EPL448: Data Mining on the Web – Labs 8

University of Cyprus
Department of
Computer Science

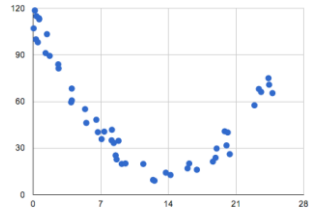
Παύλος Αντωνίου

Γραφείο: Β109, ΘΕΕ01

Predictive modeling techniques

- Predictive modeling techniques help translate raw data into value
  - Machine learning predictive techniques such as Support Vector Machines (SVMs), Decision trees, boosting methods, learn from data and build models
- Data + Predictive Modeling Technique → Predictive Model
  - 3 phases to prepare a predictive model: Training – Validation – Testing
    - Split initial dataset into 3 smaller datasets (e.g. 80%-10%-10%); one for each phase

1. Learning/training phase:
    - Training data are used to train a predictive modelling technique and create a model
      - model represents what was learned by a machine learning algorithm
    - Example:
      - Predictive modelling technique to train: Polynomial equation: $y = \beta_0 + \beta_1 X + \beta_2 X^2$
        - » Equation parameters: β_0, β_1, β_2 will be estimated during training
        - » Equation hyperparameter: degree of the polynomial function (configured prior training)
      - Dataset to train the algorithm and find the "best curve" that passes between points →
      - The outcome of training phase is the model e.g.: $y = 0.45 + 0.7X + 1.2X^2$



X	y
0.10	1.51
0.15	0.92
0.17	1.96
0.22	0.53
0.27	0.38

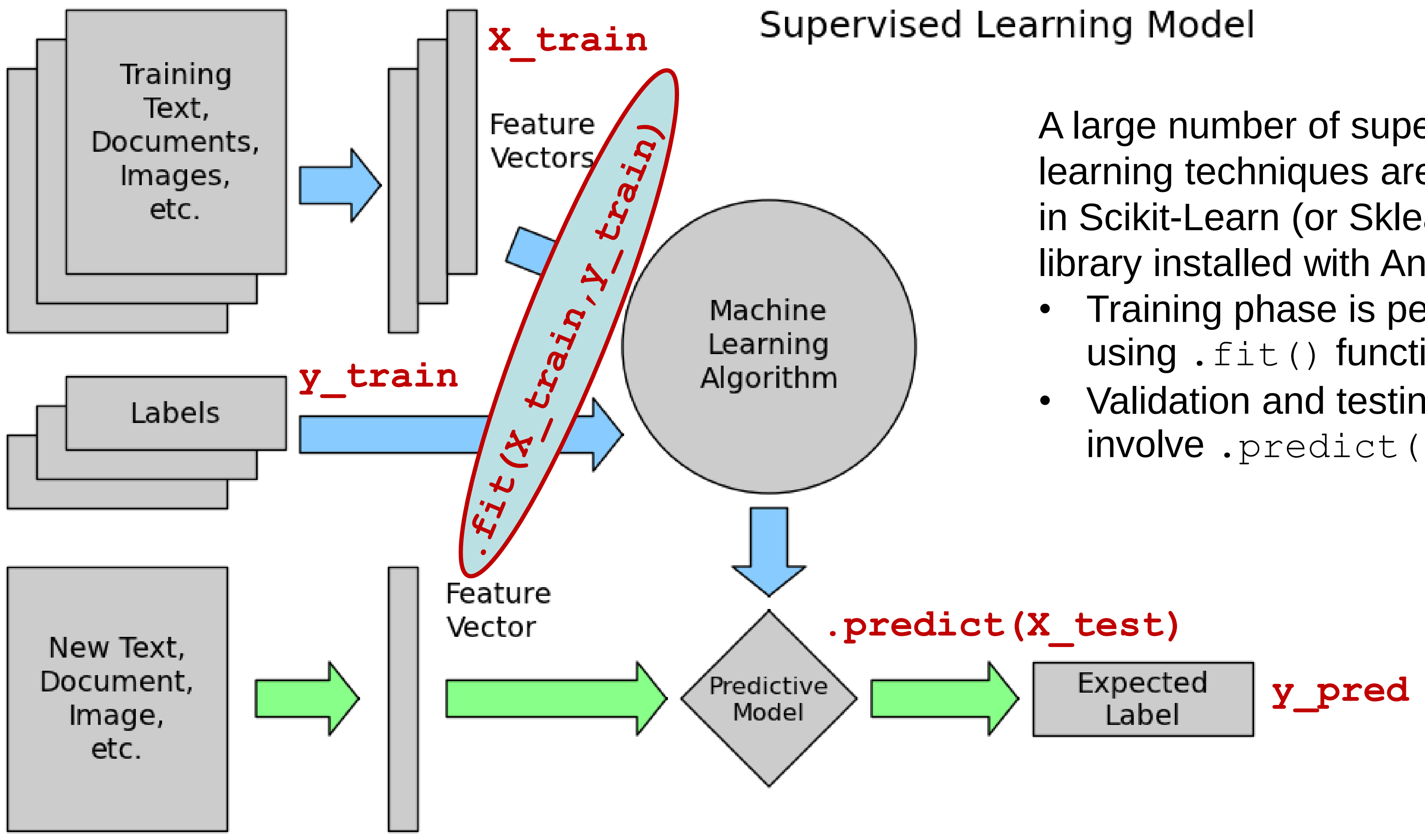
Predictive modeling techniques

2. Validation phase
   - Validation data (not seen during training phase) used to make predictions and measure the performance (e.g. mean squared error) of the model and to tune hyperparameters
   - Example:
     - After measuring the performance of the quadratic (2nd degree) model, change the degree of the polynomial equation e.g. to 3, re-run on training (phase) data to create a new cubic (3rd degree) model and measure the performance of the new model on validation data – repeat by changing the degree until the best model (with best performance, e.g. lower error) is achieved
3. Testing phase
   - Estimate the performance of the final model (with "best" parameters and hyperparameters) using test data (not seen during training and validation phases)
   - This is the final performance of the model
4. Prediction (or application) phase:
   - Apply the final model e.g.: $y = 0.65 + 0.13X + 1.9X^2 + 0.77X^3$ to real-world input data (a new value of X not in the initial dataset) and predict output y

Supervised learning

- You have input variables (X) and an output variable (y) available and use a predictive modelling technique to build a model that captures the relationship between input and output data
  - Majority of predictive techniques are supervised learning techniques
- Supervised learning problems can be further grouped into:
  - **Classification problems**: A classification problem is when the output variable (y) is a category, such as "disease" or "no disease" (binary classification) and "red" or "blue" or "green" (multiclass classification)
    - Popular techniques: Logistic Regression (binary classification), Linear Discriminant Analysis (LDA), K-Nearest Neighbors (KNN), Decision Trees (Random Forest), Support Vector Machine (SVM), Naïve Bayes, Gaussian Naïve Bayes, XGBoost, AdaBoost
  - **Regression problems**: A regression problem is when the output variable (y) is a numerical value, such as "price" or "weight"
    - Popular techniques: Linear Regression, Polynomial Regression, Support Vector Regression (SVR), Random Forest Regression, XGBoost Regression, AdaBoost Regression

Supervised learning



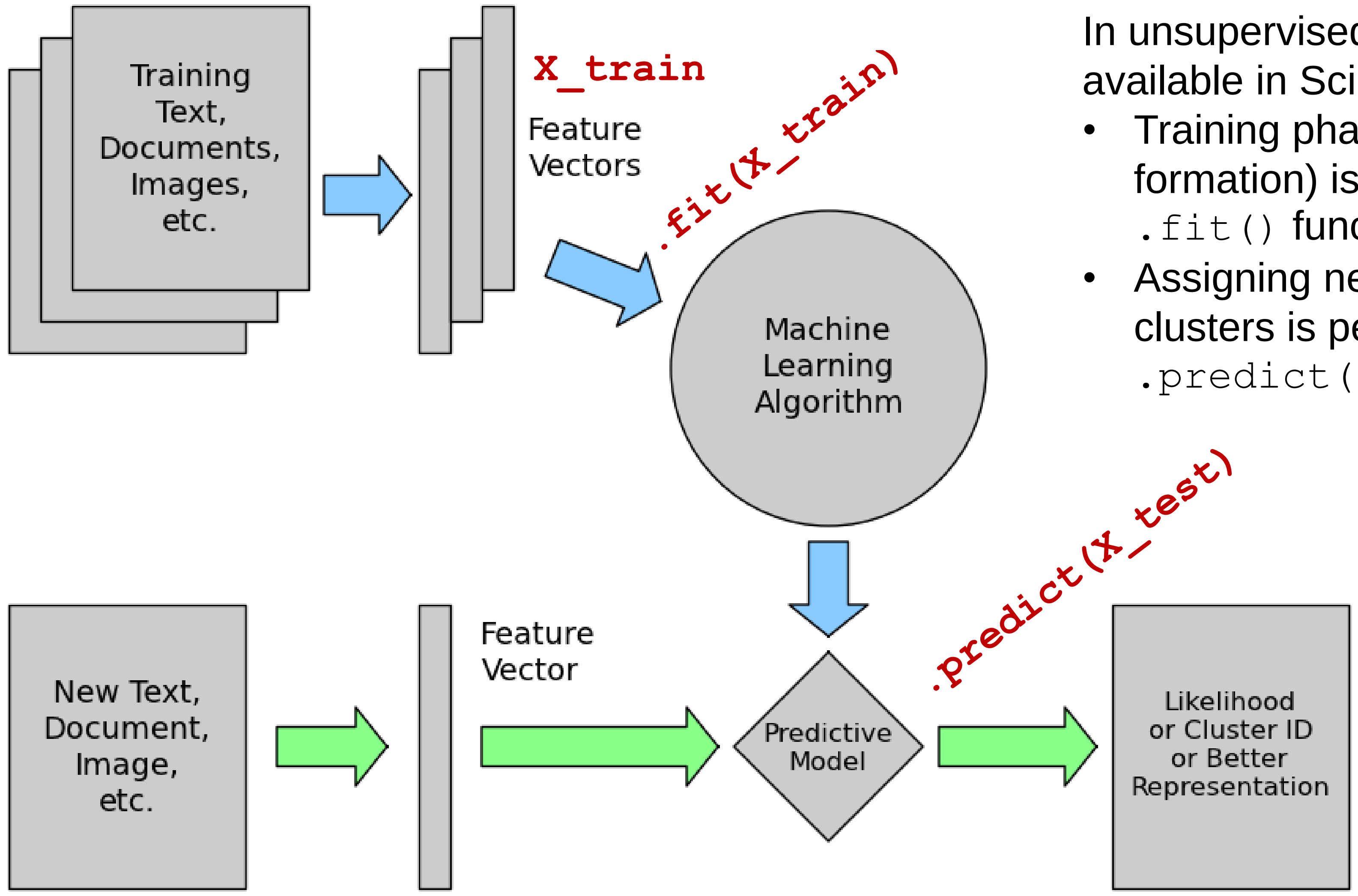
A large number of supervised learning techniques are available in Scikit-Learn (or Sklearn) library installed with Anaconda

- Training phase is performed using `.fit()` function
- Validation and testing phase involve `.predict()` function

Unsupervised learning

- You only have input vars (X) and no corresponding output variable (y)
  - no mapping from input to output data
- Goal: model the underlying structure or distribution in the data in order to learn more about the data, extract insights
- Unsupervised learning problems can be further grouped into:
  - **Clustering problems**: A clustering problem is where you want to discover the inherent groupings in the data, such as grouping customers by purchasing behavior.
    - Popular techniques: k-means
  - **Association problems**: An association rule learning problem is where you want to discover rules that describe large portions of your data, such as people that buy X1 also tend to buy X2
    - Popular techniques: Apriori algorithm

Unsupervised learning


Training Text, Documents, Images, etc.
X_train
Feature Vectors
.fit(X_train)
Machine Learning Algorithm
.predict(X_test)
New Text, Document, Image, etc.
Feature Vector
Predictive Model
Likelihood or Cluster ID or Better Representation

In unsupervised learning techniques available in Scikit-Learn (or Sklearn)

- Training phase (e.g. cluster formation) is performed using `.fit()` function
- Assigning new data into existing clusters is performed by `.predict()` function

Regression

- The process of estimating the relationships between a dependent variable (or target variable) y which takes numerical values and one or more independent (or input) variables (called features) x
  - Example: Estimate the relationship between the house price (dependent var) and the house area in square meters (independent var)
  - House area is independent variable because we cannot mathematically determine it. But, we can determine / predict house price value based on the house area.
- Some regression algorithms:
  - Linear Regression (simple, multiple) – first degree equation
  - Polynomial Regression – higher degree (2nd, 3rd, …) equations
  - Support Vector Regression
  - Ensemble Regression (e.g. Random Forest Regressor, Ada Boost Regressor)

Linear Regression (LR)

- Linear regression assumes that the relationships between the dependent (target) variable and the independent variables are linear
- Therefore, the dependent variable y can be calculated from a linear combination of the independent variables (X):

$$y = \beta_0 + \sum_{j=1}^{p} \beta_j * X_j = \beta_0 + \beta_1 * \mathrm{X}_1 + \beta_2 * \mathrm{X}_2 + \cdots$$

- Vector β involves initially unknown coefficients (parameters), which will be evaluated using a dataset with values for target variable and features

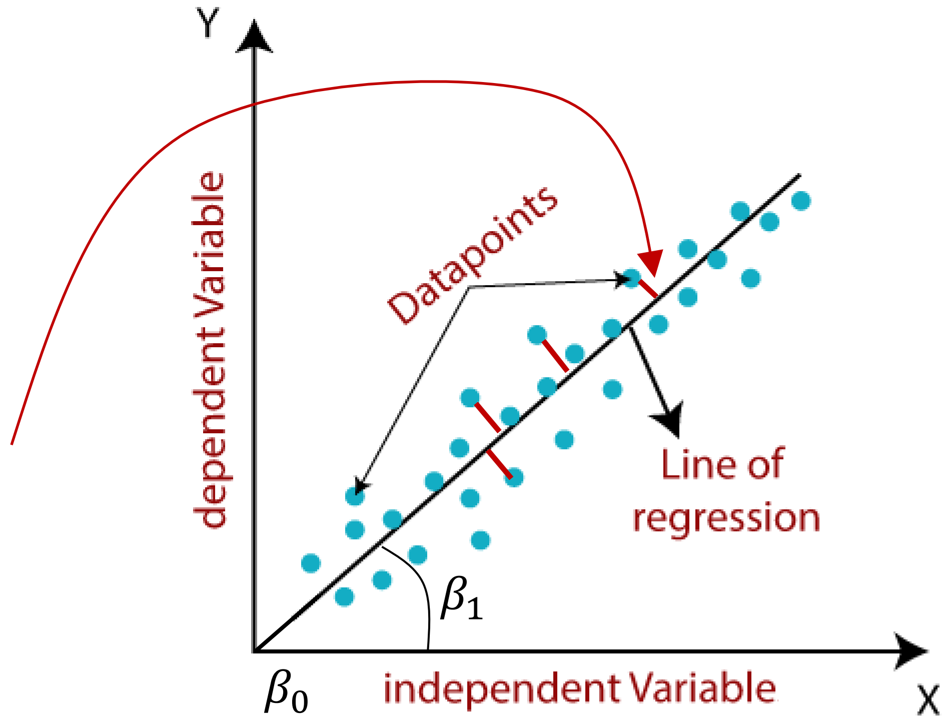
Simple Linear Regression

- **Simple Linear regression**: one independent variable X:

$$y = \beta_0 + \beta_1 X + \epsilon$$

  - **Goal: Fit the best intercept line (evaluate β$_0$ and β$_1$) that passes between all data points that minimizes the error**
  - y : Dependent variable (target variable)
  - X : Independent variable (feature)
  - β$_0$: Intercept (the target value when X = 0)
  - β$_1$: Slope. Explains the change in Y when X changes by 1 unit = Δy/ΔX
  - ∈ : Error. This represents the residual value, i.e. the difference between the observed and the fitted (predicted) value

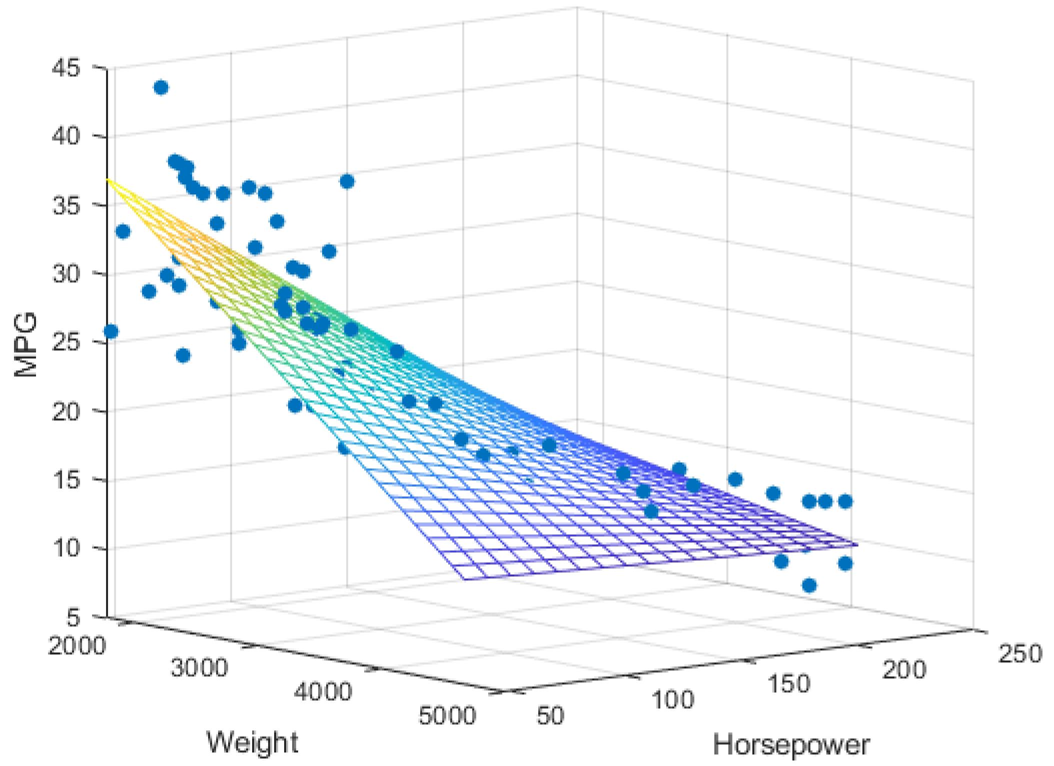
```
X       y
0.10    1.51
0.15    0.92
0.17    1.96
0.22    0.53
0.27    0.38
```



Multiple Linear Regression

- **Multiple Linear regression**: more than one independent variables Xi in the linear function:

$$y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3\, X_3 + \cdots \beta_n X_n + \epsilon$$



In this image n=2
Two independent variables:
- Weight
- Horsepower

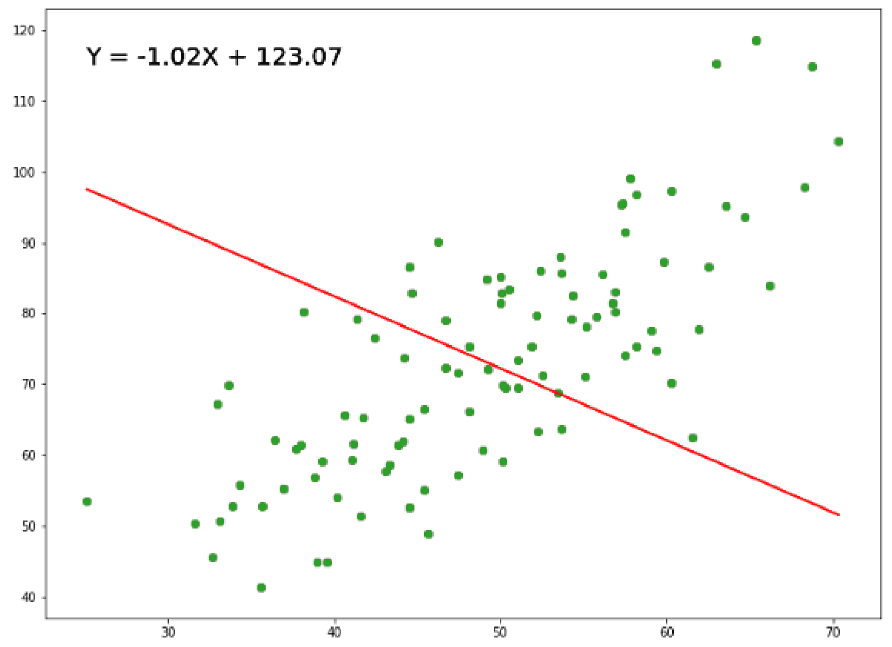
Dependent variable:
- MPG (miles per gallon)

Regression finds the best-fitting plane that passes through all points minimizing the error

Linear Regression Methods

- **Ordinary least squares** (OLS) is a non-iterative method that fits a model (line or plane) such that the sum of squared error is minimized.

- **Gradient descent** finds the linear model coefficients iteratively



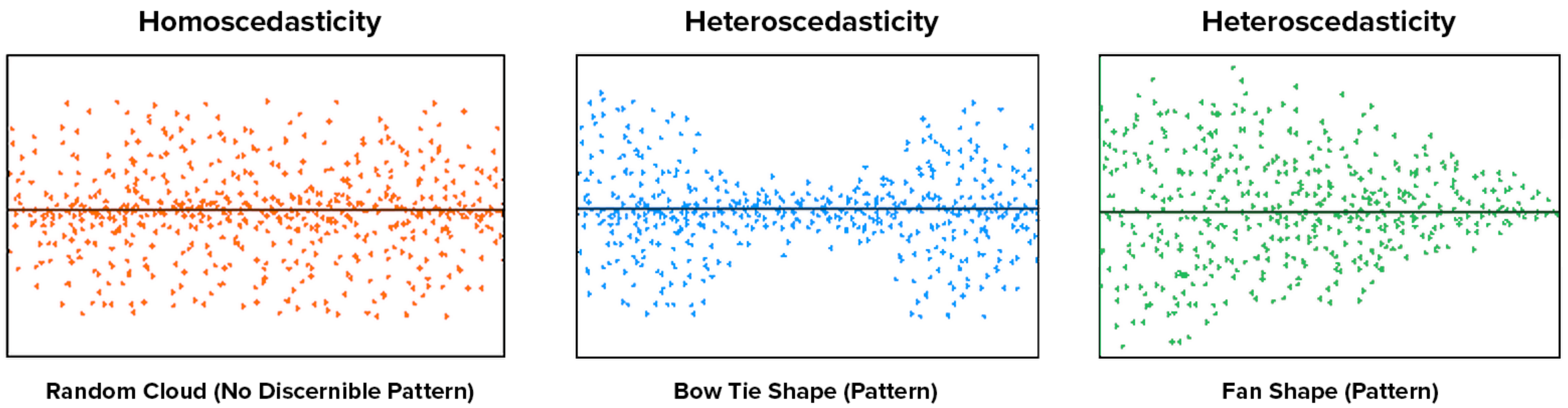
- When the β coefficients are estimated, the equation can be used to predict the target value y given an input X vector

Assumptions for using Linear Regression

- Linear relationships
  - relationship between each independent variable and the dependent variable needs to be linear – can best be tested with scatter plots / pair plots
- No or little multicollinearity
  - Multicollinearity: two or more independent variables are highly correlated to one another – can be checked with correlation matrix (visualized by heat map)
    - If multicollinearity is discovered, the analyst may drop one of the two variables that are highly correlated, or simply leave them in and note that multicollinearity is present.
    - There are some techniques to remove multicollinearity such as centering each correlated variable (remove mean value from all observed values of each variable) -- StandardScaler
- Normality of residuals
  - LR requires the residuals (error terms) of the model to be normally distributed, with mean equal to 0 – can best be checked with a histogram of the residuals; normality test functions are also available

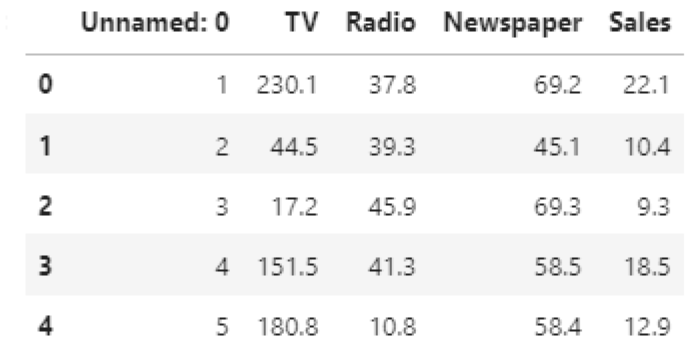
Linear Regression Assumptions

- No auto-correlation of residuals
  - Autocorrelation in residuals occurs when the values of residuals are dependent from each other – use the Ljungbox test on residuals
    - i.e. the current value of a residual is dependent of the previous (historic) residual values
      - this is more evident in time series data as there is a pattern of time (e.g. in stock markets, people tend to buy stocks more towards the beginning of weekends and tend to sell more on Mondays)
- Homoscedasticity of residuals
  - Homoscedasticity means that the residuals (error) are constant along the values of the dependent variable



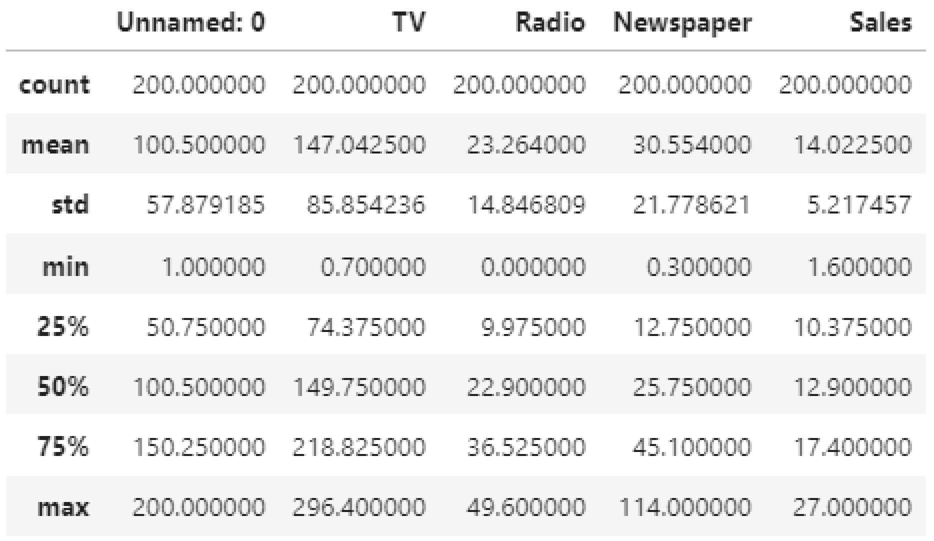
Linear Regression: Get to know data

```
import pandas as pd
import numpy as np
df = pd.read_csv('Advertising.csv')
df.head()
```



	Unnamed: 0	TV	Radio	Newspaper	Sales
0	1	230.1	37.8	69.2	22.1
1	2	44.5	39.3	45.1	10.4
2	3	17.2	45.9	69.3	9.3
3	4	151.5	41.3	58.5	18.5
4	5	180.8	10.8	58.4	12.9

```
df.describe()
```



	Unnamed: 0	TV	Radio	Newspaper	Sales
count	200.000000	200.000000	200.000000	200.000000	200.000000
mean	100.500000	147.042500	23.264000	30.554000	14.022500
std	57.879185	85.854236	14.846809	21.778621	5.217457
min	1.000000	0.700000	0.000000	0.300000	1.600000
25%	50.750000	74.375000	9.975000	12.750000	10.375000
50%	100.500000	149.750000	22.900000	25.750000	12.900000
75%	150.250000	218.825000	36.525000	45.100000	17.400000
max	200.000000	296.400000	49.600000	114.000000	27.000000

Independent variables (features)

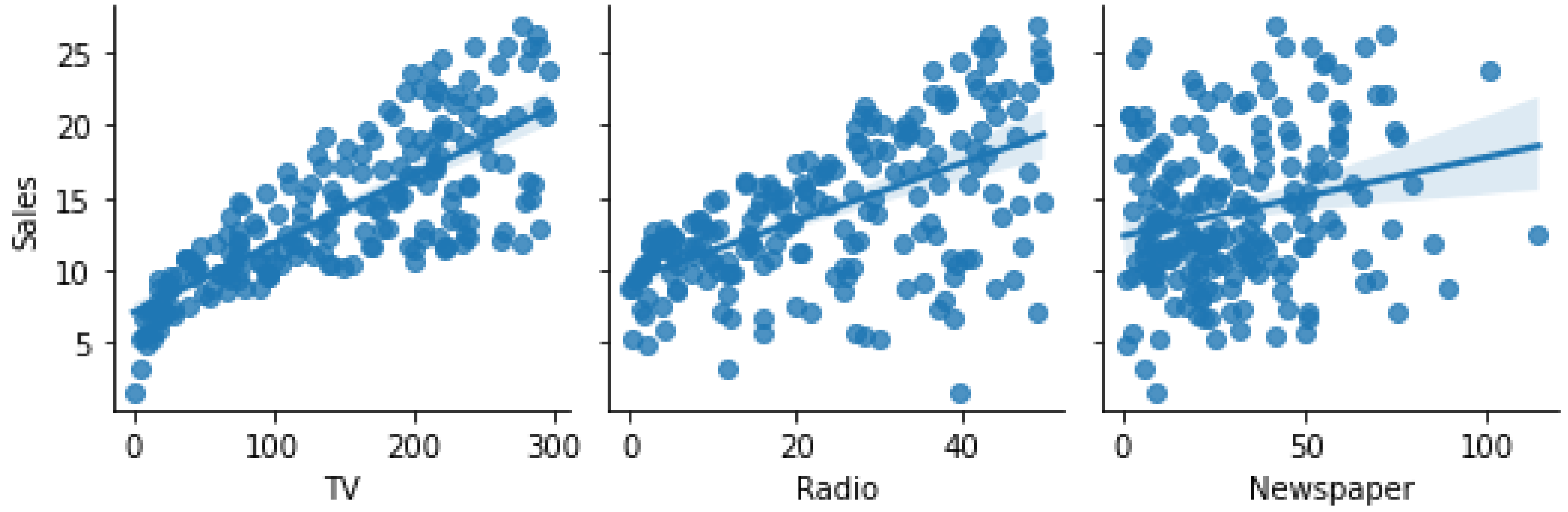
target variable

Dataset description: Sales (in thousands of units) for a particular product based on the advertising budgets (in thousands of dollars) for TV, radio, and newspaper media.

Linear Regression: Testing assumptions

- Linearity

```
sns.pairplot(df,x_vars=["TV","Radio","Newspaper"],y_vars= "Sales",kind="reg")
```

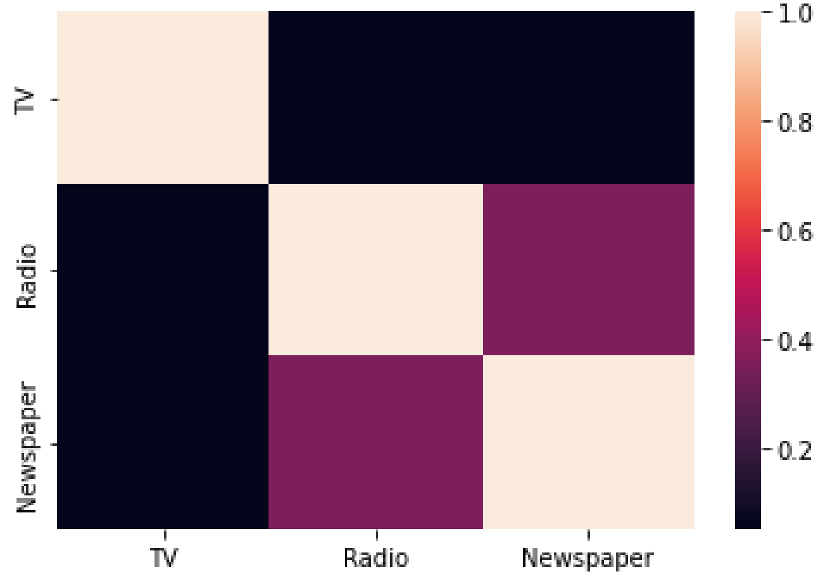


By looking at the plots we can see that none of the independent variables has an accurately linear relationship with Sales but TV and Radio do still better than Newspaper which seems to hardly have any specific shape. So, it shows that a linear regression fitting might not be the best model for it. A linear model might not be able to *efficiently* explain the data in terms of variability, prediction accuracy etc.

Linear Regression: Testing assumptions

- Multicollinearity
  - Independent variables seem to be uncorrelated (there is no correlation between independent variables > 0.75)

```
df_features = df[["TV", "Radio", "Newspaper"]]
sns.heatmap(data=df_features.corr())
plt.show()
```



Linear Regression: Prepare variable vectors

- Rest of the assumptions require us to perform the regression and calculate the residuals (error terms)

```
# get the values of the dataframe that will be used in the regression model
dataset = df.values

# extract the features (independent variables)
X = dataset[:,1:4]
print(X[0:10])

# extract the dependent (target) variable
y = dataset[:,4]
print(y[0:10])
```

```
[[230.1  37.8  69.2]
 [ 44.5  39.3  45.1]
 [ 17.2  45.9  69.3]
 [151.5  41.3  58.5]
 [180.8  10.8  58.4]
 [  8.7  48.9  75. ]
 [ 57.5  32.8  23.5]
 [120.2  19.6  11.6]
 [  8.6   2.1   1. ]
 [199.8   2.6  21.2]]
```

```
[22.1 10.4  9.3 18.5 12.9  7.2 11.8 13.2  4.8 10.6]
```

Non standardized independent values

Linear Regression: Testing assumptions

LinearRegression class uses Ordinary Least Squares (OLS) solver from scipy

```
from sklearn.linear_model import LinearRegression
lregr = LinearRegression()

from sklearn.model_selection import train_test_split
X_train, X_2, y_train, y_2 = train_test_split(X, y, train_size=0.80)
```

Training data size: 80%
Remaining data (X_2, y_2) size: 20%

X_train

```
[[230.1   37.8   69.2]
 [ 44.5   39.3   45.1]
 [ 17.2   45.9   69.3]
 [151.5   41.3   58.5]
 [180.8   10.8   58.4]
 [  8.7   48.9   75. ]
```

X_2

```
 [ 57.5   32.8   23.5]
 [120.2   19.6   11.6]
 [  8.6    2.1    1. ]
 [199.8    2.6   21.2]]
```

y_train

```
[22.1
 10.4
  9.3
 18.5
 12.9
  7.2
```

y_2

```
 11.8
 13.2
  4.8
 10.6]
```

Non standardized independent values

Linear Regression: Testing assumptions

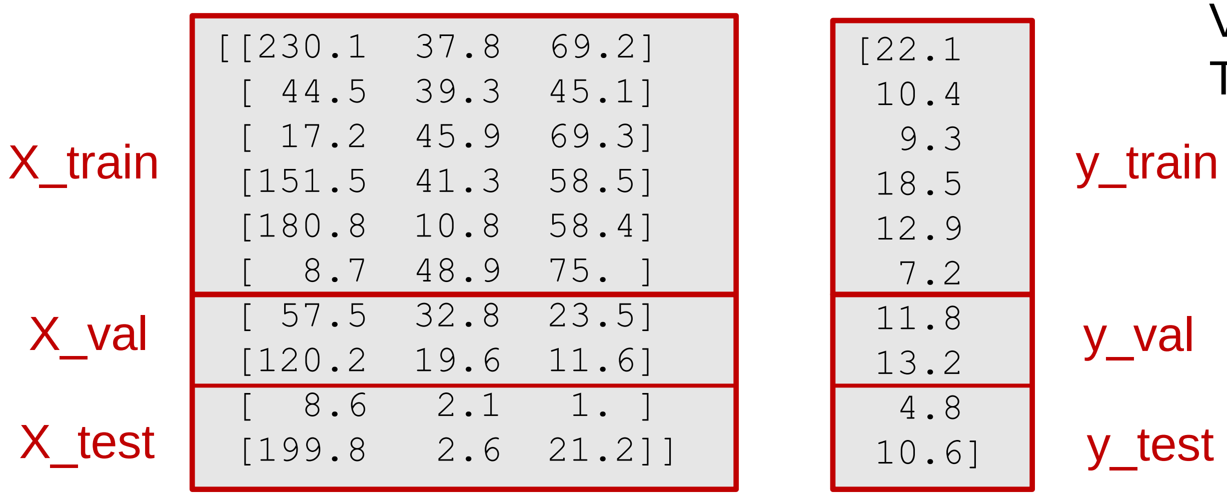
```
from sklearn.linear_model import LinearRegression
lregr = LinearRegression()
```

LinearRegression class uses Ordinary Least Squares (OLS) solver from scipy

```
from sklearn.model_selection import train_test_split
X_train, X_2, y_train, y_2 = train_test_split(X, y, train_size=0.80)
X_val, X_test, y_val, y_test = train_test_split(X_2, y_2, train_size=0.50)
```

Validation data size: 50% of remaining
Testing data size: 50% of remaining

Training data size: 80%
Validation data size: 10%
Testing data size: 10%



```
[[230.1  37.8  69.2]
 [ 44.5  39.3  45.1]
 [ 17.2  45.9  69.3]
 [151.5  41.3  58.5]
 [180.8  10.8  58.4]
 [  8.7  48.9  75. ]
 [ 57.5  32.8  23.5]
 [120.2  19.6  11.6]
 [  8.6   2.1   1. ]
 [199.8   2.6  21.2]]
```

```
[22.1
 10.4
  9.3
 18.5
 12.9
  7.2
 11.8
 13.2
  4.8
 10.6]
```

Non standardized independent values

Linear Regression: Testing assumptions

LinearRegression class uses Ordinary Least Squares (OLS) solver from scipy

```
from sklearn.linear_model import LinearRegression
lregr = LinearRegression()

from sklearn.model_selection import train_test_split
X_train, X_2, y_train, y_2 = train_test_split(X, y, train_size=0.80)
X_val, X_test, y_val, y_test = train_test_split(X_2, y_2, train_size=0.50)

# train model (Fit linear model) and evaluate model β coefficients
model = legr.fit(X_train, y_train)
# print model intercept (β0)
print("β0 =", model.intercept_)
# print model coefficients
print("[β1,β2,β3] =", model.coef_)
```

```
β0 = 2.99489303049533
[β1,β2,β3] = [ 0.04458402  0.19649703 -0.00278146]
Residuals: [ 0.11256448  2.16206142 -9.18318566
0.21444367  0.62679197 -1.90974587
 -2.03802209  0.9477193   0.30597666  0.03544328]
```

We apply knowledge (X_train, y_train) to explain the phenomenon and create a "model" of reality

Model is a function (with that you may accept or reject as) being representative of describing your phenomenon: $y = 2.99 + 0.044*x_1 + 0.196*x_2 - 0.0027*x_3$

Non standardized independent values

Linear Regression: Testing assumptions

LinearRegression class uses Ordinary Least Squares (OLS) solver from scipy

```
from sklearn.linear_model import LinearRegression
lregr = LinearRegression()

from sklearn.model_selection import train_test_split
X_train, X_2, y_train, y_2 = train_test_split(X, y, train_size=0.80)
X_val, X_test, y_val, y_test = train_test_split(X_2, y_2, train_size=0.50)

# train model (Fit linear model) and evaluate model β coefficients
model = legr.fit(X_train, y_train)
# print model intercept (β0)
print("β0 =", model.intercept_)
# print model coefficients
print("[β1,β2,β3] =", model.coef_)

# estimate residuals
# predict
y_pred = model.predict(X_val)
# residuals is the differences between real y values (y_val) and predicted y values
residuals = y_val - y_pred
print("Residuals:", residuals[:10])
```

```
β0 = 2.89257005115115
[β1,β2,β3] = [0.04416235 0.19900368 0.00116268]
Residuals: [ 1.2505431   0.96947665  1.72847857
1.23621333 -0.30215643  2.15665355
 -5.89526331 -1.75879164 -1.80528358  0.32765447]
```

"TV", "Radio", "Newspaper"

We cannot say that Radio has the most influence on sales (even if $\beta_2 > \beta_1 > \beta_3$)

Data scaling/standardization

- The values of β coefficients represent the influence of each input feature on the target variable: $y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 + \cdots \beta_n X_n + \epsilon$
- We cannot compare the size of the various β coefficients if the input variables are measured on different scales
    - For example, number of bedrooms (X_1) in a house can be measured on a scale from 0 to 10. Area (X_2) can be measured on a scale from 50 to 1600 sqm. By looking at the values of coefficients β_1 and β_2 we cannot directly tell which independent variable has the most effect/influence on the dependent variable Y (house price)
- Rescale input features
    - Using MaxMinScaler, StandardScaler, RobustScaler shown in Lab 4
- With scaled/standardized variables, coefficients are directly comparable to one another, with the largest coefficient indicating which independent variable has the greatest influence on the dependent variable

When to rescale features?

- Min-max scaler rescales each feature individually into a given range, e.g. [0, 1]
- Standard scaler rescales each feature individually to make values have zero mean ($\mu = 0$) and unit variance ($\sigma^2 = 1$)
  - Assumes that feature fits a Gaussian distribution (bell curve) with a well-behaved mean and standard deviation
  - Centers data around zero
- Robust scaler rescales each feature individually to make values have zero median (median=0) and unit interquartile range (IQR=1)
  - Center data around zero
  - Robust to outliers

When to rescale features?

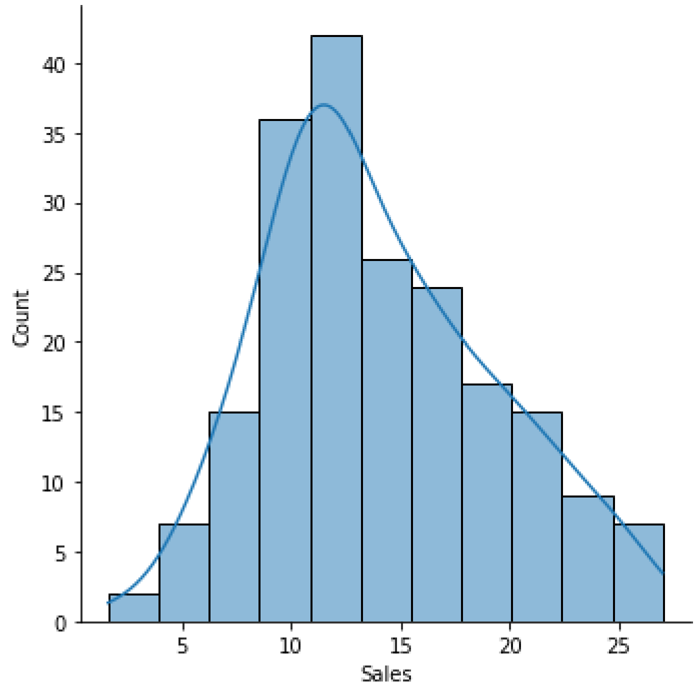
- Technically, feature scaling does not make a difference in linear regression (if the OLS method is used), however can be used to make β coefficients directly comparable to one another and reveal the influence of each feature on target
- In gradient descent based algorithms (such as SGDRegressor used in linear regression) feature scaling is needed to speed up the process of convergence

When to unskew features/target variable?

- Unskewing transformations attempt to make long-tail distribution of a variable symmetric as Gaussian/normal distribution
  - None of the previous scaling techniques changes the distribution shape
  - Unskewing transformation: BoxCox, Sqrt, Log
- Linear regression (using the OLS method) does not require feature and target variable distributions to be normal but requires normality of residuals
  - But, the presence of highly skewed features and/or target variable can, more likely, influence the distribution of residuals making them, in turn, non-normal
  - Thus, for very skewed features it might be a good idea to transform the data to eliminate the harmful effects

Linear Regression: Target variable distribution

- We prefer distribution of target variable to be symmetric (unskewed) => predictive algorithm will learn all sales values without bias
- Distribution plot of the target value: right skewed (long tail to the right)



```
import seaborn as sns
# distribution plot of the target variable
sns.displot(df['Sales'], kde=True)

# computing the p-value for the null-hypothesis that
this distribution is a normal distribution
from scipy import stats
_, p = stats.normaltest(y)
# p-value of 0.05 or greater means that the distribution
is a normal distribution
print(p) # => 0.025430412805360583, not normal distrib.
```

Distribution is skewed (not symmetrical) -- that it has a higher number of data points having low values, i.e., products with less Sales. So, when we train our model on this data, it will perform better at predicting the Sales of products with lower Sales as compared to those with higher Sales ➔ Solution: Unskew target variable (See Lab4)

Linear Regression: Standardize data

```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()

X_scaled = sc.fit_transform(X)
print(X_scaled[0:10])
```

```
[[ 0.96985227  0.98152247  1.77894547]
 [-1.19737623  1.08280781  0.66957876]
 [-1.51615499  1.52846331  1.78354865]
 [ 0.05204968  1.21785493  1.28640506]
 [ 0.3941822  -0.84161366  1.28180188]
 [-1.61540845  1.73103399  2.04592999]
 [-1.04557682  0.64390467 -0.32470841]
 [-0.31343659 -0.24740632 -0.87248699]
 [-1.61657614 -1.42906863 -1.36042422]
 [ 0.61604287 -1.39530685 -0.43058158]]
```

Target variable values are transformed to be unskewed

```
y_scaled, lambda_bc = boxcox(y)
print(y_scaled[0:10])
```

```
[8.62268888 4.96674482 4.54776147 7.61169565
5.8546036  3.6846038 5.47381496 5.95604738
2.55346522 5.04087376]
```

Transformed vector

Selected lambda (λ) value
(λ value can be used in reverse Box Cox transformation)

Standardized independent values

Linear Regression: Testing assumptions

```
# create a new model to be trained on scaled data
lregr_scaled = LinearRegression()

# split the dataset to X_scaled(_train, _val, _test), y_scaled(_train, _val, _test)

# train model (Fit linear model) and evaluate model β coefficients
model_scaled = lregr_scaled.fit(X_scaled_train, y_scaled_train)
# print model intercept
print("β0 =", model_scaled.intercept_)
# print model coefficients
print("[β1,β2,β3] =", model_scaled.coef_)

# estimate residuals
# predict and estimate residuals
y_scaled_pred = model_scaled.predict(X_scaled_val)
residuals_scaled = y_scaled_val - y_scaled_pred
print("Residuals:", residuals_scaled[:10])
```

```
β0 = 6.10377923104033
[β1,β2,β3] = [1.28474958  0.91806823 -0.01296285]
Residuals: [0.14146777  0.6577312  -4.39407098
0.21187781  0.09952789 -0.56718007
 -0.63370272  0.25698273  0.18450472  0.02443209]
```

"TV", "Radio", "Newspaper"

- Standardization changes the interpretation of coefficients.
- Reveals the “importance” (influence) of each independent variable in predicting the dependent variable.
- TV has the highest coefficient, thus can be inferred that it is the most important factor for increasing sales.

Standardized independent values

Linear Regression: Testing assumptions

SGDRegression object uses stochastic gradient descent method

```
from sklearn.linear_model import SGDRegressor
sgdr_scaled = SGDRegressor()

# train model (Fit linear model) and evaluate model β coefficients
model_sgdr = sgdr_scaled.fit(X_scaled_train, y_scaled_train)
# print model intercept
print("β0 =", model_sgdr.intercept_)
# print model coefficients
print("[β1,β2,β3] =", model_sgdr.coef_)

# estimate residuals
# predict and estimate residuals
y_sgdr_pred = model_sgdr.predict(X_scaled_val)
residuals_sgdr = y_scaled_val - y_sgdr_pred
print("Residuals:", residuals_sgdr[:10])
```

```
β0 = [6.08596269]
[β1,β2,β3] = [1.28237186  0.90856964 -0.00590614]
Residuals: [ 0.16586702  0.6620378  -4.36274108
0.23261795  0.13181271 -0.54991989
 -0.620275    0.26857194  0.19853716  0.0364894 ]
```

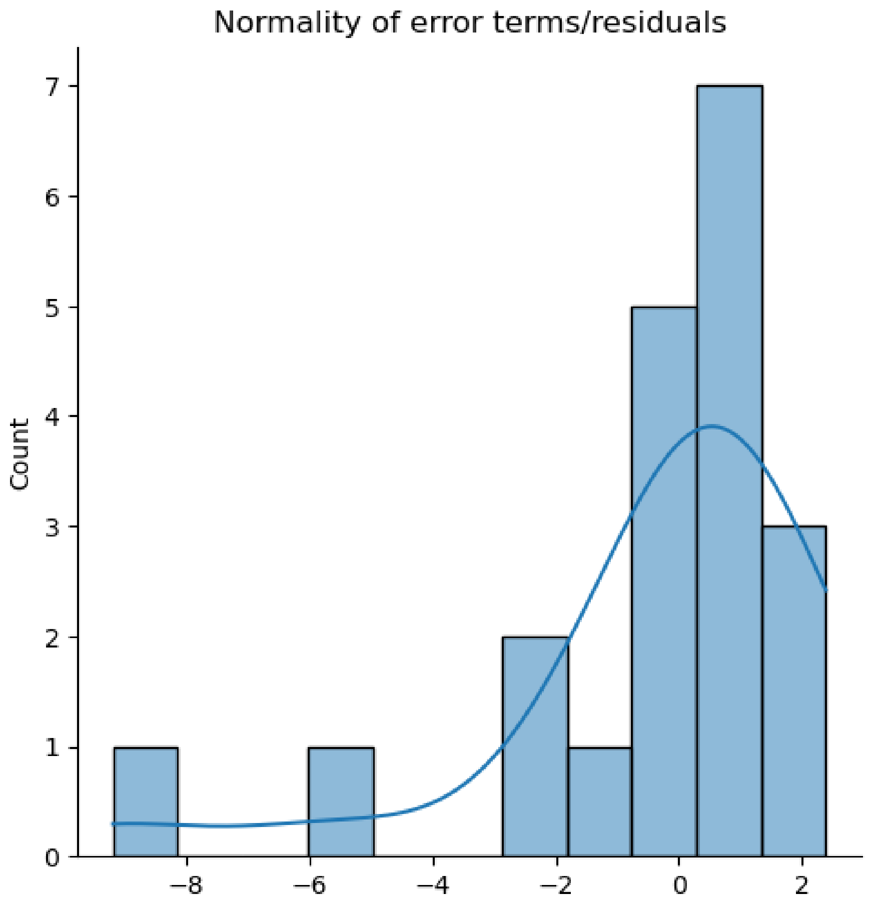
- SGRRegressor uses the iterative method gradient descent to estimate the coefficients
- The main reason why gradient descent is used for linear regression (compared to LinearRegressor) is the computational complexity: it's computationally cheaper (faster) to find the solution using the gradient descent in datasets with large number of features.

Non standardized independent values

Linear Regression: Testing assumptions

- Normality of residuals
  - Residuals (error terms) of unstandardized input does not seem to be normally distributed
  - Run normality check to test whether the residuals differ from a normal distribution

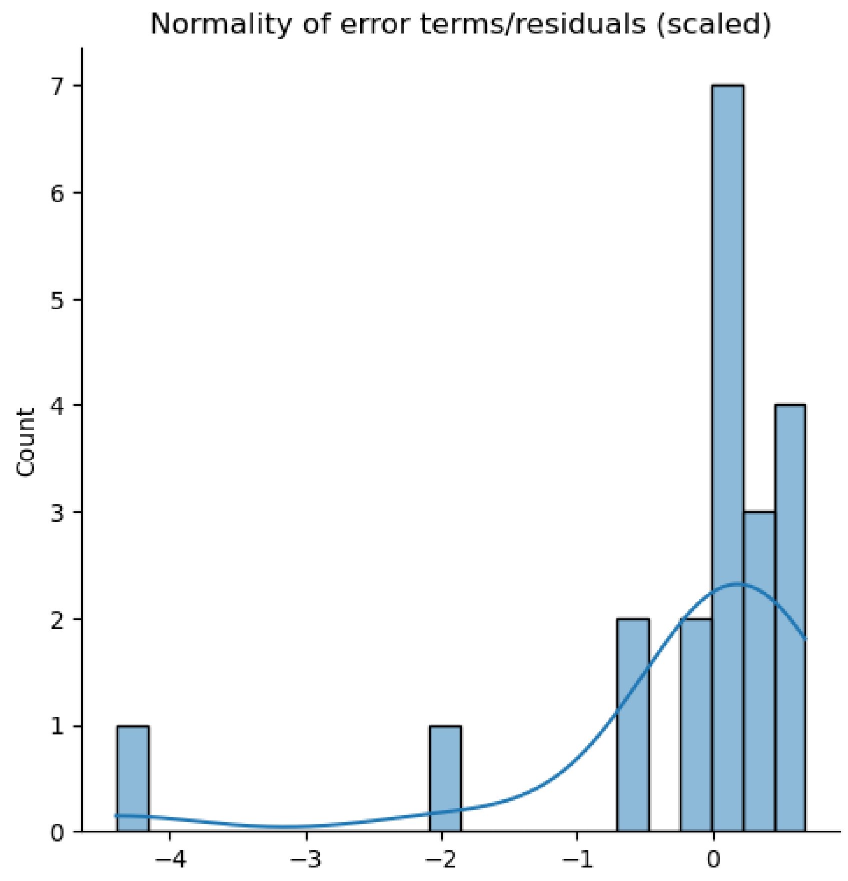
```
_, p = stats.normaltest(residuals)
# p-value of 0.05 or greater means that the
distribution is a normal distribution
print(p) # => 3.463801353587156e-10, residuals
differ from normal distrib.
```



Linear Regression: Testing assumptions

- Normality of residuals
  - Residuals (error terms) of standardized input does not seem to be normally distributed as well
  - Run normality check to test whether the residuals differ from a normal distribution

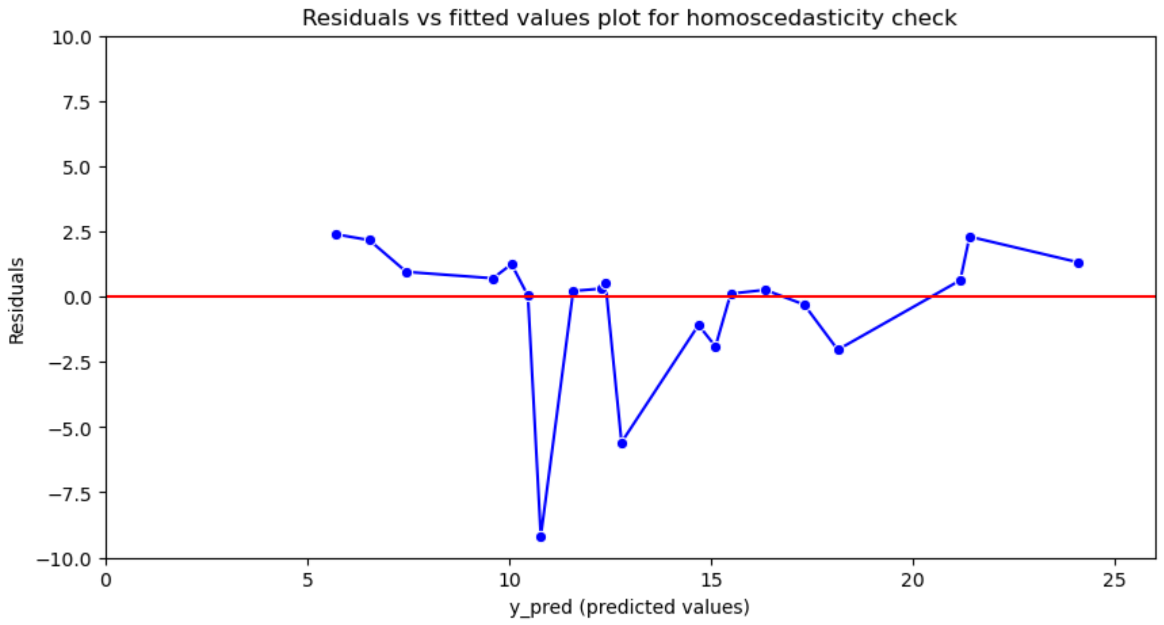
```
_, p = stats.normaltest(residuals_scaled)
# p-value of 0.05 or greater means that the
distribution is a normal distribution
print(p) # => 4.668655843075813e-16, residuals
differ from normal distrib.
```



Linear Regression: Testing assumptions

- Homoscedasticity
  - Data is not fully homoscedastic since the residuals (error) are not always constant along the values of the dependent variable

```
plt.figure(figsize=(10,5))
sns.lineplot(x=y_pred,y=residuals,marker='o',color='blue')
plt.xlabel('y_pred (predicted values)')
plt.ylabel('Residuals')
plt.ylim(-10,10)
plt.xlim(0,26)
sns.lineplot(x=[0,26],y=[0,0],color='red')
plt.title('Residuals vs fitted values plot
for homoscedasticity check')
plt.show()
```



Linear Regression: Model evaluation

- Model evaluation is a core part of building an effective machine learning model
- Evaluation metrics provide a measure of how good a model performs and how well it approximates the relationship between the dependent variable and the independent variables
- Some regression evaluation metrics:

minimize

  – MSE: Mean Squared Error $MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$
    - Error is squared: Large prediction errors are penalized
  – MAE: Mean Absolute Error $MAE = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i|$
    - Does not penalize large prediction errors
  – RMSE: Root Mean Squared Error $RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}$

maximize

  – R-squared (R2): a statistical measure of how close the data are to the fitted regression line on a convenient 0-1.0 scale (0: poor fitting, 1: perfect fitting)

n = number of data points
y_i = observed value i
$\hat{y}_i$ = predicted value i

Evaluation metrics discussion

- The idea behind the squared (MSE) and the absolute error (MAE) is to avoid mutual cancellation of the positive and negative errors
  - MSE and MAE have only non-negative values
- In MSE, error is squared => prediction error is being heavily penalized
  - In case of data outliers, MSE will become much larger compared to MAE
  - Based on the application, this property may be considered positive or negative:
    - For example, emphasizing large errors may be a desirable discriminating measure when evaluating models
- MAE preserves the same units of measurement
- In MSE, the unit of measurement is squared
- **RMSE** is used then to return the MSE error to the original unit by taking the square root of it, while maintaining the property of penalizing higher errors

Linear Regression: Evaluate model

```
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score

# prediction on validation data
# Model trained on unstandardized features and non-transformed target values
y_pred = model.predict(X_val)
print(y_pred[0:10])

# Mean Squared Error (MSE)
MSE = mean_squared_error(y_val, y_pred)
# Root Mean Squared Error (RMSE)
RMSE = np.sqrt(MSE)
r2 = r2_score(y_val, y_pred)
print("MSE:", MSE, ", RMSE:", RMSE , ", R2:", r2)
```

```
[15.48743552  6.53793858 10.78318566 11.58555633 21.17320803
 15.10974587  18.13802209  7.4522807  12.29402334 10.46455672]
MSE: 7.289025693003447 , RMSE: 2.699819566749498 , R2:
0.7703057423991149
```

Linear Regression: Evaluate model

```
# prediction on validation data
# Model trained on standardized features and (box-cox) transformed target values
y_scaled_pred = model_scaled.predict(X_scaled_val)
print(y_scaled_pred[0:10])
```

```
[6.59363166 3.65266796 4.93409818 5.26193715 8.44170914
 6.52322744 7.52455855 3.93214744 5.56765427 4.9794517 ]
MSE: 1.276160482816573 , RMSE: 1.129672732616209 , R2:
0.6769041124104881
```

```
# Mean Squared Error (MSE)
MSE_scaled = mean_squared_error(y_scaled_val, y_scaled_pred)
# Root Mean Squared Error (RMSE)
RMSE_scaled = np.sqrt(MSE_scaled)
r2_scaled = r2_score(y_scaled_val, y_scaled_pred)
print("MSE:", MSE_scaled, ", RMSE:", RMSE_scaled, ", R2:", r2_scaled)
```

Predicted values for Sales are transformed!!
(normal values for Sales are 5-26)

Note: if the target values used in training (y_train) were transformed, the predicted target values are also in the same transformation scale and need to revert them back to the original scale. Therefore, we apply the inverse box-cox transformation and the re-calculate the MSE, RMSE and r2:

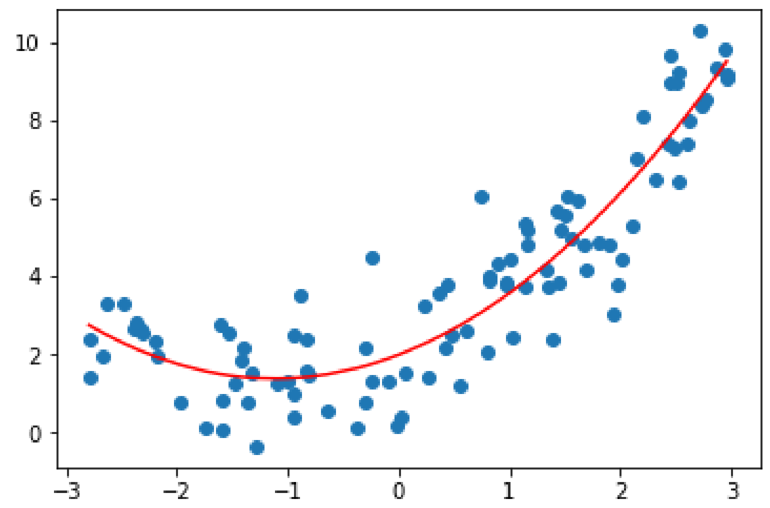
```
y_unscaled_pred = inv_boxcox(y_scaled_pred, lambda_bc)
```

Predicted sales:

```
[15.15165194  7.12667986 10.31242981 11.20594497 21.43577544
 14.93058249  18.2024254   7.77902257 12.06723119 10.43416896]
MSE: 6.105524793692385 , RMSE: 2.47093601570182 , R2:
0.8076006254035979
```

Polynomial (or non-linear) regression

- Let’s consider a case where during testing for linearity assumption between dependent and independent variables (scatter plot) a non-linear relationship (curve) is observed
- This is where polynomial Regression comes to the play which predicts the best fit line that follows the pattern (curve) of the data, as shown in the pic below:



- Polynomial Regression is generally used when the points in the data are not captured by the Linear Regression Model and the Linear Regression fails in describing the best result (in terms of low error) clearly

Polynomial regression

- Relationship between the independent variable(s) x and the dependent variable y are modelled as an nth degree polynomial in x

- Example (for one independent variable X):
  - quadratic model (2nd degree) : $y = \beta_0 + \beta_1 X + \beta_2 X^2 + \epsilon$
  - cubic model (3rd degree) : $y = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X^3 + \epsilon$

- Predictive performance of the model tends to increase (i.e. error is getting lower) as we increase the degree of the model

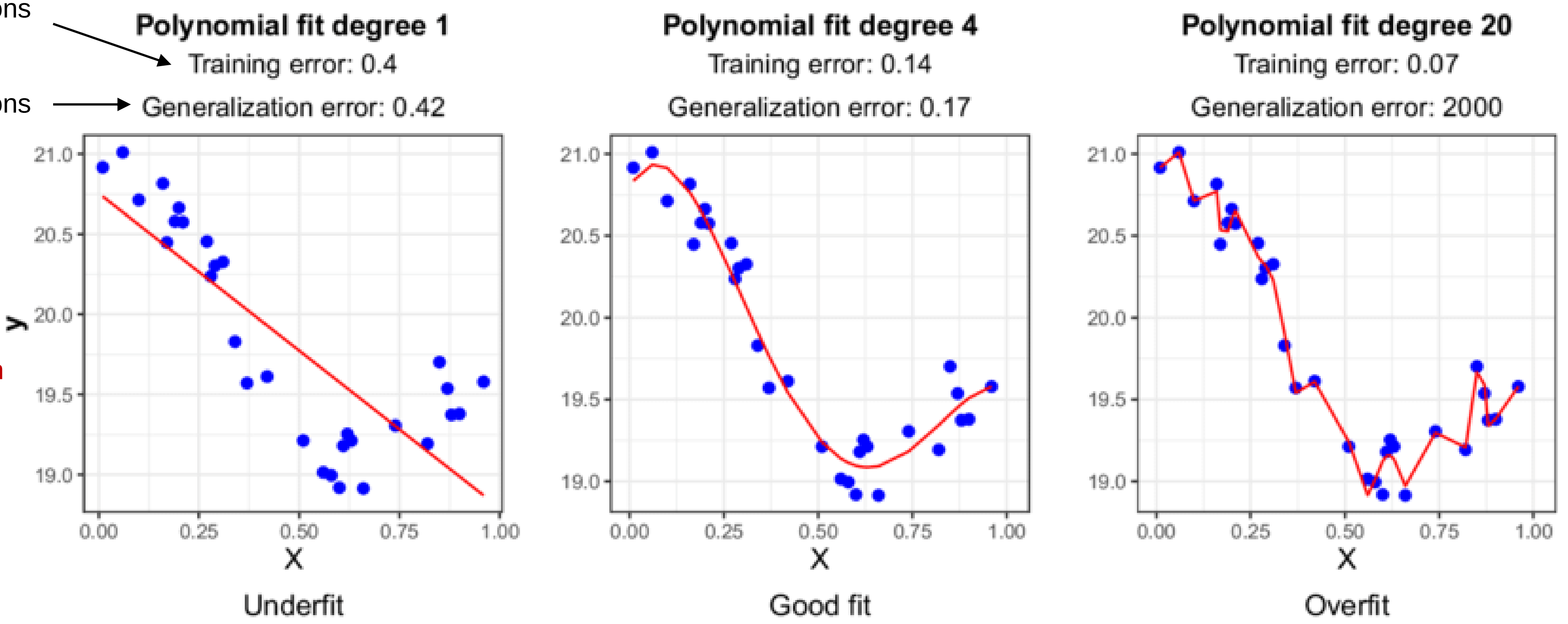
Polynomial regression: of which degree?

- Increasing the degrees of the model also increases the risk of over-fitting the data

Error when making predictions on the training dataset

Error when making predictions on the validation dataset (unseen data that was not used during training phase)

Common methodological mistake to make predictions on the training dataset which was used to train the model.



- The degree of the polynomial to fit is a hyperparameter that cannot be inferred while fitting the machine to the training set because it needs to be set prior the learning phase

How to find the right degree of the equation?

- In order to find the right degree for the model to prevent over-fitting or under-fitting, we can use any of the two approaches below:
  - Forward Degree Selection:
    - Start with a model of degree=1 and at each step gradually increase the model's degree until the best possible model (e.g. that minimizes MSE, RMSE) is reached
  - Backward Degree Selection:
    - Start with model of a large degree and at each step gradually decrease the model's degree until the best possible model is reached
  - At each step:
    - **Train** the model using the ***training dataset***
    - **Predict** the target value using the ***validation dataset***
    - Evaluate the performance of the model using any evaluation measure (MSE, RMSE, R2)
  - At the end, when the best model is chosen, evaluate its **final performance** by predicting the target value using the **testing dataset**.

Training Polynomial regression model using Linear Regressor

- Let's say we have dataset of one input feature, and we need to build a polynomial regression model of 3rd degree (cubic model)
  - $y = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X^3$
- Polynomial regression model can be trained using linear regressor (LR) since LR doesn't know that X^2 and X^3 are the square of X and the cube of X respectively, it just thinks they are another features
  - Prior running LR we expand the dataset, i.e. beyond the column X of the dataset, we create the extra columns X^2 and X^3
    - The unknown parameters to be estimated after training are $\beta_0, \beta_1, \beta_2, \beta_3$
- In a two-feature dataset X_1, X_2
  - 2nd degree polynomial model : $y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1^2 + \beta_4 \underbrace{X_1 X_2}_{\text{Interaction term}} + \beta_5 X_2^2$
    - You apply linear regression for five inputs: x_1, x_2, $x_1{}^2$, $x_1 x_2$, and $x_2{}^2$
    - Result of regression: the values of six parameters $\beta_0, \beta_1, \beta_2, \beta_3, \beta_4, \beta_5$

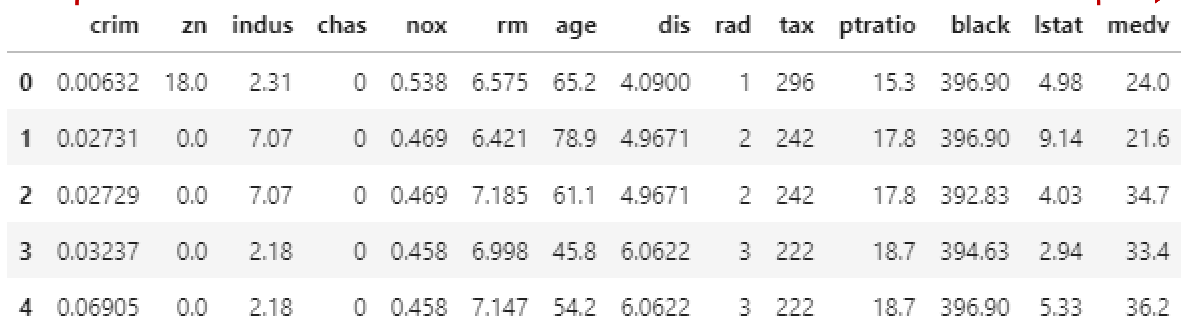
Is rescaling/unskewing needed?

- While creating power terms (e.g. X_1^2 , X_1^3), if X_1 is not centered first (using StandardScaler or RobustScaler), the squared and cubic terms will be highly correlated with X_1
- While creating interaction terms (e.g. X_1X_2), if both X_1 and X_2 are not centered first, some amount of collinearity will be induced, i.e. X_1X_2 will be correlated with X_1 and X_2
- Both situations can negatively affect the estimation of the β coefficients, therefore centering can be applied on all input features prior creating power and interaction terms (see here)
- Feature and target variable distributions are not required to be Gaussian, but unskewing transformation is generally recommended if distributions are heavily skewed

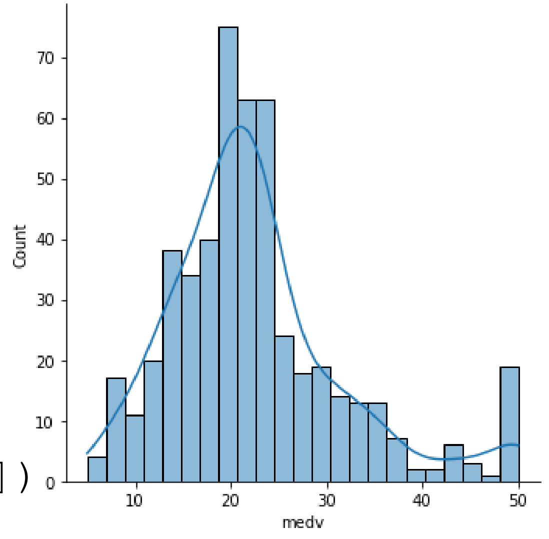
Polynomial Regression: Boston Housing Dataset

- Dataset: 506 houses by 13 features
- Objective: predict house prices

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

import pandas as pd
import seaborn as sns

boston = pd.read_csv('Boston.csv')
boston.head()

# distribution of the target values
sns.displot(boston['medv'], kde=True)
plt.show()

# statistical test
# p-value >= 0.05 means that the
distribution is a normal distribution
from scipy import stats
_, p = stats.normaltest(boston['medv'])
print(p) # => 1.76 e-20
```



	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	black	lstat	medv
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2



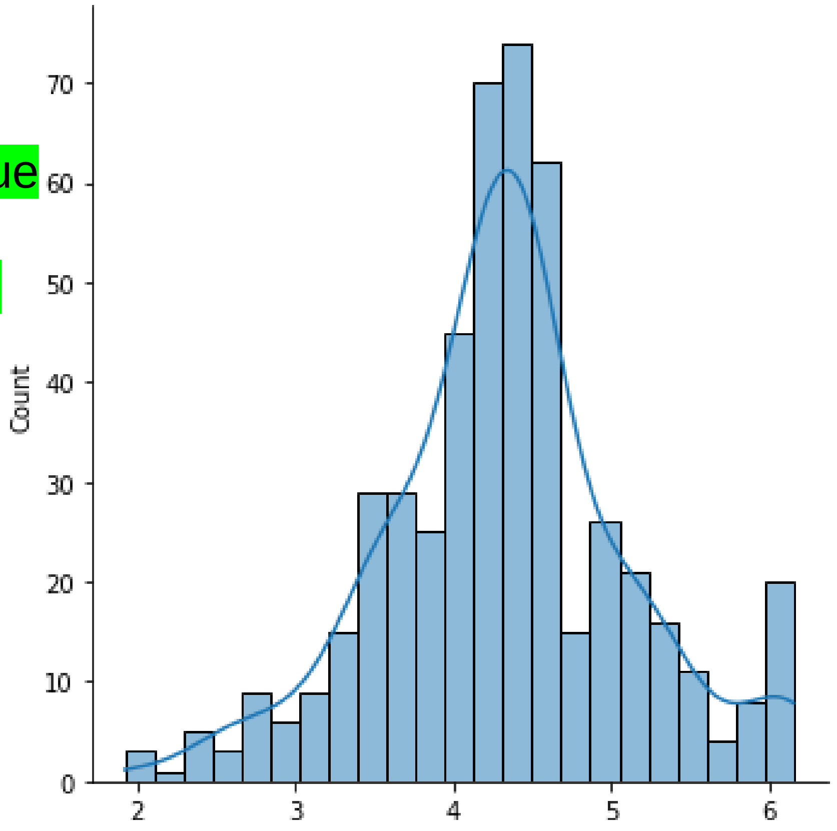
Distribution is skewed (not symmetrical): The mean is around 20 and the first part already looks quite like a normal distribution. But there is a large right tail of higher MEDV values. This could lead to the problem, that the model better predicts the MEDV values around the mean but is quite bad at predicting the MEDV values from the right tail. This is because most of the time the model sees values around the mean and is therefore biased towards these MEDV values. ➔ Solution: Unskew target variable (See Appendix)

Target value transformation

- Use boxcox transformation

```
# y - transformation (box cox)
from scipy.stats import boxcox
y_bc, lambda_bc = boxcox(boston['medv'])
_, p = stats.normaltest(y_bc)
print(p) # => 0.1046886692817602
sns.displot(y_bc, kde=True)
```

Transformed vector
Selected lambda (λ) value
(λ value can be used in
reverse Box Cox transf.)

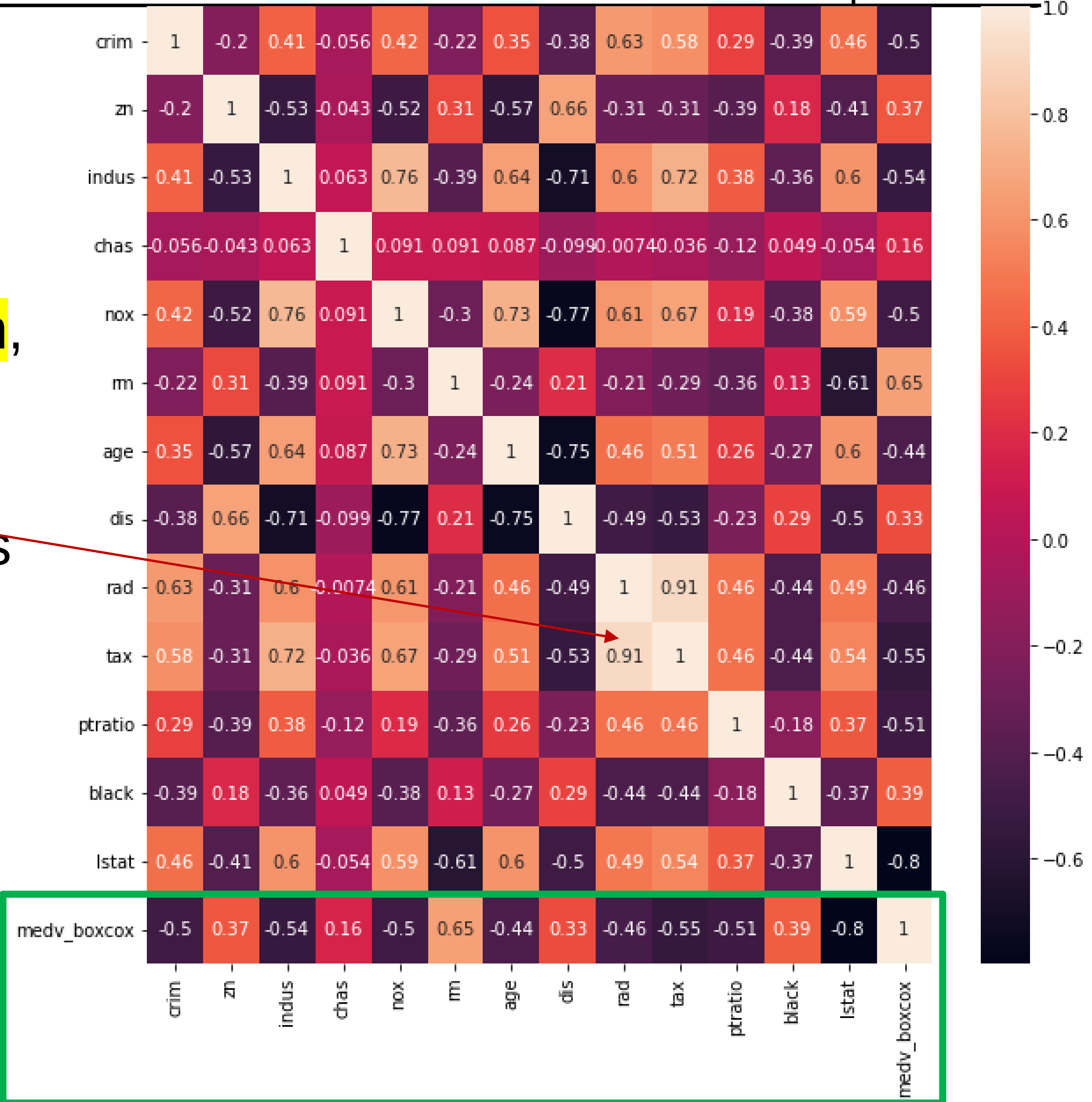


- Distribution of the transformed target variable
  - This distribution already looks quite similar to a normal distribution and achieves a p-value of 0.1, which is larger than 0.05. Therefore, we can say that the distribution equals a normal distribution
- Create boston2 (copy of boston) having target value unskewed

```
boston2 = boston.copy()
boston2['medv_boxcox']=y_bc
boston2.drop(columns=['medv'], inplace=True)
```

Feature Selection – Correlation matrix

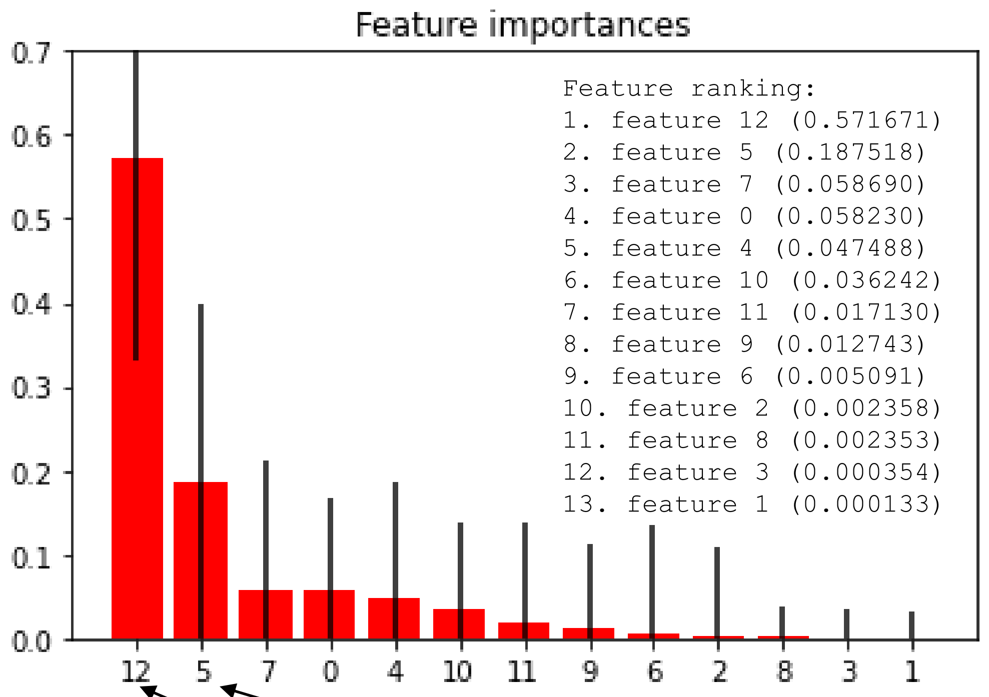
- Create correlation matrix of the boston2 dataframe
- Observations:
  - As we can see, only the features rm, and lstat are highly correlated with the output variable medv_boxcox
  - Avoid using high correlated features together to avoid multi-collinearity
    - rad / tax are strongly correlated
    - dis / indus / age are strongly correlated



	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	black	lstat	medv_boxcox
crim	1	-0.2	0.41	-0.056	0.42	-0.22	0.35	-0.38	0.63	0.58	0.29	-0.39	0.46	-0.5
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	0.37
indus	0.41	-0.53	1	0.063	0.76	-0.39	0.64	-0.71	0.6	0.72	0.38	-0.36	0.6	-0.54
chas	-0.056	-0.043	0.063	1	0.091	0.091	0.087	-0.099	-0.0074	-0.036	-0.12	0.049	-0.054	0.16
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	-0.5
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	0.65
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	-0.44
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	0.33
rad	0.63	-0.31	0.6	-0.0074	0.61	-0.21	0.46	-0.49	1	0.91	0.46	-0.44	0.49	-0.46
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	-0.55
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
black	-0.39	0.18	-0.36	0.049	-0.38	0.13	-0.27	0.29	-0.44	-0.44	-0.18	1	-0.37	0.39
lstat	0.46	-0.41	0.6	-0.054	0.59	-0.61	0.6	-0.5	0.49	0.54	0.37	-0.37	1	-0.8
medv_boxcox	-0.5	0.37	-0.54	0.16	-0.5	0.65	-0.44	0.33	-0.46	-0.55	-0.51	0.39	-0.8	1

Feature Selection – Importance

```
# Feature Importance using ExtraTreeClassifier
from sklearn.ensemble import GradientBoostingRegressor
# Build an estimator and compute the feature importances
estimator = GradientBoostingRegressor(n_estimators=100, random_state=0)

X = boston.values[:,0:-1]
y = boston.values[:,-1]

estimator.fit(X,y)
# Lets get the feature importances.
# Features with high importance score higher.
importances = estimator.feature_importances_

std = np.std([tree[0].feature_importances_ for tree in estimator.estimators_], axis=0)
indices = np.argsort(importances)[::-1]

# Print the feature ranking
print("Feature ranking:")
for f in range(X.shape[1]):
    print("%d. feature %d (%f)" % (f + 1, indices[f], importances[indices[f]]))

# Plot the feature importances
plt.figure()
plt.title("Feature importances")
plt.bar(range(X.shape[1]), importances[indices], color="r", yerr=std[indices], align="center")
plt.xticks(range(X.shape[1]), indices)
plt.xlim([-1, X.shape[1]])
plt.ylim([0, 0.7])
plt.show()
```

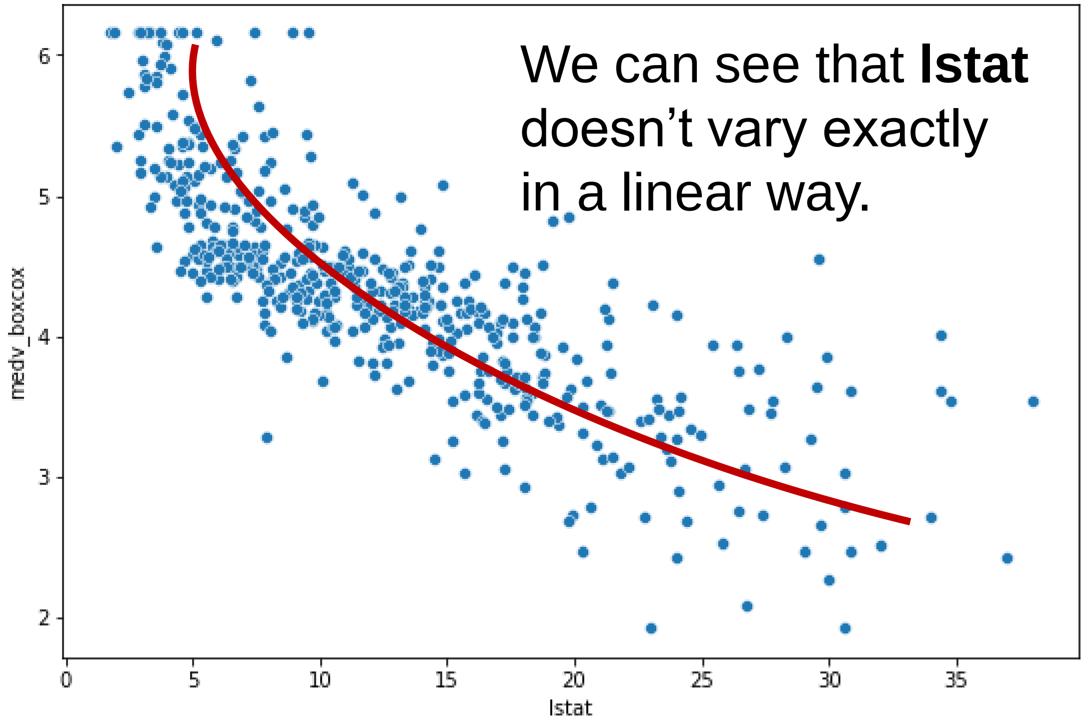


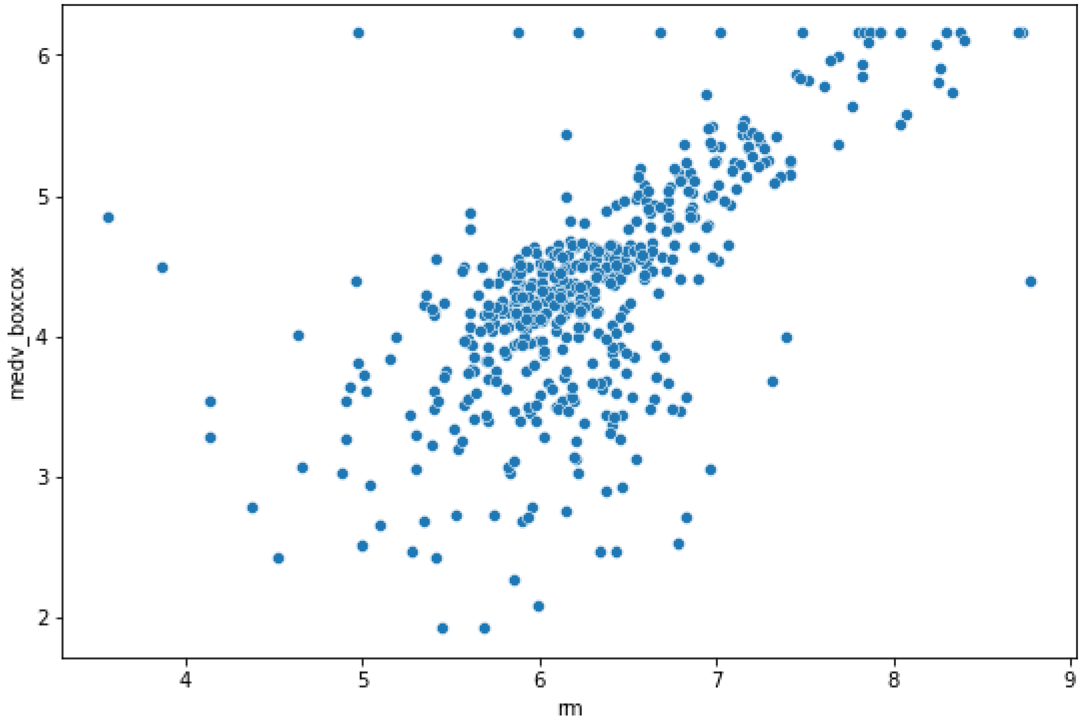
```
Feature ranking:
1. feature 12 (0.571671)
2. feature 5 (0.187518)
3. feature 7 (0.058690)
4. feature 0 (0.058230)
5. feature 4 (0.047488)
6. feature 10 (0.036242)
7. feature 11 (0.017130)
8. feature 9 (0.012743)
9. feature 6 (0.005091)
10. feature 2 (0.002358)
11. feature 8 (0.002353)
12. feature 3 (0.000354)
13. feature 1 (0.000133)
```

As we can see, the features lstat, and rm achieve the highest importance among all features for predicting the target variable medv_boxcox

Feature Selection

- We decide to keep only these two features (lstat and rm)
- We use both Linear and Polynomial regression to build a predictive model for predicting the target variable

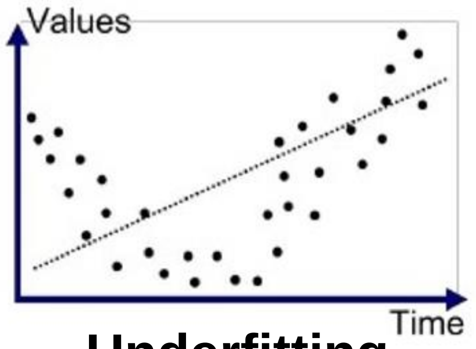




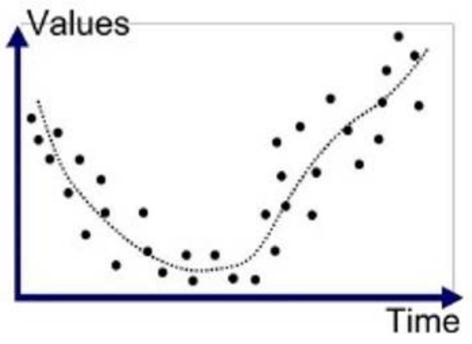
Hyperparameter/parameter tuning

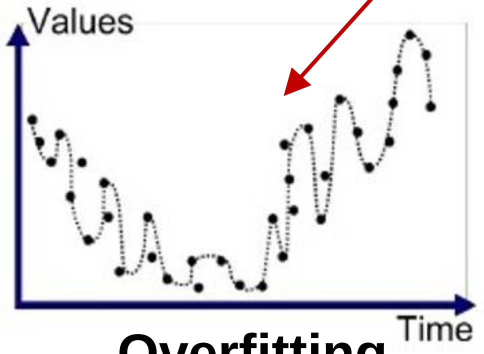
- Tuning (training) the hyperparameters and the parameters of a predictive modelling technique (predictor / classifier) and testing its performance on the same data is a methodological mistake
  - High accuracy on seen data
  - May fail to predict / classify unseen data

Overfitting



Underfitting





Overfitting

- Solution: split available dataset into 3 parts; training, validation and testing datasets

Data Preparation

- Extract features and the target value to X and y respectively from boston2 (involving transformed target variable)

```
# extract 2 features
X = boston2[['lstat', 'rm']]
# extract target variable
y = boston2['medv_boxcox']
```

- Split train/test dataset

```
from sklearn.model_selection import train_test_split

# splits dataset to training, validation and testing datasets: 80% / 10% / 10%
X_train, X_2, y_train, y_2 = train_test_split(X, y, train_size = 0.8)
X_val, X_test, y_val, y_test = train_test_split(X_2, y_2, train_size = 0.5)
```

Linear Regression

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

lr = LinearRegression()

# model training
lin_model = lr.fit(X_train, y_train)

# model evaluation for validation set
y_val_predict = lin_model.predict(X_val)

# root mean square error of the model
rmse = (np.sqrt(mean_squared_error(y_val, y_val_predict)))

# r-squared score of the model
r2 = r2_score(y_val, y_val_predict)

print("Model performance on validation dataset")
print("--------------------------------------")
print('RMSE is {}'.format(rmse))
print('R2 score is {}'.format(r2))
```

```
Model performance on validation dataset
--------------------------------------
RMSE is 0.48347898587331134
R2 score is 0.578350023179578
```

Linear Regression (results on original scale)

```
Model performance on validation dataset
--------------------------------------
RMSE is 0.48347898587331134
R2 score is 0.578350023179578
```

Model Sales predictions are in the Box-Cox scale
We can transform them back to the original scale

```
from scipy.special import inv_boxcox
# inverse Box Cox transformation
y_val_init = inv_boxcox(y_val, lambda_bc)
y_pred_init = inv_boxcox(y_val_predict, lambda_bc)
rmse_init = (np.sqrt(mean_squared_error(y_val_init,
y_pred_init)))
r2_init = r2_score(y_val_init, y_pred_init)
print('RMSE is {}'.format(rmse_init))
print('R2 score is {}'.format(r2_init))
```

```
RMSE is 4.770862055815939
R2 score is 0.6900277569655442
```

Model performance if target is not transformed?

- Extract features and target value from the original boston dataframe (target variable was not transformed)

```
# extract 2 features
X_original = boston[['lstat', 'rm']]
# extract target variable
Y_original = boston['medv']
```

- Split X_original, y_original into training, validation, testing
- Train linear regression model using X_train_orig, y_train_orig
- Use X_val_orig to predict y_val_predict_orig
- Evaluate model performance using RMSE, R2

```
RMSE is 5.203457199881524
R2 score is 0.631266105649837
```

- Using boston2 (with box cox applied on y):

```
RMSE is 4.770862055815939
R2 score is 0.6900277569655442
```

- Better performance is experienced when target variable is unskewed

Linear Regression (with hyperparameters)

- No hyperparameters used thus far: `lr = LinearRegression()`
- If hyperparameters are to be used, they need to be set prior training
- Linear regression can set the `fit_intercept` hyperparameter
  - The intercept term (often labeled the constant β_0) is the expected mean value of Y when all X=0
  - Default value is true: β_0 is part of the model
- Set `lr = LinearRegression(`**`fit_intercept=False`**`)` and follow the process (training, prediction on validation dataset, model evaluation) using the boston2 dataset
  - Slight improvement of the model

```
Model performance on validation dataset (without intercept term)
----------------------------------------------------------------
RMSE is 0.4449854229486139
R2 score is 0.642818921239515
```

Problem with dataset splitting

- Results (RMSE, R2) may depend on a particular choice (split) for the training, validation & testing datasets
  - What if split isn't random?
  - What if one subset of our data has only data from a certain category?

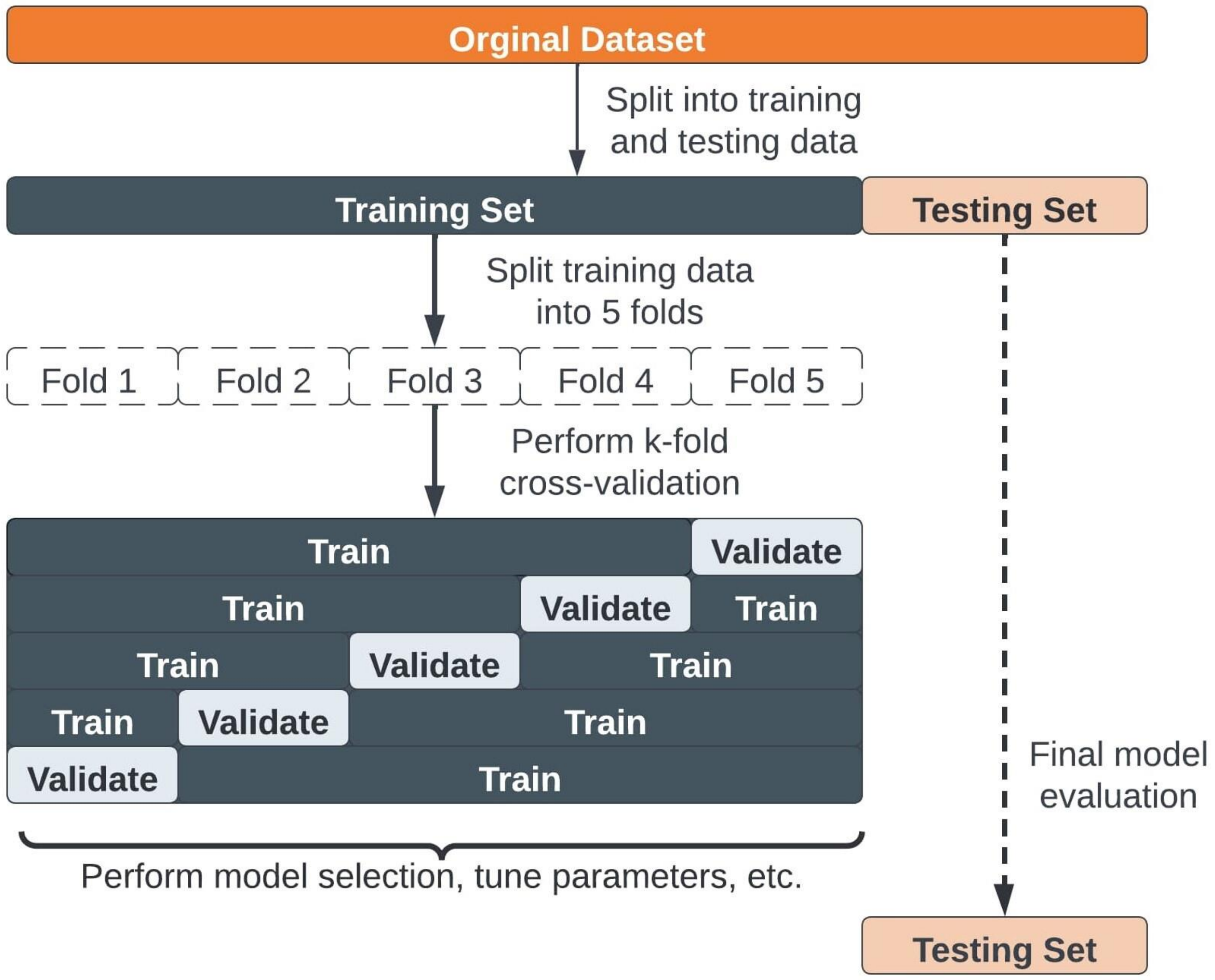
  → Overfitting again !!!
- Solution: Repeat the process of randomly splitting data into subsets and average error results => **Cross Validation (CV)**

K-folds Cross Validation

- Prior running Cross-Validation, split initial dataset into train/test
- Split train dataset randomly into k subsets called folds
- Repeat:
  - Train model on k-1 folds
  - Use k^{th} fold as validation dataset to measure model performance
    - Measure score (e.g. RMSE, R2 for regression, accuracy, f1-score for classification)
- Until each of k folds has served as validation fold
- Combine (average) k recorded scores to estimate the error/accuracy of the model: cross-validation score
- Modify model hyperparameters and re-run cross validation to find the best hyperparameter values
- Test dataset is used for the final evaluation of the model with the best model parameters and hyperparameters

Cross validation

k-folds Cross Validation



k-folds Cross Validation

```
from sklearn import model_selection

# initialize k-folds cross-validator, with k=10
kfold = model_selection.KFold(n_splits=10)

# perform cross validation using k-folds cv
lr = LinearRegression()
rmse = model_selection.cross_val_score(lr, X_train,
y_train, cv=kfold, scoring = "neg_root_mean_squared_error")

print("Mean RMSE:", -rmse.mean())
print("Standard deviation RMSE:", rmse.std())
```

```
Mean RMSE: 0.43850956855296425
Standard deviation RMSE: 0.06639513033656468
```

- The unified sklearn scoring API always maximizes the score, so scores which need to be minimized like RMSE are negated in order for the unified scoring API to work correctly

Polynomial Regression (degree = 2)

```python
from sklearn.preprocessing import PolynomialFeatures

poly_features = PolynomialFeatures(degree=2)

# transform training set features to higher degree features
X_train_poly = poly_features.fit_transform(X_train)
print(X_train[0:5])
print(X_train_poly[0:5])


# fit the transformed features to Linear Regression
poly_model = LinearRegression()
# train the model
poly_model.fit(X_train_poly, y_train)
```

convert the original features (X_train) into their higher order terms (X_train_poly) via the PolynomialFeatures class

```
     lstat     rm
33   18.35  5.701
283   3.16  7.923
418  20.62  5.957
502   9.08  6.120
402  20.31  6.404
```

lstat rm $lstat^2$ lstat * rm rm^2

```
[[  1.      18.35     5.701    336.7225   104.61335   32.501401]
 [  1.       3.16     7.923      9.9856    25.03668   62.773929]
 [  1.      20.62     5.957    425.1844   122.83334   35.485849]
 [  1.       9.08     6.12      82.4464    55.5696    37.4544  ]
 [  1.      20.31     6.404    412.4961   130.06524   41.011216]]
```

Bias column: Feature in which all polynomial powers are zero. Acts as an intercept term in a linear model.

```python
# transform validation set features to higher degree features
X_val_poly = poly_features.fit_transform(X_val)

# predicting on validation dataset
y_val_predict = poly_model.predict(X_val_poly)
```

Polynomial Regression (degree = 2)

```
# evaluating the model on validation dataset
rmse_val = np.sqrt(mean_squared_error(y_val, y_val_predict))
r2_val = r2_score(y_val, y_val_predict)

print("Model performance on validation dataset")
print("--------------------------------------------")
print("RMSE is {}".format(rmse_val))
print("R2 score is {}".format(r2_val))
```

```
The model performance for the validation set
-------------------------------------------
RMSE of training set is 0.4235493601190515
R2 score of training set is 0.676402665405311
```

We can observe that the RMSE error has reduced after using polynomial regression as compared to linear regression. However, CV needs to be performed along with hyperparameter tuning:

- **explore different polynomial degrees beyond 2**
- **keep interaction_only features (e.g. remove lstat^2 and rm^2), default is False**
- **try without include_bias, default is True**

Exhaustive param search: GridSearchCV

- Exhaustive search with Cross-Validation over specified hyper parameter combination for a predictive technique [see here]
- Grid of parameter values is specified with the param_grid list
  - For example, for Polynomial Features with `degree`, `interaction_only` and `include_bias` hyperparameters:

```
param_grid = [
    { "degree": [1, 2, 3, 4], "interaction_only": [True, False] },
    { "degree": [1, 2, 3], "include_bias ": [True, False] }
]
```

  - specifies that two grids should be explored:
    - combination of degree values [1, 2, 3, 4] and interaction_only True/False,
    - combination of degree values [1, 2, 3, 4] and include_bias True/False

```
grid = GridSearchCV(Predictive Technique, parameter grid, scoring = 'neg_root_mean_squared_error',
cv=10, n_jobs=-1) # default cv value is 5, n_jobs = 1 mean run using all processors
grid.fit(X_train, y_train)
```

n_jobs parameter is provided by many sklearn estimators (e.g. in RandomForest, GridsearchCV, Extratree, etc.). It accepts number of cores to use for parallelization. If value of -1 is given then it uses all cores. It uses joblib parallel processing library for running things in parallel in background. Therefore, I would like to recommend to you to use **n_jobs=-1** where applicable to speed-up your computations.

Pipeline

- Previous (polynomial regression) process involves two sequential steps:
  - Create polynomial features
  - Run linear regression
- With scikit learn, it is possible to create a pipeline combining these two steps (PolynomialFeatures and LinearRegression).
- We can also run this pipeline with a combination of hyperparameters of both steps through GridSearchCV

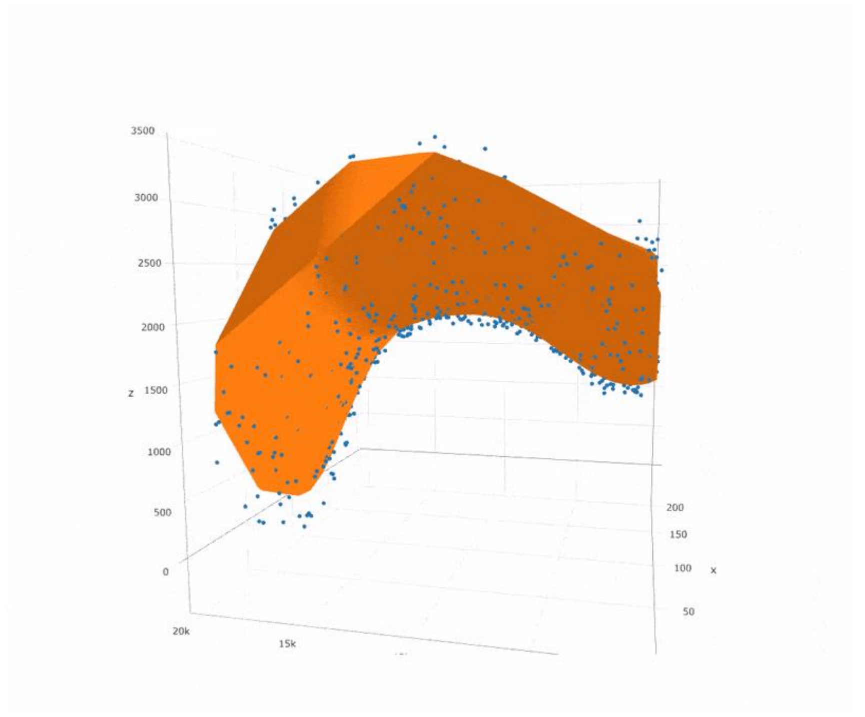
Polynomial regression: Pipeline with GridSearchCV

```
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV

# Define a pipeline involving PolynomialFeatures
# and LinearRegression steps
pf = PolynomialFeatures()
lr = LinearRegression()
# name each step
pipe = Pipeline(steps=[("poly", pf), ("linear", lr)])

# Parameters of pipelines can be set using '__' separated parameter names:
param_grid = [
    { "poly__degree": [1, 2, 3, 4, 5], "poly__interaction_only": [True, False], "poly__include_bias": [True, False] },
    { "poly__degree": [1, 2, 3, 4], "poly__interaction_only": [True, False], "poly__include_bias": [True, False], "linear__fit_intercept": [True, 
False] }
]
# make grid object for GridSearchCV and fit the dataset
search = GridSearchCV(pipe, param_grid, scoring = 'neg_root_mean_squared_error', cv=10, n_jobs=-1)
search.fit(X_train, y_train)

# print results
print(" Results from Grid Search " )
print("\n The best estimator across ALL searched params:\n", search.best_estimator_)
print("\n The best score across ALL searched params:\n", -search.best_score_)
print("\n The best parameters across ALL searched params:\n", search.best_params_)
```

```
The best score across ALL searched params:
0.402418593698313

The best parameters across ALL searched params:
{'poly__degree': 2, 'poly__include_bias': True,
'poly__interaction_only': True}
```

Support Vector Regression

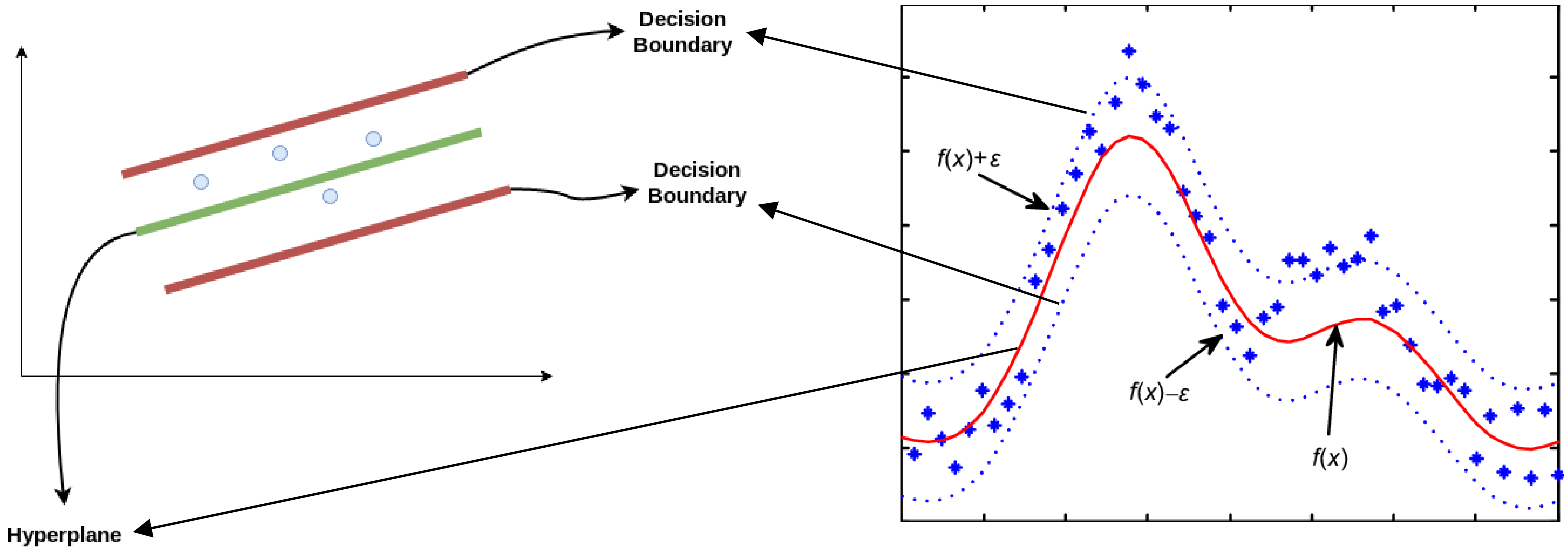
- Basic idea of support vector regression
  - Find optimal hyperplane that has maximum number of points



- Hyperplane: A hyperplane is a decision surface that is used to predict the continuous output. The data points on either side of the hyperplane that are closest to the hyperplane are called Support Vectors. These are used to plot the required surface that shows the predicted output of the algorithm

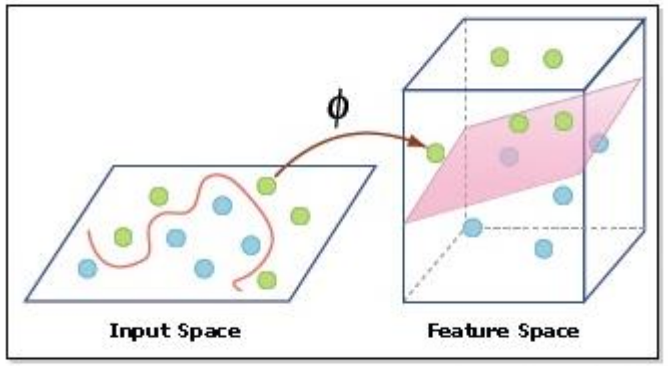
Support Vector Regression

- Decision Boundaries: These are the two surfaces that are drawn around the hyperplane at a distance of ε (epsilon).
  - SVR basically considers the points that are within the decision boundaries
  - Best fit: the hyperplane that has a maximum number of points.



Support Vector Regression

- Kernel: A kernel is a set of mathematical functions that takes data as input and transform it into the required form. These are generally used for finding a better hyperplane in a higher dimensional space
  - The most widely used kernels include linear, polynomial (poly), radial basis function (rbf) and sigmoid. By default, RBF is used as the kernel. Each of these kernels are used depending on the dataset.



Support Vector Regression

- SVR important hyperparameters:
  - kernel: default value is rbf
  - C: Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. Default value is 1.0
  - epsilon: boundary threshold (ε), default value is 0.1
  - gamma: kernel coefficient for rbf, poly and sigmoid, default value is ‘scale’
  - degree: degree of the polynomial kernel (poly)
- In distance-based regression algorithms (such as Support Vector Regressor - SVR) that use (Euclidean or Manhattan) distances between data points, feature scaling is needed so that all the features contribute equally to the distance otherwise distance may be dominated by features with larger scales
    - E.g. $Distance(X_1, X_2) = \sqrt{(3-1027)^2 + (4-2123)^2}$ distance is dominated by X_2 values

Support Vector Regression

```
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.preprocessing import RobustScaler

# scale the input features
scaler = RobustScaler()
X_scaled = scaler.fit_transform(X)

svr = SVR()
# model training
svr_model = svr.fit(X_train, y_train)
# model evaluation for testing set
y_test_predict = svr_model.predict(X_test)
# root mean square error of the model
rmse = (np.sqrt(mean_squared_error(y_test, y_test_predict)))
# r-squared score of the model
r2 = r2_score(y_test, y_test_predict)

print("Model performance on testing dataset")
print("--------------------------------------")
print('RMSE is {}'.format(rmse))
print('R2 score is {}'.format(r2))
```

```
The model performance for validation set
--------------------------------------
RMSE is 0.41430394276538857
R2 score is 0.690375720858043
```

We can observe that the RMSE error (with default hyperparameter values) has slightly increased compared to polynomial regression. However, we need to re-evaluate the model with different combinations of hyperparameters (kernel, C, gamma, epsilon)

SVR with GridSearchCV

- Exhaustive search over specified parameter values for an estimator

```
from sklearn.model_selection import GridSearchCV
# Define a pipeline involving Robust Scaler and SVR
rs = RobustScaler()
svr = SVR()
pipe_svr = Pipeline(steps=[("scaler", rs), ("svr", svr)])

# parameter grid (as described in previous slide)
parameter_grid = [
 {'svr__C': [1, 10, 100, 1000], 'svr__kernel': ['linear']},
 {'svr__C': [1, 10, 100, 1000], 'svr__gamma': [0.001, 0.0001], 'svr__kernel': ['rbf']},
 {'svr__C': [1, 10, 100, 1000], 'svr__degree': [1, 2, 3, 4, 5, 6], 'svr__kernel': ['poly']}
]

# make grid_SVC object for GridSearchCV and fit the dataset
grid_SVR = GridSearchCV(pipe_svr, parameter_grid, scoring = 'neg_root_mean_squared_error', n_jobs=-1)
grid_SVR.fit(X_train, y_train)

# print results
print(" Results from Grid Search " )
print("\n The best estimator across ALL searched params:\n", grid_SVR.best_estimator_)
print("\n The best score across ALL searched params:\n", -grid_SVR.best_score_)
print("\n The best parameters across ALL searched params:\n", grid_SVR.best_params_)
```

```
The best estimator across ALL searched params:
Pipeline(steps=[('scaler', RobustScaler()), ('svr',
SVR(C=1000, gamma=0.001))])

The best score across ALL searched params:
0.40327382455884486

The best parameters across ALL searched params:
{'svr__C': 1000, 'svr__gamma': 0.001, 'svr__kernel':
'rbf'}
```

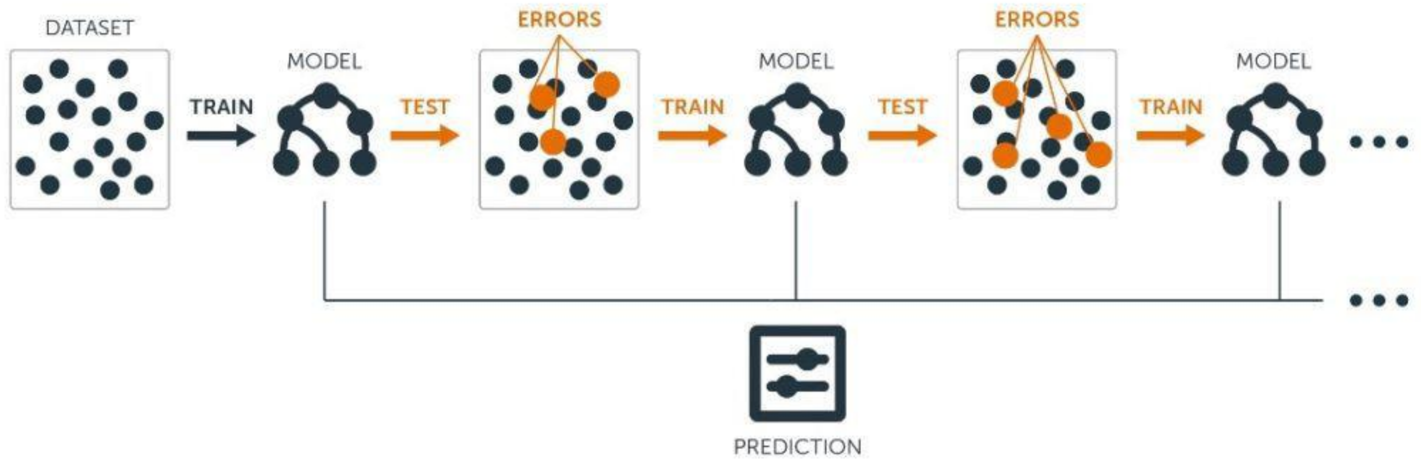
SVR model is still unable to outperform the polynomial model. Both achieve similar RMSE.

Ensemble learning

- Ensemble learning: train multiple ML algorithms (learners) and combine their predictions in some way
- Ensemble model is a model that consists of many base (weak) models
- Tends to make more accurate predictions than individual (weak) base models
- We have three kinds of ensemble methods using:
  - Sequential Homogeneous Learners (Boosting)
  - Parallel Homogeneous Learners (Bagging)
  - Parallel Heterogeneous Learners (Stacking)

Basic Types of Ensemble Learning

- Sequential Ensemble Learning (boosting)
  - Key ideas:
    - base learners are dependent on the results from previous base learners
    - every subsequent base model corrects the prediction made by its predecessor fixing the errors in it
    - overall performance can be gradually increased
  - Cons: tends to overfit the training data
  - Examples: AdaBoost, Stochastic Gradient Boosting, XGBoost, CatBoost

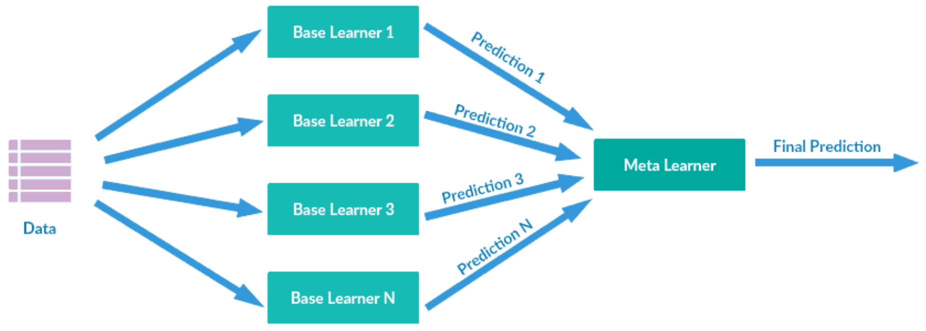


Basic Types of Ensemble Learning

- Parallel ensemble learning using homogeneous learners (also called bagging)
  - all base learners are homogeneous (same machine learning algorithm) and execute in parallel on different random subsets of the original dataset
  - no dependency between the base learners
  - results of all base models are combined in the end (using averaging for regression and voting for classification problems)
    - Averaging: every learner make a prediction (predicted value) for each data point, and the final predicted value for that point is the average of all predicted values
    - Voting: every learner makes a prediction (votes) for each data point (row in dataset) to which category should be assigned to and the final output prediction for that point is the category that receives more than half (or the majority) of the votes
  - See more here
  - Examples: sklearn.ensemble.BaggingRegressor, sklearn.ensemble.RandomForestRegressor

Basic Types of Ensemble Learning

- Parallel ensemble learning using heterogeneous learners (also called stacking)
  - all base learners are heterogeneous (different machine learning algorithm) and execute in parallel
    - Base Learners are trained using the available data
  - meta learner combines predictions of base learners
    - Meta Learner is trained to make a final prediction using the Base Learners' predictions on the input data – base models' predictions are used as input features to meta learner
  - stacking obtains better performance results than any of the individual weak learners



  - Example: [sklearn.ensemble.StackingRegressor]()

Random Forest Regression

- A Random Forest is a bagging ensemble technique
- Performs both regression and classification tasks with the use of multiple decision trees as base models
- The name “Random Forest” comes from the bootstrapping idea of data randomization (training datasets for each tree taken from random subsets of the initial training dataset) and building multiple Decision Trees (Forest)
- RandomForestRegressor class
  - sklearn.ensemble.RandomForestRegressor
  - More info here

Is rescaling/unskewing needed?

- Ensemble methods (Random Forest, XGBoost, AdaBoost) do not require feature rescaling to be performed as they are not sensitive to the variance in the data
- A skewed dependent variable is not necessarily a problem for ensemble methods per se – there are no assumptions as for example the normality of residuals (errors) that need to be met like in the linear model

Random Forest Regression

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score

rf = RandomForestRegressor()

# model training
rf_model = rf.fit(X_train, y_train)

# model evaluation for validation set
y_val_predict = rf_model.predict(X_val)
# root mean square error of the model
rmse = (np.sqrt(mean_squared_error(y_val, y_val_predict)))

# r-squared score of the model
r2 = r2_score(y_val, y_val_predict)

print("Model performance on validation dataset")
print("--------------------------------------")
print('RMSE is {}'.format(rmse))
print('R2 score is {}'.format(r2))
```

```
Model performance on validation dataset
--------------------------------------
RMSE is 0.4995876150283884
R2 score is 0.5497847487449045
```

Random Forest Regressor with the default hyper parameters achieves slightly worse performance compared to the polynomial model but a combination of hyper parameters needs to be considered.

RandomForestRegressor with GridSearchCV

```
# Number of trees in random forest
n_estimators = [int(x) for x in np.linspace(start = 200, stop = 1000, num = 10)]
# Number of features to consider at every split
max_features = ['auto', 'sqrt']
# Maximum number of levels in tree
max_depth = [int(x) for x in np.linspace(10, 110, num = 11)]
max_depth.append(None)
# Minimum number of samples required to split a node
min_samples_split = [2, 5, 10]
# Minimum number of samples required at each leaf node
min_samples_leaf = [1, 2, 4]
# Method of selecting samples for training each tree
bootstrap = [True, False]
# Create the random grid
parameter_grid = {'n_estimators': n_estimators,
               'max_features': max_features,
               'max_depth': max_depth,
               'min_samples_split': min_samples_split,
               'min_samples_leaf': min_samples_leaf,
               'bootstrap': bootstrap}

rf = RandomForestRegressor()
# make grid_SVC object for GridSearchCV and fit the dataset
grid_SVR = GridSearchCV(rf, parameter_grid, scoring = 'neg_root_mean_squared_error', n_jobs=-1)
grid_SVR.fit(X_train, y_train)
# print results
print(" Results from Grid Search " )
print("\n The best estimator across ALL searched params:\n", grid_SVR.best_estimator_)
print("\n The best score across ALL searched params:\n", -grid_SVR.best_score_)
print("\n The best parameters across ALL searched params:\n", grid_SVR.best_params_)
```

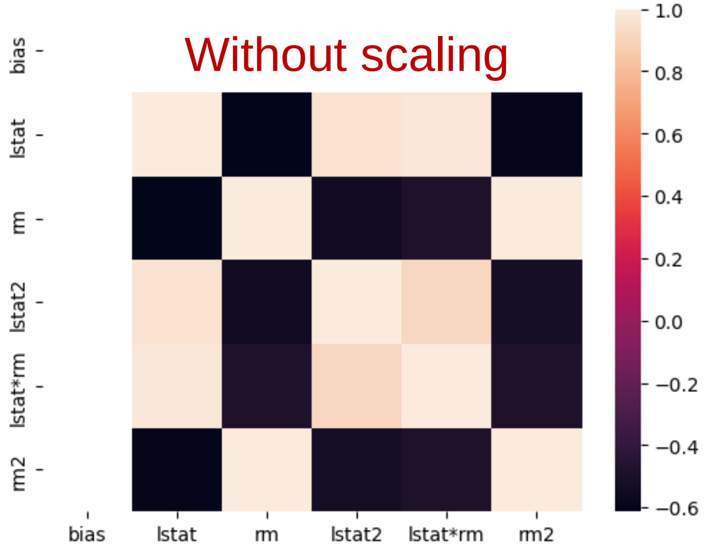
```
The best estimator across ALL searched params:
RandomForestRegressor(max_depth=10,
max_features='sqrt', min_samples_leaf=4,
                      n_estimators=200)

The best score across ALL searched params:
0.39623826398974954

The best parameters across ALL searched params:
{'bootstrap': True, 'max_depth': 10,
'max_features': 'sqrt', 'min_samples_leaf': 4,
'min_samples_split': 2, 'n_estimators': 200}
```

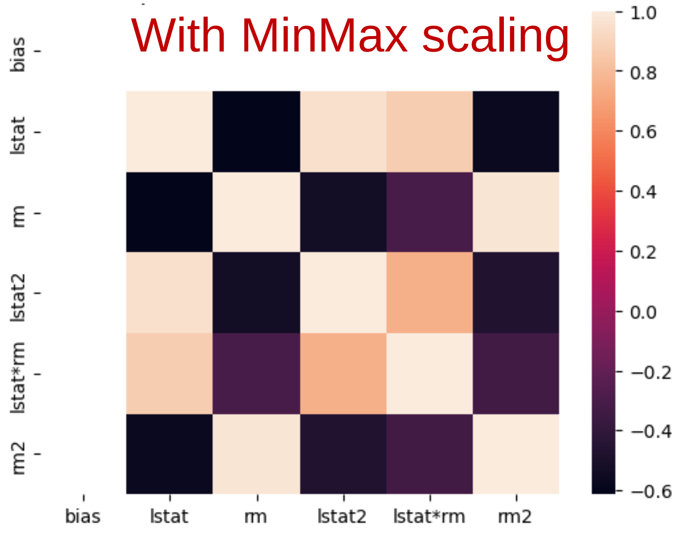
Slightly better results than SVR model but still slightly worse than the polynomial model.

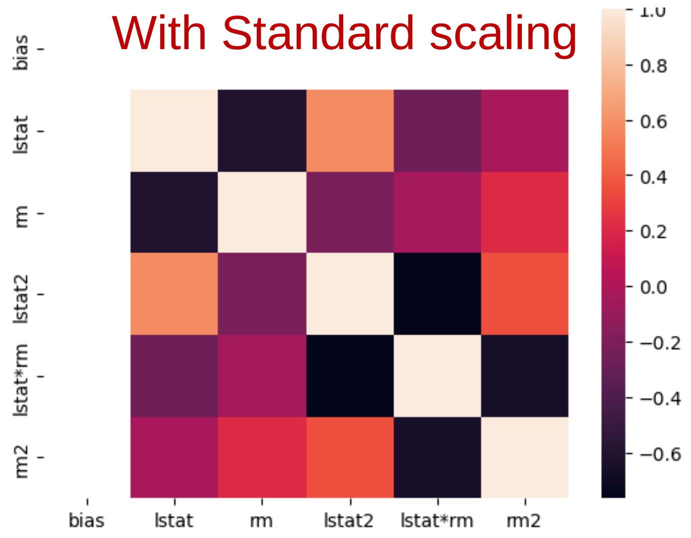
Warning: This may run several minutes!!

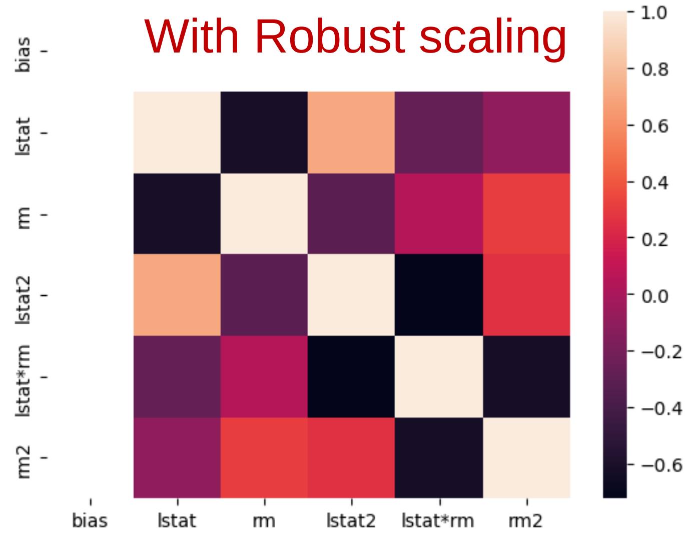
Appendix: Scaling vs correlation

- Correlation among original features, power and interaction terms







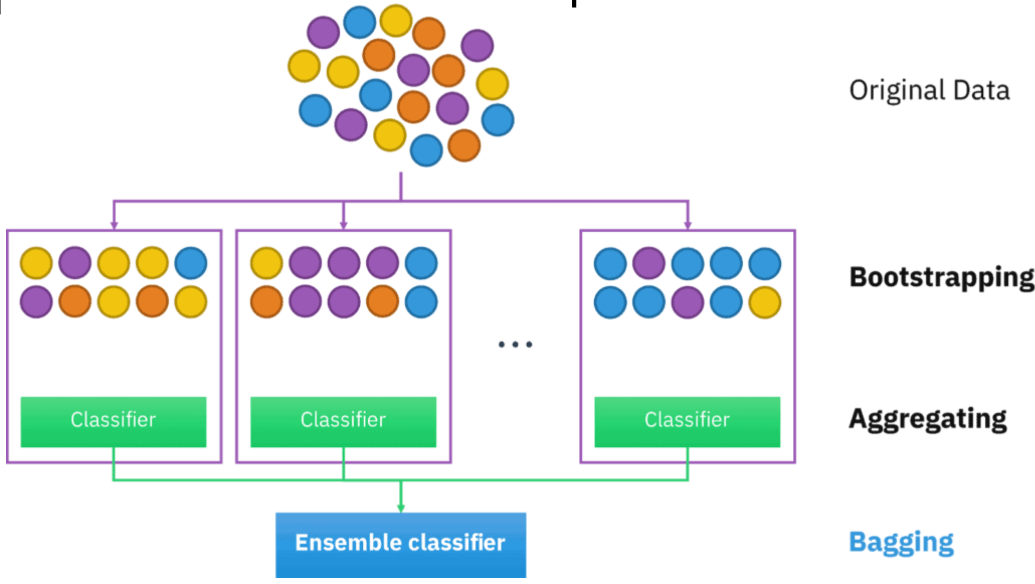


- There is minimal correlation when centering-based scalers (Standard, Robust) are applied
- Source code is found here

Appendix: Bagging in detail

- Parallel Ensemble Learning of homogeneous learners: Bootstrapping (resampling) => Aggregating => Bagging
  1. To start with, let's assume you have some original data that you want to use as your training set (dataset D with N samples). You want to have K base models in our ensemble.
  2. In order to promote model variance, Bagging requires training each model in the ensemble on a randomly drawn subset of the training set. The number of samples in each subset is usually equal to the original dataset (N), although it can be smaller.
  3. To create each subset, you need to use a bootstrapping technique:
     a) First, randomly pull a sample from your original dataset D and put it to your subset
     b) Second, return the sample to D (this technique is called sampling with replacement)
     c) Third, perform steps (a) and (b) N (or less) times to fill your subset
     d) Then perform steps (a), (b), and (c) K – 1 time to have K subsets for each of your K base models
  4. Train each of K base models on its subset, make predictions using test (unseen) dataset
  5. Combine (aggregate) the prediction of each sample (row) from the test dataset and evaluate the final result for each sample

Training dataset: D
N samples



If you are solving a Classification problem, you should use a voting process to determine the final result. The result is usually the most frequent class among K model predictions. In the case of Regression, you should just take the average of the K model predictions.

Appendix: Bagging in detail (sampling with replacement)

Training datasets (with 10 samples/rows each)

Original dataset	1	2	3	4	5	6	7	8	9	10
Base Model 1 dataset	7	8	10	8	2	5	10	10	5	9
Base Model 2 dataset	1	4	9	1	2	3	2	7	3	2
Base Model 3 dataset	1	8	5	10	5	5	9	6	3	7

- Boostrapping process creates a new training dataset for each base model
- Some samples (rows) of the initial training dataset can be selected multiple times within a base model’s training dataset
- Build multiple base models – each one trained on its own dataset
- Use each base model to make a prediction using the test dataset
- Combine (average) predictions to provide the final ensemble algorithm prediction