Conceptos basicos

Scientifics computing libraries in Python

1. Scientifics Computing Libraries

- **Pandas** (Data structures & tools)
- **NumPy** (Arrays & matrices)
- **SciPy** (Integrals, solving differential equations, optimization)

Visualization libraries in Python

2. Visualization Libraries

- **Matplotlib** (plots & graphs, most popular)
- **Seaborn** (plots: heat maps, time series, violin plots)

Algorithmic libraries in Python

3. Algorithmic Libraries

- **Scikit-learn** (Machine Learning : Regression, classification, and so on)
- **Statsmodels** (Explore data, estimate statistical models, and perform statistical tests)

Importing a CSV without a header

```
import pandas as pd

url = "https://archive.ics.uci.edu/ml/machine-learning-databases/autos/imports-85.data"

df = pd.read_csv(url, header = None)
```

Printing the dataframe in Python

- **df** prints the entire dataframe (not recommended for large datasets)
- **df.head(n)** to show the first *n* rows of data frame
- **df.tail(n)** shows the bottom *n* rows of data frame

df.head()

n=5

	0	1	2	3	4	5	6	7	8	9	...	16	17	18	19	20	21	22	23	24	25
0	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	9.0	111	5000	21	27	13495
1	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	9.0	111	5000	21	27	16500
2	1	?	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...	152	mpfi	2.68	3.47	9.0	154	5000	19	26	16500
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...	109	mpfi	3.19	3.40	10.0	102	5500	24	30	13950
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...	136	mpfi	3.19	3.40	8.0	115	5500	18	22	17450

Adding headers

- Replace default header (by df.columns = headers)

```
headers = ["symboling","normalized-losses","make","fuel-type","aspiration", "num-of-doors","body-style",
         "drive-wheels","engine-location","wheel-base", "length","width","height","curb-weight","engine-type",
         "num-of-cylinders", "engine-size","fuel-system","bore","stroke","compression-ratio","horsepower", "peak-rpm","city-mpg","highway-mpg","price"]
```

```
df.columns=headers
```

Exporting to different formats in Python

Data Format	Read	Save
csv	pd.read_csv()	df.to_csv()
json	pd.read_json()	df.to_json()
Excel	pd.read_excel()	df.to_excel()
sql	pd.read_sql()	df.to_sql()

Basic insights from the data: Data types

Pandas Type	Native Python Type	Description
object	string	numbers and strings
int64	int	numeric characters
float64	float	numeric characters with decimals
Datetime64, timedelta[ns]	N/A (but see the datetime module in Python's standard library	time data

Basic insights from the data: Data types

- In pandas, we use `dataframe.dtypes` to check data types

```
df.dtypes
```

```
symboling             int64
normalized-losses    object
make                 object
fuel-type            object
aspiration           object
num-of-doors         object
body-style           object
drive-wheels         object
engine-location      object
wheel-base          float64
length              float64
width               float64
height              float64
curb-weight           int64
engine-type          object
num-of-cylinders     object
engine-size           int64
fuel-system          object
bore                 object
stroke               object
compression-ratio   float64
horsepower           object
peak-rpm             object
city-mpg              int64
highway-mpg           int64
price                object
dtype: object
```

dataframe.describe()

- Returns a statistical summary

```
df.describe()
```

	symboling	wheel-base	length	width	height	curb-weight	engine-size	compression-ratio	city-mpg	highway-mpg
count	205.000000	205.000000	205.000000	205.000000	205.000000	205.000000	205.000000	205.000000	205.000000	205.000000
mean	0.834146	98.756585	174.049268	65.907805	53.724878	2555.565854	126.907317	10.142537	25.219512	30.751220
std	1.245307	6.021776	12.337289	2.145204	2.443522	520.680204	41.642693	3.972040	6.542142	6.886443
min	-2.000000	86.600000	141.100000	60.300000	47.800000	1488.000000	61.000000	7.000000	13.000000	16.000000
25%	0.000000	94.500000	166.300000	64.100000	52.000000	2145.000000	97.000000	8.600000	19.000000	25.000000
50%	1.000000	97.000000	173.200000	65.500000	54.100000	2414.000000	120.000000	9.000000	24.000000	30.000000
75%	2.000000	102.400000	183.100000	66.900000	55.500000	2935.000000	141.000000	9.400000	30.000000	34.000000
max	3.000000	120.900000	208.100000	72.300000	59.800000	4066.000000	326.000000	23.000000	49.000000	54.000000

dataframe.describe (include="all")

- Provides full summary statistics

```
df.describe(include="all")
```

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	engine-size	fuel-system	bore	stroke
count	205.000000	205	205	205	205	205	205	205	205	205.000000	...	205.000000	205	205	205
unique	NaN	52	22	2	2	3	5	3	2	NaN	...	NaN	8	39	37
top	NaN	?	toyota	gas	std	four	sedan	fwd	front	NaN	...	NaN	mpfi	3.62	3.40
freq	NaN	41	32	185	168	114	96	120	202	NaN	...	NaN	94	23	20
mean	0.834146	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	98.756585	...	126.907317	NaN	NaN	NaN
std	1.245307	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	6.021776	...	41.642693	NaN	NaN	NaN
min	-2.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	86.600000	...	61.000000	NaN	NaN	NaN
25%	0.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	94.500000	...	97.000000	NaN	NaN	NaN
50%	1.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	97.000000	...	120.000000	NaN	NaN	NaN
75%	2.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	102.400000	...	141.000000	NaN	NaN	NaN
max	3.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	120.900000	...	326.000000	NaN	NaN	NaN

Writing code using DB-API

```
from dmodule import connect

#Create connection object
connection = connect( 'databasename' , 'usermame' , 'pswd' )

#Create a cursor object
cursor = connection.cursor ( )

#Run queries
cursor.execute( 'select * from mytable' )
results = cursor.fetchall ( )

#Free resources
cursor.close ( )
connection.close ( )
```

Manipulación

How to drop missing values in Python

- Use dataframes.dropna():

highway-mpg	price
...	...
20	23875
22	NaN
29	16430
...	...

→

highway-mpg	price
...	...
20	23875
29	16430
...	...

axis=**0** drops the entire row
axis=**1** drops the entire column

```
df.dropna(subset=["price"], axis=0, inplace = True)
df = df.dropna(subset=["price"], axis=0)
```

How to replace missing values in Python

Use dataframe.replace(missing_value, new_value):

normalized-losses	make
...	...
164	audi
164	audi
NaN	audi

→

normalized-losses	make
...	...
164	audi
164	audi
162	audi
158	audi
...	...

mean = df["normalized-losses"].mean()
df["normalized-losses"].replace(np.nan, **mean**)

Applying calculations to an entire column

- Convert "mpg" to "L/100km" in Car dataset

city-mpg
21
21
19
...

→

city-L/100km
11.2
11.2
12.4
...

```
df["city-mpg"]= 235/df["city-mpg"]

df.rename(columns={"city_mpg": "city-L/100km"}, inplace=True)
```

Correcting data types

To *identify* data types:

- Use **`dataframe.dtypes()`** to identify data type

To *convert* data types:

- Use **`dataframe.astype()`** to convert data type

Example: Convert data type to integer in column “price”

```
df["price"] = df["price"].astype("int")
```

Methods of normalizing data

Several approaches for normalization:

① $x_{new} = \frac{x_{old}}{x_{max}}$ **Simple Feature scaling**

② $x_{new} = \frac{x_{old} - x_{min}}{x_{max} - x_{min}}$ **Min-Max**

③ $x_{new} = \frac{x_{old} - \mu}{\sigma}$ **Z-score**

Simple Feature Scaling in Python

With Pandas:

length	width	height
168.8	64.1	48.8
168.8	64.1	48.8
180.0	65.5	52.4
...	...	...

→

length	width	height
0.81	64.1	48.8
0.81	64.1	48.8
0.87	65.5	52.4
...	...	...

```
df["length"] = df["length"]/df["length"].max()
```

Min-max in Python

With Pandas:

length	width	height
168.8	64.1	48.8
168.8	64.1	48.8
180.0	65.5	52.4
...	...	...

→

length	width	height
0.41	64.1	48.8
0.41	64.1	48.8
0.58	65.5	52.4
...	...	...

```
df["length"] = (df["length"]-df["length"].min())/
               (df["length"].max()-df["length"].min())
```

Z-score in Python

With Pandas:

length	width	height
168.8	64.1	48.8
168.8	64.1	48.8
180.0	65.5	52.4
...	...	...

→

length	width	height
-0.034	64.1	48.8
-0.034	64.1	48.8
0.039	65.5	52.4
...	...	...

```
df["length"] = (df["length"]-df["length"].mean())/df["length"].std()
```

Binning in Python Pandas

price
13495
16500
18920
41315
5151
6295
...

→

price	price-binned
13495	Low
16500	Low
18920	Medium
41315	High
5151	Low
6295	Low
...	...

```
bins = np.linspace(min(df["price"]), max(df["price"]), 4)
group_names = ["Low", "Medium", "High"]
df["price-binned"] = pd.cut(df["price"], bins, labels=group_names, include_lowest=True )
```

Dummy variables in Python pandas

- Use pandas.get_dummies() method.
- Convert categorical variables to dummy variables (0 or 1)

fuel
gas
diesel
gas
gas

→

gas	diesel
1	0
0	1
1	0
1	0

```
pd.get_dummies(df['fuel'])
```

groupby(): Example

```
df_test = df[['drive-wheels', 'body-style', 'price']]
df_grp = df_test.groupby(['drive-wheels', 'body-style'], as_index=False).mean()
df_grp
```

	drive-wheels	body-style	price
0	4wd	hatchback	7603.000000
1	4wd	sedan	12647.333333
2	4wd	wagon	9095.750000
3	fwd	convertible	11595.000000
4	fwd	hardtop	8249.000000
5	fwd	hatchback	8396.387755
6	fwd	sedan	9811.800000
7	fwd	wagon	9997.333333
8	rwd	convertible	23949.600000
9	rwd	hardtop	24202.714286
10	rwd	hatchback	14337.777778
11	rwd	sedan	21711.833333
12	rwd	wagon	16994.222222

Pandas method: Pivot()

- One variable is displayed along the columns, and the other variable is displayed along the rows

```
df_pivot = df_grp.pivot(index='drive-wheels', columns='body-style')
```

	price				
body-style	**convertible**	**hardtop**	**hatchback**	**sedan**	**wagon**
drive-wheels					
4wd	20239.229524	20239.229524	7603.000000	12647.333333	9095.750000
fwd	11595.000000	8429.000000	8396.387755	9811.8000000	9997.333333
rwd	23949.600000	24202.714286	14337.777778	21711.833333	16994.222222

Estadística básica / Representación

Descriptive Statistics: Describe()

- Summarize statistics using pandas **describe()** method

```
df.describe()
```

	Unnamed: 0	symboling	normalized-losses	wheel-base	length	width	height	curb-weight	engine-size	bore	stroke
count	201.000000	201.000000	164.000000	201.000000	201.000000	201.000000	201.000000	201.000000	201.000000	201.000000	201.000000
mean	100.000000	0.840796	122.000000	98.797015	174.200995	65.889055	53.766667	2555.666667	126.875622	3.319154	3.256766
std	58.167861	1.254802	35.442168	6.066366	12.322175	2.101471	2.447822	517.296727	41.546834	0.280130	0.316049
min	0.000000	-2.000000	65.000000	86.600000	141.100000	60.300000	47.800000	1488.000000	61.000000	2.540000	2.070000
25%	50.000000	0.000000	NaN	94.500000	166.800000	64.100000	52.000000	2169.000000	98.000000	3.150000	3.110000
50%	100.000000	1.000000	NaN	97.000000	173.200000	65.500000	54.100000	2414.000000	120.000000	3.310000	3.290000
75%	150.000000	2.000000	NaN	102.400000	183.500000	66.600000	55.500000	2926.000000	141.000000	3.580000	3.410000
max	200.000000	3.000000	256.000000	120.900000	208.100000	72.000000	59.800000	4066.000000	326.000000	3.940000	4.170000

Descriptive Statistics: Value_Counts()

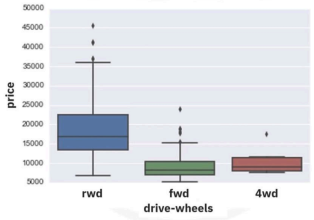
- Summarize the categorical data is by using the **value_counts()** method

```
drive_wheels_counts=df["drive-wheels"].value_counts()

drive_wheels_counts.rename(colunms={'drive-wheels':'value_counts' inplace=True)
drive_wheels_counts.index.name= 'drive-wheels'
```

	value_counts
drive-wheels	
fwd	118
rwd	75
4wd	8

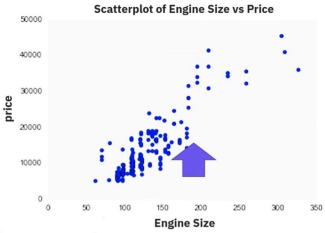
Box Plot: Example

```
sns.boxplot(x= "drive-wheels", y= "price",data=df)
```



Scatterplot: Example

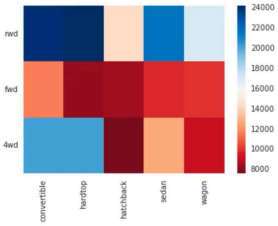
```
y=df["price"]
x=df["engine-size"]
plt.scatter(x,y)

plt.title("Scatterplot of Engine Size vs Price")
plt.xlabel("Engine Size")
plt.ylabel("Price")
```



Heatmap

- Plot target variable against multiple variables

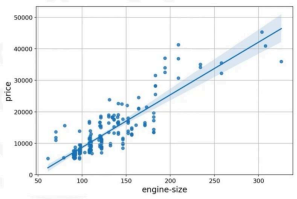
```
plt.pcolor(df_pivot, cmap='RdBu')
plt.colorbar()
plt.show()
```



Correlation: Positive linear relationship

- Correlation between two features (engine-size and price)

```
sns.regplot(x="engine-size", y= "price",
data=df)
plt.ylim(0,)
```



Pearson Correlation

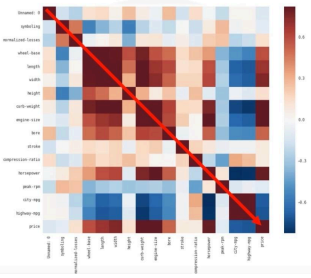
- Measure the strength of the correlation between two features
  - Correlation coefficient
  - P-value
- Correlation coefficient
  - Close to +1: Large Positive relationship
  - Close to -1: Large Negative relationship
  - Close to 0: No relationship
- Strong correlation
  - Correlation coefficient close to 1 or -1
  - P value less than 0.001
- P-value
  - P-value < 0.001 Strong certainty in the result
  - P-value < 0.05 Moderate certainty in the result
  - P-value < 0.1 Weak certainty in the result
  - P-value > 0. 1 No certainty in the result

Pearson Correlation- Example

```
pearson_coef, p_value = stats.pearsonr(df['horsepower'], df[ 'price'])
```

- Pearson correlation: 0.81
- P-value : 9.35 e-48

Correlation- Heatmap



Modelos Lineales y Polinómicos

Fitting a simple linear model estimator

- X :Predictor variable
- Y:Target variable

1. Import linear_model from scikit-learn

```
from sklearn.linear_model import LinearRegression
```

1. Create a Linear Regression Object using the constructor:

```
lm=LinearRegression()
```

Fitting a simple linear model

- We define the predictor variable and target variable

```
X = df[['highway-mpg']]
Y = df['price']
```

- Then use lm.fit (X, Y) to fit the model , i.e find the parameters b_0 and b_1

```
lm.fit(X, Y)
```

- We can obtain a prediction

```
Yhat=lm.predict(X)
```

Yhat	X
2	5
:	
3	4

SLR - Estimated linear model

- We can view the intercept (b_0): `lm.intercept_`
  `38423.305858`
- We can also view the slope (b_1): `lm.coef_`
  -821.73337832

Fitting a multiple linear model estimator

1. We can extract the for 4 predictor variables and store them in the variable Z

```
Z = df[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']]
```

2. Then train the model as before:

```
lm.fit(Z, df['price'])
```

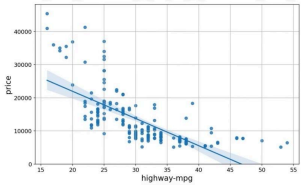
3. We can also obtain a prediction

```
Yhat=lm.predict(X)
```

Regression plot

```
import seaborn as sns
```

```
sns.regplot(x="highway-mpg", y="price", data=df)
plt.ylim(0,)
```



Polynomial Regression

1. Calculate Polynomial of 3rd order

```
f=np.polyfit(x,y,3)
```

```
p=np.poly1d(f)
```

2. We can print out the model

```
print (p)
```

$$-1.557(x_1)^3 + 204.8(x_1)^2 + 8965\,x_1 + 1.37\text{x}10^5$$

Polynomial Regression with more than one dimension

```
pr=PolynomialFeatures(degree=2)
```

X_1	X_2
1	2

```
pr.fit_transform([1,2], include_bias=False)
```

X_1	X_2	X_1X_2	${X_1}^2$	${X_2}^2$
1	2	(1) 2	1	$(2)^2$

1	2	2	1	4

Pre-procesamiento

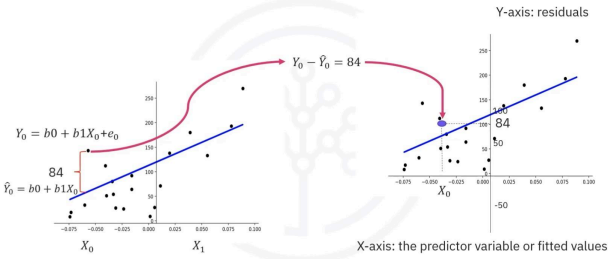
Pre-processing

- For example we can Normalize the each feature simultaneously:

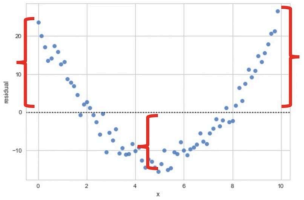
```
from sklearn.preprocessing import StandardScaler

SCALE=StandardScaler()
SCALE.fit(x_data[['horsepower', 'highway-mpg']])

x_scale=SCALE.transform(x_data[['horsepower', 'highway-mpg']])
```

Evaluación

Residual plot



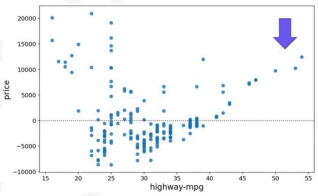
Residual plot



- Not randomly spread out around the x-axis
- Nonlinear model may be more appropriate

Residual plot

```
import seaborn as sns

sns.residplot(df['highway-mpg'],
df['price'])
```



MLR: Distribution plots

Distribution plots

```
import seaborn as sns
```

```
ax1 = sns.distplot(df['price'], hist=False, color="r", label="Actual Value")

sns.distplot(Yhat, hist=False, color="b", label="Fitted Values" , ax=ax1)
```

Mean squared error (MSE)

- In Python, we can measure the MSE as follows

```
from sklearn.metrics import mean_squared_error

mean_squared_error(actual_value,predicted_value)
```

R-squared/ R^2

- The Coefficient of Determination or R squared (R^2)
  - Is a measure to determine how close the data is to the fitted regression line
- The base model is the average of the y-values
- R^2 compares the regression line to this average

Coefficient of Determination (R^2)

$$R^2 = \left(1 - \frac{MSE(y, \hat{y})}{MSE(y, \bar{y})}\right)$$

R-squared/ R^2

We can calculate the R^2 as follows

```
X = df[['highway-mpg']]
Y = df['price']

lm.fit(X, Y)

lm.score(X,y)
```

Output:
0.496591188 → 49.7%

Do the predicted values make sense?

- First we train the model
  `lm.fit(df['highway-mpg'],df['prices'])`
- Let's predict the price of a car with 30 highway-mpg
  `lm.predict(np.array(30.0).reshape(-1,1))`
- Result: $ 13771.30

```
lm.coef_
-821.73337832
```

  - **Price** = 38423.31 - 821.73 * **highway-mpg**

Do the predicted values make sense?

- First, we import numpy

```
import numpy as np
```

- We use the numpy function arrange to generate a sequence from 1 to 100

```
new_input=np.arange(1,101,1).reshape(-1,1)
```

1	2	...	99	100

Do the predicted values make sense?

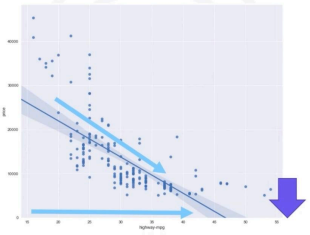
- We can predict new values

```
yhat=lm.predict(new_input)
```

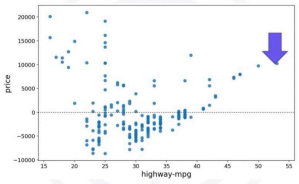
```
array([ 37601.57247984,  36779.83910151,  35958.10572319,  35136.37234487,
        34314.63896655,  33492.90558823,  32671.1722099 ,  31849.43883158,
        31027.70545326,  30205.97207494,  29384.23869662,  28562.50531829,
        27740.77193997,  26919.03856165,  26097.30518333,  25275.57180501,
        24453.83842668,  23632.10504836,  22810.37167004,  21988.63829172,
        21166.9049134 ,  20345.17153508,  19523.43815675,  18701.70477843,
        17879.97140011,  17058.23802179,  16236.50464347,  15414.77126514,
        14593.03788682,  13771.3045085 ,  12949.57113018,  12127.83775186,
        11306.10437353,  10484.37099521,   9662.63761689,   8840.90423857,
         8019.17086025,   7197.43748192,   6375.70410358,   5553.97072528,
         4732.23734696,   3910.50396864,   3088.77059031,   2267.03721199,
         1445.30383367,    623.57045535,   -198.16291298,   -1019.89629131,
        -1841.62967962,  -2663.36305794,  -3485.09643626,  -4306.82981458,
        -5128.5631929 ,  -5950.29657123,  -6772.02994955,  -7593.76332787,
        -8415.49670619,  -9237.23008451, -10058.96346284, -10880.69684116,
       -11702.43021948, -12524.1635978 , -13345.89697612, -14167.63035445,
       -14989.36373277, -15811.09711109, -16632.83048941, -17454.56386773,
       -18276.29724606, -19098.03062438, -19919.7640027 , -20741.49738102,
       -21563.23075934, -22384.96413767, -23206.69751599, -24028.43089431,
       -24850.16427263, -25671.89765095, -26493.63102927, -27315.3644076 ,
       -28137.09778592, -28958.83116424, -29780.56454256, -30602.29792088,
       -31424.03129921, -32245.76467753, -33067.49805585, -33889.23143417,
       -34710.96481249, -35532.69819082, -36354.43156914, -37176.16494746,
       -37997.89832578, -38819.6317041 , -39641.36508243, -40463.09846075,
       -41284.83183907, -42106.56521739, -42928.29859571])
```

Visualization

- Simply visualizing your data with a regression

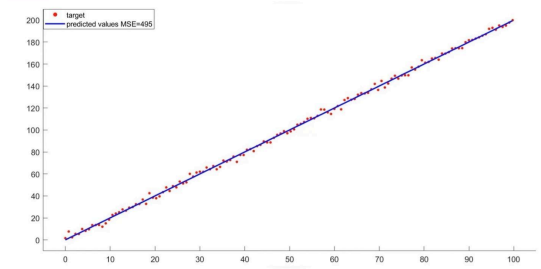


Residual plot

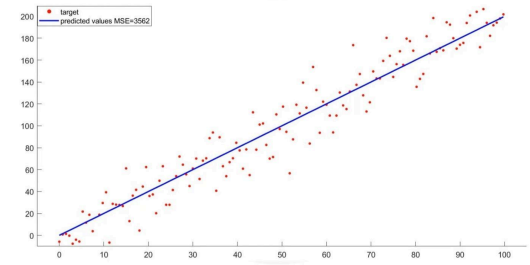


Los residuales han de estar distribuidos uniformemente, sino un modelo polinomial puede resultar más adecuado

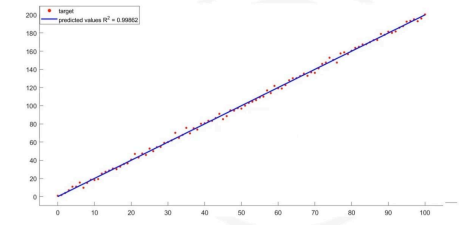
Numerical measures for Evaluation



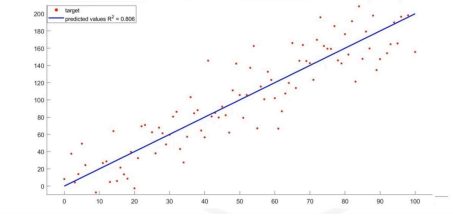
Numerical measures for Evaluation



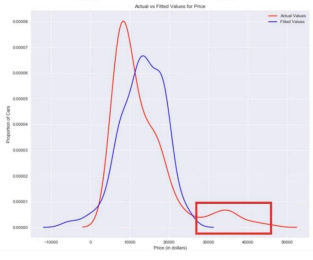
Numerical measures for Evaluation



Numerical measures for Evaluation



Visualization



Comparing MLR and SLR

1. Is a lower MSE always implying a better fit?
   - Not necessarily
2. MSE for an MLR model will be smaller than the MSE for an SLR model since the errors of the data will decrease when more variables are included in the model
3. Polynomial regression will also have a smaller MSE than regular regression
4. A similar inverse relationship holds for R^2
5. In the next section, we will look at better ways to evaluate the model

Pipelines

Pipelines

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
```

Pipeline constructor

```
Input=[('polynomial',PolynomialFeature(degree=2)),('scale',StandardScaler()),...
('Model',LinearRegression())]
```

- Pipeline constructor

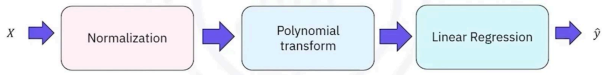
```
pipe=Pipeline(Input)
```

pipeline object

Pipeline constructor

- We can train the pipeline object

```
Pipe.fit(df[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']],y)

yhat=Pipe.predict(X[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])
```

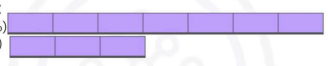


Test/Train split (fraccionar datos)

Training/Testing sets

Data:

- Split dataset into:
  - training set (70%)
  - testing set (30%)



- Build and train the model with a training set

Function training_test_split()

- Split data into random train and test subsets

```
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=0.3, random_state=0)
```

- **x_data**: features or independent variables
- **y_data**: dataset target: df['price']
- **x_train, y_train**: parts of available data as training set
- **x_test, y_test**: parts of available data as testing set
- **test_size**: percentage of the data for testing (here 30%)
- **random_state**: number generator used for random sampling

Generalization/Validation

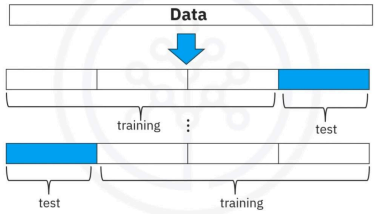
Generalization performance

- Generalization error is a measure of how well our data does at predicting previously unseen data
- The error we obtain using our testing data is an approximation of this error

Generalization error

Cross validation

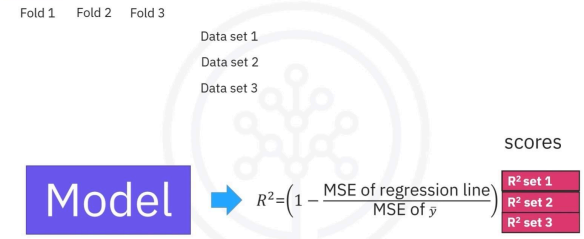
- Most common out-of-sample evaluation metrics
- More effective use of data (each observation is used for both training and testing)



Function cross_val_score()

```
from sklearn.model_selection import cross_val_score

scores= cross_val_score(lr, x_data, y_data, cv=3)
```

Evaluating cross-validation score

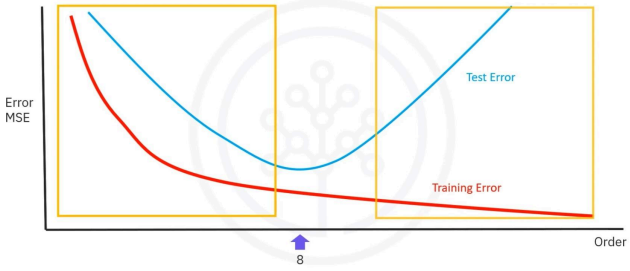


Function cross_val_predict()

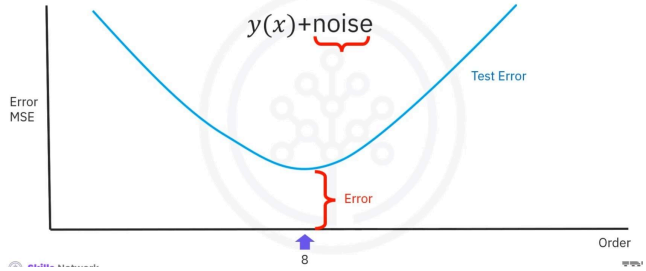
- It returns the prediction that was obtained for each element when it was in the test set.
- Has a similar interface to cross_val_score()

```
from sklearn.model_selection import cross_val_predict

yhat=cross_val_predict (lr2e, x_data, y_data, cv=3)
```

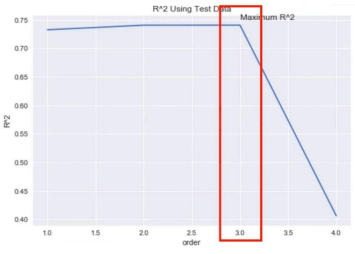
Model Selection



Model Selection



Model Selection



Ejemplo validación en diferentes grados de una regresión polinómica

```
Rsqu_test=[]
order=[1,2,3,4]
for n in order:
    pr=PolynomialFeatures(degree=n)
    x_train_pr=pr.fit_transform(x_train[['horsepower']])
    x_test_pr=pr.fit_transform(x_test[['horsepower']])
    lr.fit(x_train_pr,y_train)
    Rsqu_test.append(lr.score(x_test_pr,y_test))
```

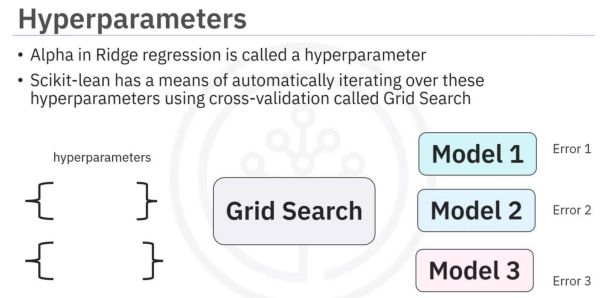
Ridge Regression

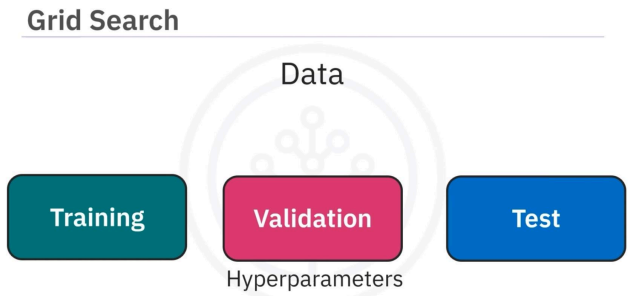
Ridge Regression

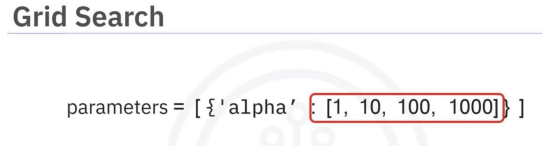
```
from sklearn.linear_model import Ridge
RidgeModel=Ridge(alpha=0.1)
RidgeModel.fit(X,y)
Yhat=RidgeModel.predict(X)
```

Ridge Regression

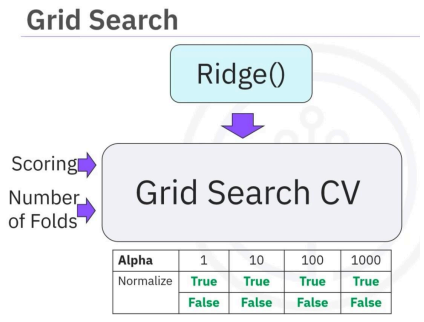
Grid Search (búsqueda para optimización de parámetros de modelos)








```
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV

parameters1= [{'alpha': [0.001,0.1,1, 10, 100, 1000,10000,100000,100000]}]

RR=Ridge()

Grid1 = GridSearchCV(RR, parameters1,cv=4)

Grid1.fit(x_data[['horsepower', 'curb-weight', 'engine-size',
'highway-mpg']],y_data)

Grid1.best_estimator_

scores  = Grid1.cv_results_
scores['mean_test_score']
```



Grid Search

```
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV

parameters1= [{'alpha': [0.001,0.1,1, 10, 100, 1000,10000,100000,100000]}

RR=Ridge()

Grid1 = GridSearchCV(RR, parameters1,cv=4)

Grid1.fit(x_data[['horsepower', 'curb-weight', 'engine-size', 'highway-
mpg']],y_data)

Grid1.best_estimator_

scores  = Grid1.cv_results_
```