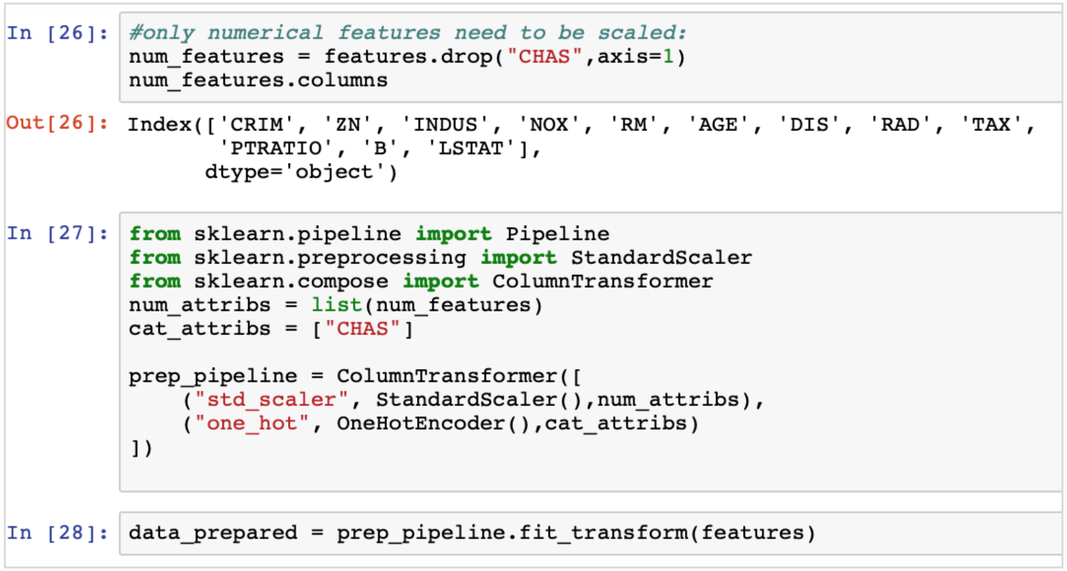
```
In [73]: from sklearn.preprocessing import OneHotEncoder
         cat_encoder=OneHotEncoder()
         CHAS_1hot = cat_encoder.fit_transform(data[["CHAS"]])
         CHAS_1hot

Out[73]: <506x2 sparse matrix of type '<class 'numpy.float64'>'
                 with 506 stored elements in Compressed Sparse Row format>

In [75]: print(CHAS_1hot.toarray())

         [[1. 0.]
          [1. 0.]
          [1. 0.]
          ...
          [1. 0.]
          [1. 0.]
          [1. 0.]]
```

## Scaling Numerical Features

- It is important to get all of the input variables in the same scale
- There are two types:
  1. Min-max scaling
  2. Standardization
- Min-max scaling forces everything to between 0 and 1
- Standardization uses the concept of unit variance
- scikit-learn has a StandardScaler method to do this

# Implementing Transformations on a Dataset

- So we need to one-hot encode our categorical variable and scale our numerical variables
- scikit-learn’s ColumnTransformer can help us with this
- It lets you specify which columns to do certain transformations on



```
In [26]: #only numerical features need to be scaled:
         num_features = features.drop("CHAS",axis=1)
         num_features.columns

Out[26]: Index(['CRIM', 'ZN', 'INDUS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX',
                'PTRATIO', 'B', 'LSTAT'],
               dtype='object')

In [27]: from sklearn.pipeline import Pipeline
         from sklearn.preprocessing import StandardScaler
         from sklearn.compose import ColumnTransformer
         num_attribs = list(num_features)
         cat_attribs = ["CHAS"]

         prep_pipeline = ColumnTransformer([
             ("std_scaler", StandardScaler(),num_attribs),
             ("one_hot", OneHotEncoder(),cat_attribs)
         ])

In [28]: data_prepared = prep_pipeline.fit_transform(features)
```

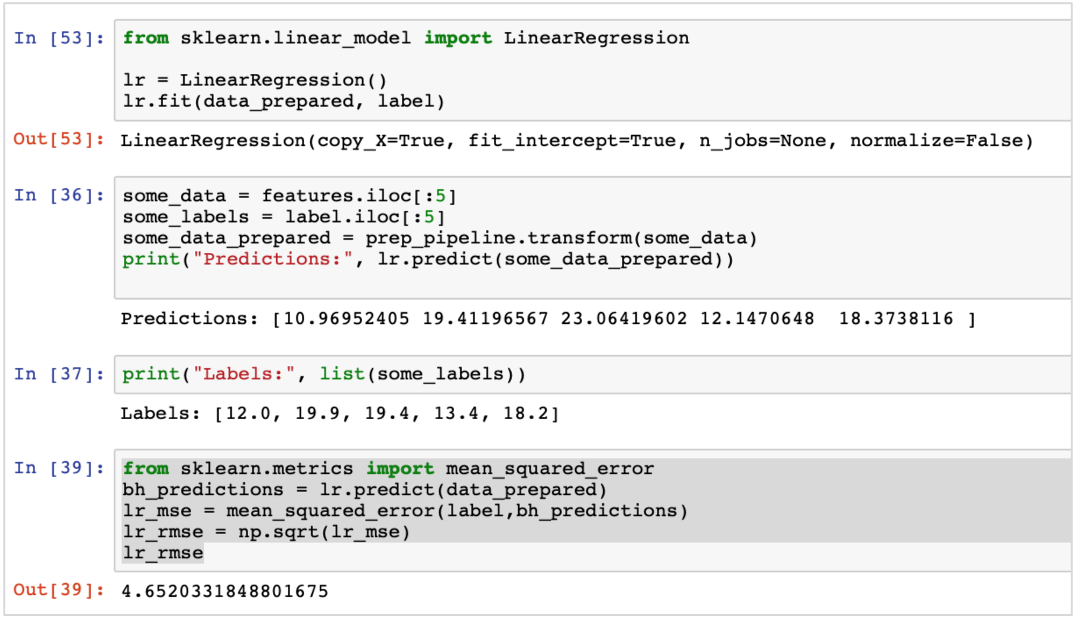
Preparing the Data

## The End

## Select and Train a Model

# Select and Train a Model

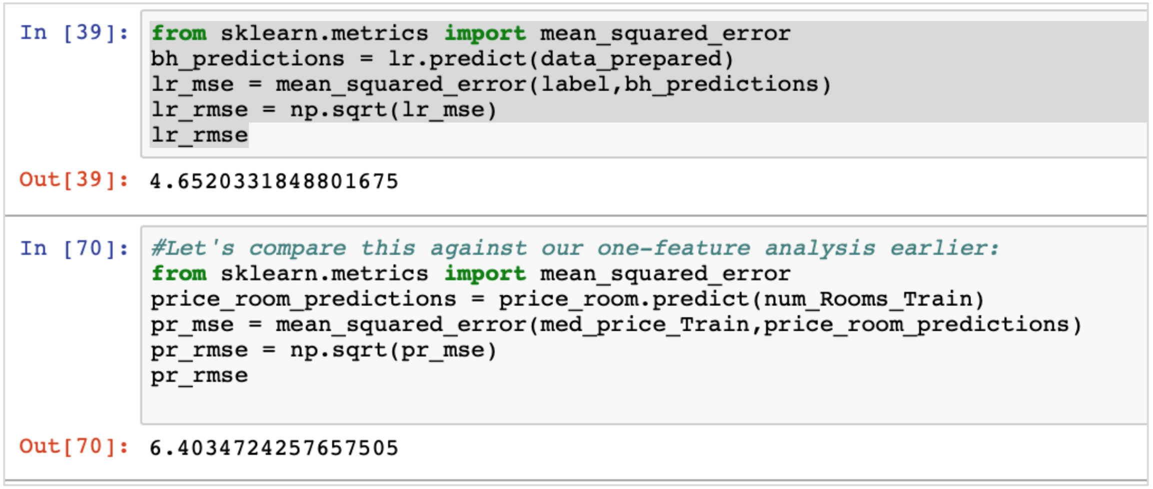
- Let’s start with a simple linear regressor
- We first fit the model to our prepared data
- We then run the predict method
- Finally, we figure out root mean square error (RMSE) using the known labels on the training set



```
In [53]: from sklearn.linear_model import LinearRegression

         lr = LinearRegression()
         lr.fit(data_prepared, label)

Out[53]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)

In [36]: some_data = features.iloc[:5]
         some_labels = label.iloc[:5]
         some_data_prepared = prep_pipeline.transform(some_data)
         print("Predictions:", lr.predict(some_data_prepared))

         Predictions: [10.96952405 19.41196567 23.06419602 12.1470648  18.3738116 ]

In [37]: print("Labels:", list(some_labels))

         Labels: [12.0, 19.9, 19.4, 13.4, 18.2]

In [39]: from sklearn.metrics import mean_squared_error
         bh_predictions = lr.predict(data_prepared)
         lr_mse = mean_squared_error(label,bh_predictions)
         lr_rmse = np.sqrt(lr_mse)
         lr_rmse

Out[39]: 4.6520331848801675
```

# Comparison With One-Feature Model

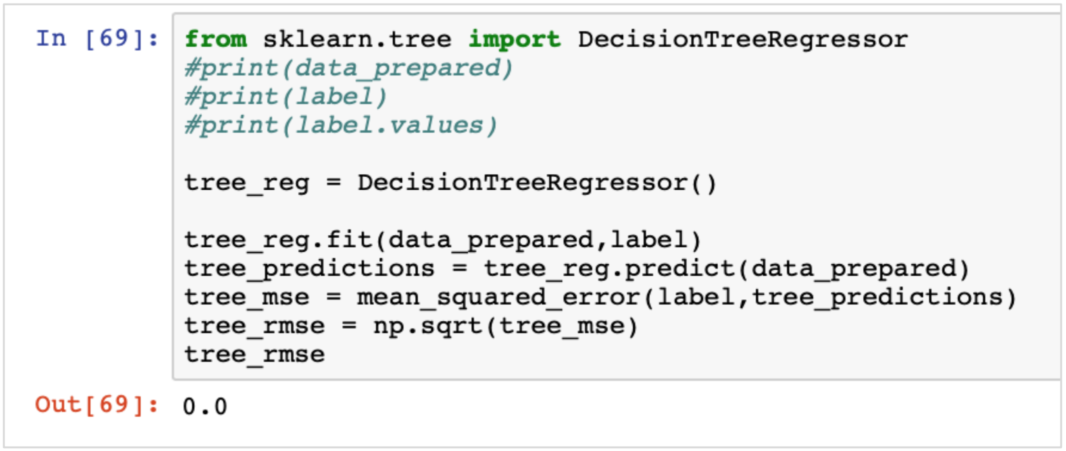
- Our RMSE is $4,600
- Comparing this against the RMSE of our model, just taking average rooms into account you can see we are doing better
- Our RMSE on the initial model is $6,400



```
In [39]: from sklearn.metrics import mean_squared_error
         bh_predictions = lr.predict(data_prepared)
         lr_mse = mean_squared_error(label,bh_predictions)
         lr_rmse = np.sqrt(lr_mse)
         lr_rmse

Out[39]: 4.6520331848801675

In [70]: #Let's compare this against our one-feature analysis earlier:
         from sklearn.metrics import mean_squared_error
         price_room_predictions = price_room.predict(num_Rooms_Train)
         pr_mse = mean_squared_error(med_price_Train,price_room_predictions)
         pr_rmse = np.sqrt(pr_mse)
         pr_rmse

Out[70]: 6.4034724257657505
```

# Let’s Try a More Complex Model

- Let’s implement a decision tree regressor
- Decision trees are known for discovering non-linear relationships in the data
- Again, this is the process:
  - Fit
  - Train
  - Evaluate
- An RMSE of 0? Do you think we might be overfitting?



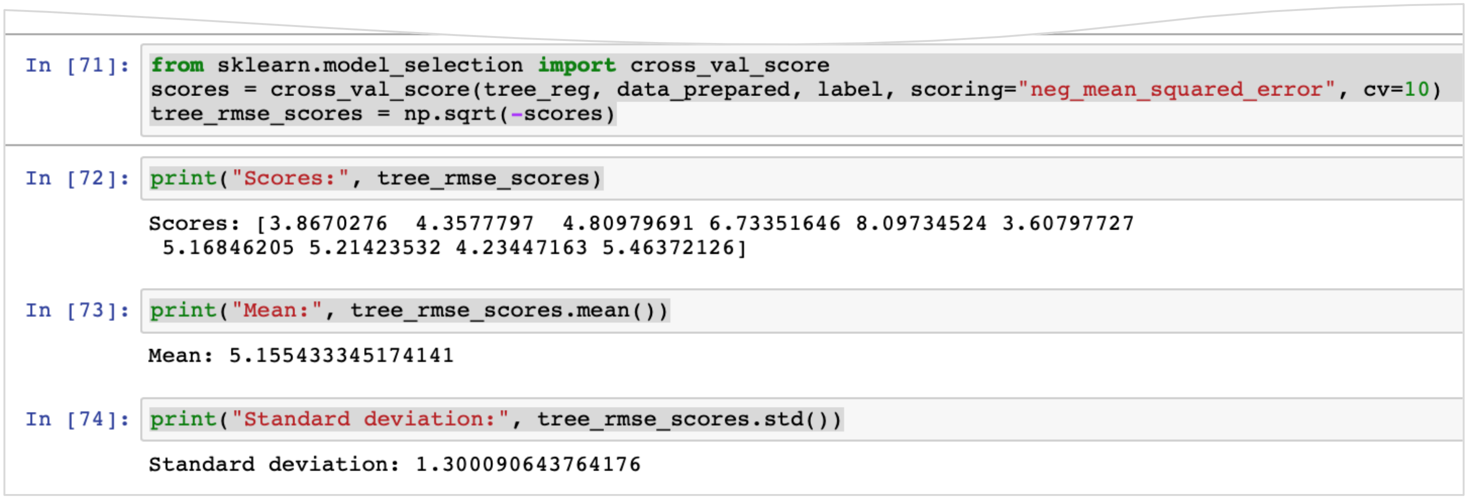
```
In [69]: from sklearn.tree import DecisionTreeRegressor
         #print(data_prepared)
         #print(label)
         #print(label.values)

         tree_reg = DecisionTreeRegressor()

         tree_reg.fit(data_prepared,label)
         tree_predictions = tree_reg.predict(data_prepared)
         tree_mse = mean_squared_error(label,tree_predictions)
         tree_rmse = np.sqrt(tree_mse)
         tree_rmse

Out[69]: 0.0
```

## Combatting Overfitting: Cross-Validation

- K-fold cross-validation
  - Randomly splits training data into K folds
  - Trains and evaluates K number of times
  - Picks a different fold for evaluation every time while training on the other ones
  - Result is an array containing 10 evaluation scores

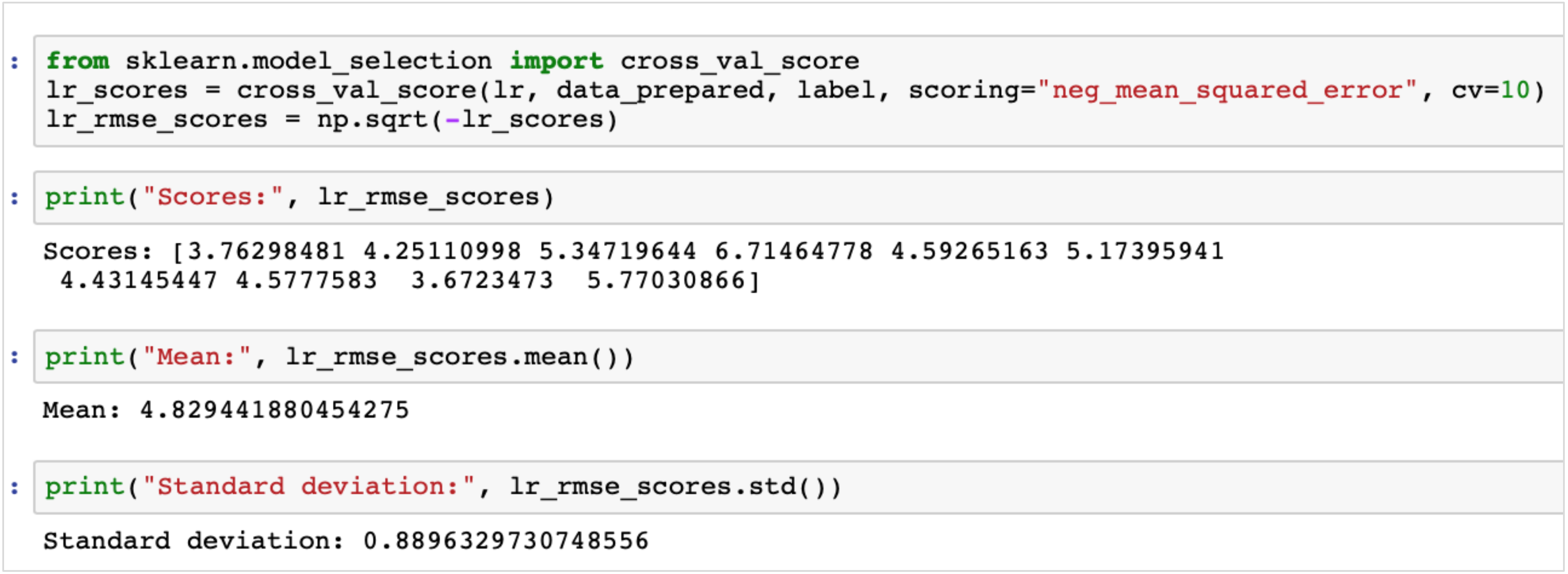
# Here Is What Cross-Validation of the Decision Tree Shows Us

- Recall our linear regression RMSE on the training set was 4.65



```
In [71]: from sklearn.model_selection import cross_val_score
         scores = cross_val_score(tree_reg, data_prepared, label, scoring="neg_mean_squared_error", cv=10)
         tree_rmse_scores = np.sqrt(-scores)

In [72]: print("Scores:", tree_rmse_scores)

         Scores: [3.8670276  4.3577797  4.80979691 6.73351646 8.09734524 3.60797727
          5.16846205 5.21423532 4.23447163 5.46372126]

In [73]: print("Mean:", tree_rmse_scores.mean())

         Mean: 5.155433345174141

In [74]: print("Standard deviation:", tree_rmse_scores.std())

         Standard deviation: 1.300090643764176
```

# Let’s Do Cross-Validation on the Linear Model As Well



```
from sklearn.model_selection import cross_val_score
lr_scores = cross_val_score(lr, data_prepared, label, scoring="neg_mean_squared_error", cv=10)
lr_rmse_scores = np.sqrt(-lr_scores)
```

```
print("Scores:", lr_rmse_scores)
```

```
Scores: [3.76298481 4.25110998 5.34719644 6.71464778 4.59265163 5.17395941
 4.43145447 4.5777583  3.6723473  5.77030866]
```

```
print("Mean:", lr_rmse_scores.mean())
```

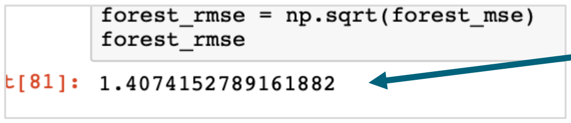
```
Mean: 4.829441880454275
```

```
print("Standard deviation:", lr_rmse_scores.std())
```

```
Standard deviation: 0.8896329730748556
```

# Let’s Try Another Type of Model

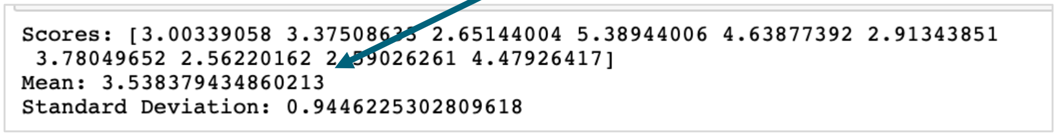
- RandomForestRegressor
  - Works by training many decision trees on random subset of features
  - Averages out their predictions
  - This is an example of ensemble learning, building a model with many other models as input



```
forest_rmse = np.sqrt(forest_mse)
forest_rmse
```

```
t[81]: 1.4074152789161882
```

**Much better!**



```
Scores: [3.00339058 3.37508635 2.65144004 5.38944006 4.63877392 2.91343851
 3.78049652 2.56220162 2.59026261 4.47926417]
Mean: 3.538379434860213
Standard Deviation: 0.9446225302809618
```

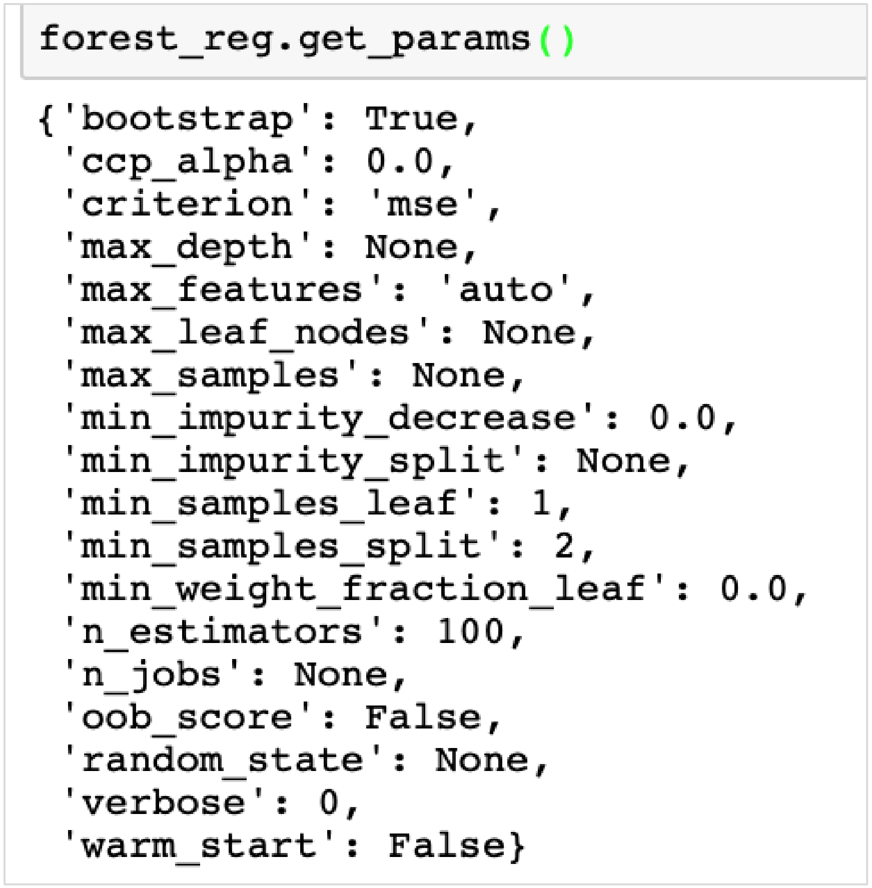
Select and Train a Model

## The End

## Tune Your Model

## Tune Your Model

- Different models have settings called hyper-parameters that you use to tune the architecture of the models
- Models in scikit-learn have a get_params method to show you what these settings are
- Let’s look at these for our forest regressor



```
forest_reg.get_params()
```

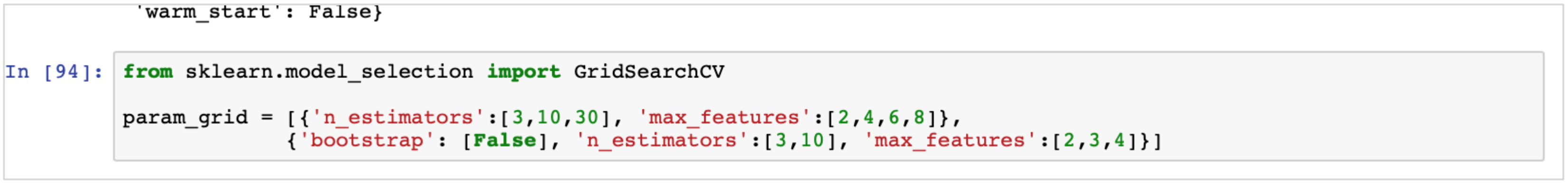
```
{'bootstrap': True,
 'ccp_alpha': 0.0,
 'criterion': 'mse',
 'max_depth': None,
 'max_features': 'auto',
 'max_leaf_nodes': None,
 'max_samples': None,
 'min_impurity_decrease': 0.0,
 'min_impurity_split': None,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'n_estimators': 100,
 'n_jobs': None,
 'oob_score': False,
 'random_state': None,
 'verbose': 0,
 'warm_start': False}
```

## scikit-learn GridSearch Utility

- To explore combinations of hyper-parameters that perform better, you can use scikit-learn’s GridSearchCV
- You tell it which hyper-parameters to experiment with
- It will evaluate combinations and values you give it with cross-validation

## Import the Constructor and Build a param_grid

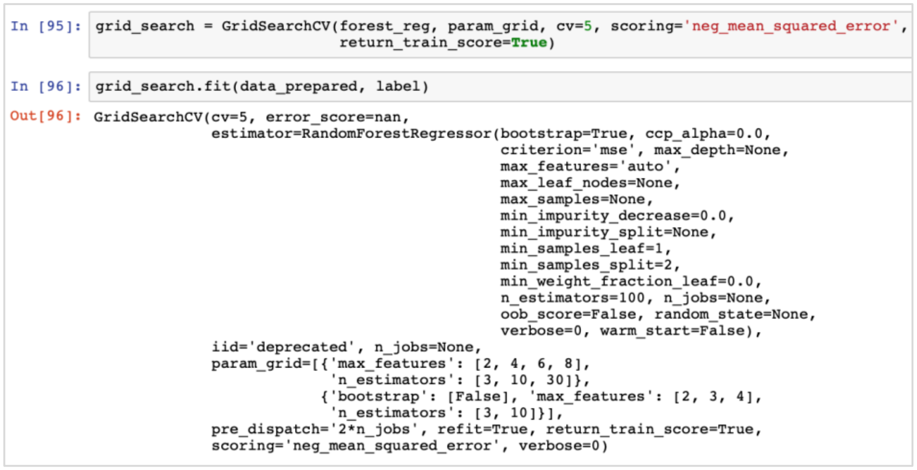
- param_grid is a list of dictionary items that define hyper-parameter combinations you want to test
- In this case, we are looking at 12 combinations of n_estimators and max_features and another one with six combinations of those hyper-parameters with bootstrapping off
- The total number of experiments will be 18



```
         'warm_start': False}

In [94]: from sklearn.model_selection import GridSearchCV

         param_grid = [{'n_estimators':[3,10,30], 'max_features':[2,4,6,8]},
                       {'bootstrap': [False], 'n_estimators':[3,10], 'max_features':[2,3,4]}]
```

# Instantiate a grid_search Object

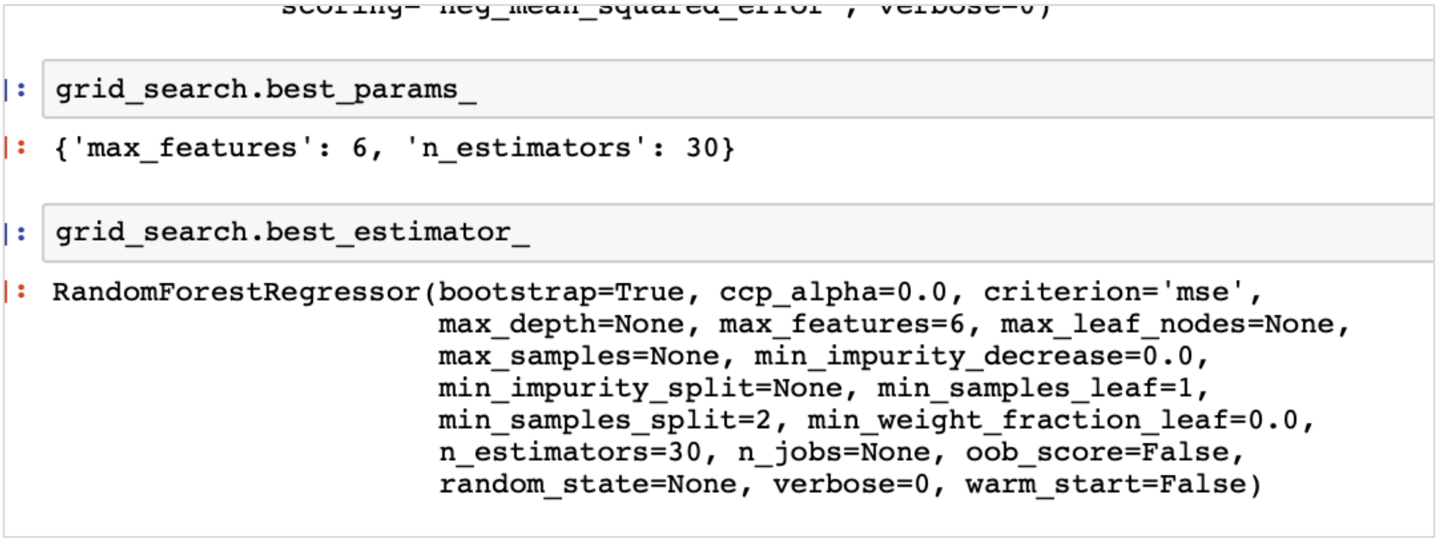
- Inputs include:
  - Your model
  - Your parameter grid
  - The number of folds for cross-validation
  - Your scoring value
- GridSearchCV is like an estimator in the fact that you can fit, predict, score, and transform your data with it



```
In [95]: grid_search = GridSearchCV(forest_reg, param_grid, cv=5, scoring='neg_mean_squared_error',
                                    return_train_score=True)

In [96]: grid_search.fit(data_prepared, label)

Out[96]: GridSearchCV(cv=5, error_score=nan,
                      estimator=RandomForestRegressor(bootstrap=True, ccp_alpha=0.0,
                                                      criterion='mse', max_depth=None,
                                                      max_features='auto',
                                                      max_leaf_nodes=None,
                                                      max_samples=None,
                                                      min_impurity_decrease=0.0,
                                                      min_impurity_split=None,
                                                      min_samples_leaf=1,
                                                      min_samples_split=2,
                                                      min_weight_fraction_leaf=0.0,
                                                      n_estimators=100, n_jobs=None,
                                                      oob_score=False, random_state=None,
                                                      verbose=0, warm_start=False),
                      iid='deprecated', n_jobs=None,
                      param_grid=[{'max_features': [2, 4, 6, 8],
                                   'n_estimators': [3, 10, 30]},
                                  {'bootstrap': [False], 'max_features': [2, 3, 4],
                                   'n_estimators': [3, 10]}],
                      pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
                      scoring='neg_mean_squared_error', verbose=0)
```

# Attributes Available for grid_search

- To come up with the best paramater combination use .best_params_
- To list the best model use .best_estimator_



```
             scoring='neg_mean_squared_error', verbose=0)
```

```
]: grid_search.best_params_
```

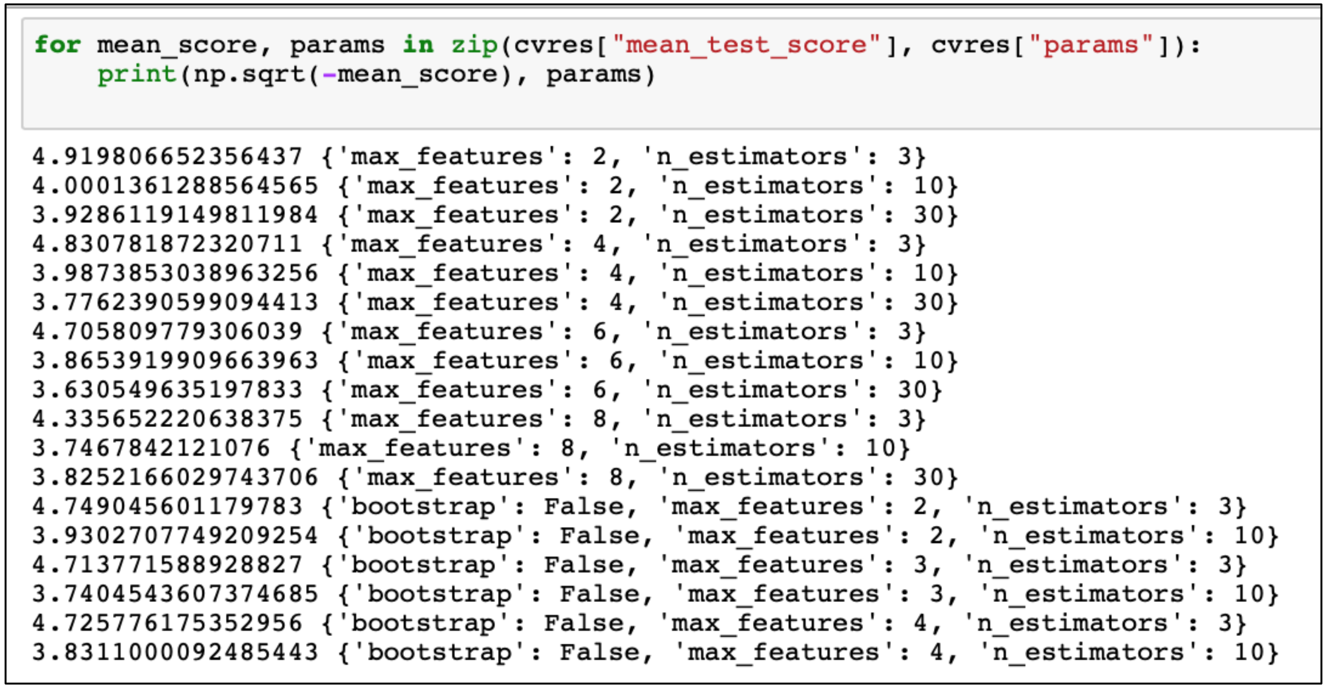
```
]: {'max_features': 6, 'n_estimators': 30}
```

```
]: grid_search.best_estimator_
```

```
]: RandomForestRegressor(bootstrap=True, ccp_alpha=0.0, criterion='mse',
                         max_depth=None, max_features=6, max_leaf_nodes=None,
                         max_samples=None, min_impurity_decrease=0.0,
                         min_impurity_split=None, min_samples_leaf=1,
                         min_samples_split=2, min_weight_fraction_leaf=0.0,
                         n_estimators=30, n_jobs=None, oob_score=False,
                         random_state=None, verbose=0, warm_start=False)
```

# Viewing Results

- .cv_results_ returns a dictionary with keys as column headers and values as columns from across all the hyperparameter combinations
- You can see here the lowest score is 3.63 with 6 max_features and 30 n_estimators



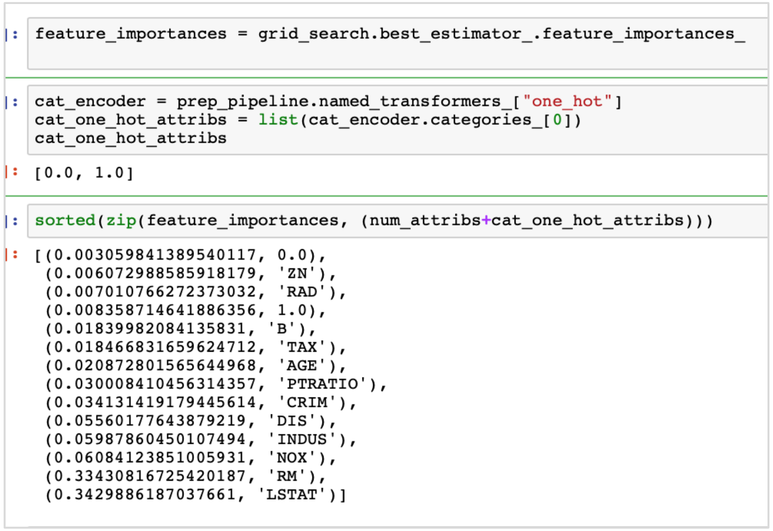
```
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(np.sqrt(-mean_score), params)
```

```
4.919806652356437 {'max_features': 2, 'n_estimators': 3}
4.0001361288564565 {'max_features': 2, 'n_estimators': 10}
3.9286119149811984 {'max_features': 2, 'n_estimators': 30}
4.830781872320711 {'max_features': 4, 'n_estimators': 3}
3.9873853038963256 {'max_features': 4, 'n_estimators': 10}
3.7762390599094413 {'max_features': 4, 'n_estimators': 30}
4.705809779306039 {'max_features': 6, 'n_estimators': 3}
3.8653919909663963 {'max_features': 6, 'n_estimators': 10}
3.630549635197833 {'max_features': 6, 'n_estimators': 30}
4.335652220638375 {'max_features': 8, 'n_estimators': 3}
3.7467842121076 {'max_features': 8, 'n_estimators': 10}
3.8252166029743706 {'max_features': 8, 'n_estimators': 30}
4.749045601179783 {'bootstrap': False, 'max_features': 2, 'n_estimators': 3}
3.9302707749209254 {'bootstrap': False, 'max_features': 2, 'n_estimators': 10}
4.713771588928827 {'bootstrap': False, 'max_features': 3, 'n_estimators': 3}
3.7404543607374685 {'bootstrap': False, 'max_features': 3, 'n_estimators': 10}
4.725776175352956 {'bootstrap': False, 'max_features': 4, 'n_estimators': 3}
3.8311000092485443 {'bootstrap': False, 'max_features': 4, 'n_estimators': 10}
```

# Inspect Your Results

- Which features are important?
- This will show it for you: .best_estimator_.feature_importances_
- Let’s look at a sorted list of the most important features
    - We need to provide a list of numerical attributes and one-hot encoded categories in order to do this
- Not surprisingly, notice the most important features were those with high correlation in our initial analysis
- We could use this information to implement a simpler model

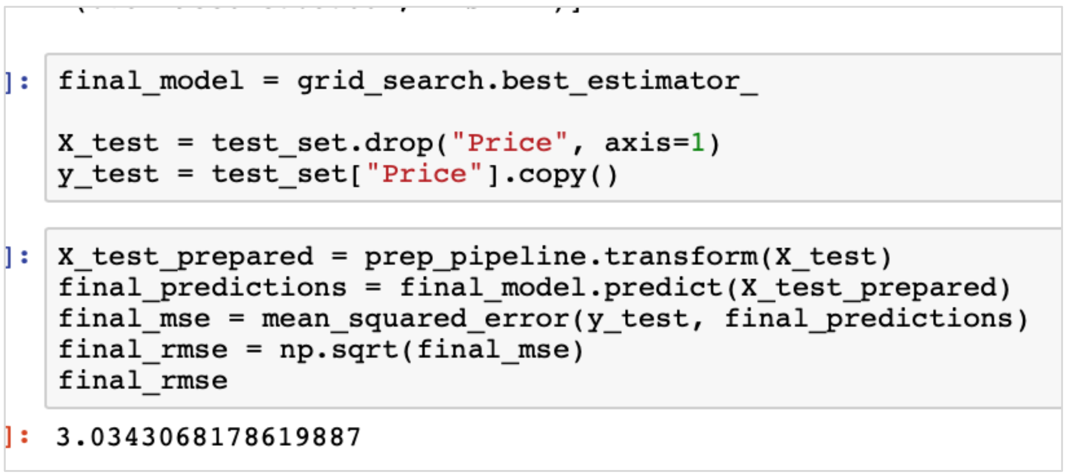
Simpler = better



```
]: feature_importances = grid_search.best_estimator_.feature_importances_

]: cat_encoder = prep_pipeline.named_transformers_["one_hot"]
   cat_one_hot_attribs = list(cat_encoder.categories_[0])
   cat_one_hot_attribs

]: [0.0, 1.0]

]: sorted(zip(feature_importances, (num_attribs+cat_one_hot_attribs)))

]: [(0.003059841389540117, 0.0),
    (0.006072988585918179, 'ZN'),
    (0.007010766272373032, 'RAD'),
    (0.008358714641886356, 1.0),
    (0.01839982084135831, 'B'),
    (0.018466831659624712, 'TAX'),
    (0.020872801565644968, 'AGE'),
    (0.030008410456314357, 'PTRATIO'),
    (0.034131419179445614, 'CRIM'),
    (0.05560177643879219, 'DIS'),
    (0.05987860450107494, 'INDUS'),
    (0.06084123851005931, 'NOX'),
    (0.33430816725420187, 'RM'),
    (0.3429886187037661, 'LSTAT')]
```

# Evaluate Your Best Model on Your Test Set

- You need to perform the same data prep on the test set to feed it to your model
- To prepare your data, remember to do just a transform, not a fit_transform
- From there, it is the same evaluation steps you did on your training data



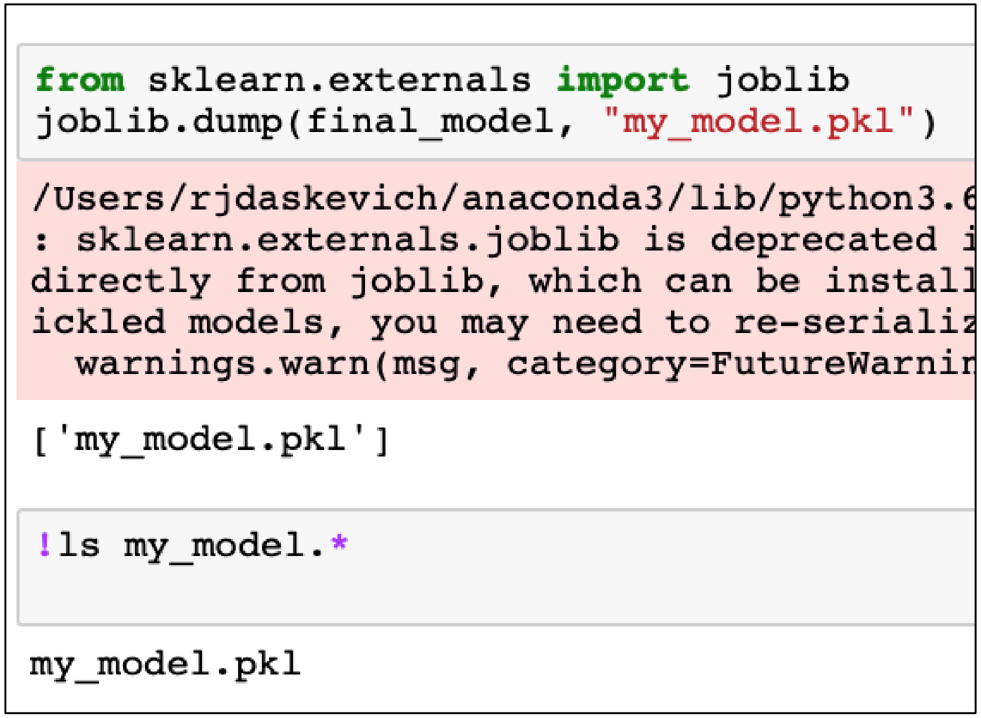
```
]: final_model = grid_search.best_estimator_

   X_test = test_set.drop("Price", axis=1)
   y_test = test_set["Price"].copy()
```

```
]: X_test_prepared = prep_pipeline.transform(X_test)
   final_predictions = final_model.predict(X_test_prepared)
   final_mse = mean_squared_error(y_test, final_predictions)
   final_rmse = np.sqrt(final_mse)
   final_rmse
```

```
]: 3.0343068178619887
```

# Save Your Model for Future Use

- You can use Python’s PICKLE object serializer to save your model
- You can also use joblib
  - It has a dump method to serialize and save your model
  - It has a load method to read it back in and make it usable



```
from sklearn.externals import joblib
joblib.dump(final_model, "my_model.pkl")
```

```
/Users/rjdaskevich/anaconda3/lib/python3.6
: sklearn.externals.joblib is deprecated i
directly from joblib, which can be install
ickled models, you may need to re-serializ
  warnings.warn(msg, category=FutureWarnin
```

```
['my_model.pkl']
```

```
!ls my_model.*
```

```
my_model.pkl
```

Tune Your Model

## The End

## Present Your Solution

## Present Your Solution

- What have you learned?
- What worked?
- What didn’t work?
- What assumptions were made?
- What limitations did you have?
- What are your final recommendations?
- Don’t forget to provide visualizations and clear results

## Launch Production System

- Hook your system up to production input data
- Check prediction performance at regular intervals
- Look at input data quality
- Make sure you schedule time to retrain your models regularly

Present Your Solution

## The End