DSC510: Introduction to Data Science and Analytics
Lab 6: Regression

University of Cyprus
Department of
Computer Science

Pavlos Antoniou

Office: B109, FST01

Recap from last lab

- In the last two labs, we focused on data preparation.
- We applied several preprocessing techniques and tools to clean, encode and transform the data.
- In the most recent lab, we explored feature selection and feature extraction techniques to identify or create the most informative / important features.
- Once data preparation is complete, the processed features can be used to train machine learning models to uncover insights or make predictions.

Predictive modeling techniques

- Predictive modeling techniques turn data into value
  - algorithms or mathematical frameworks that use historical data to forecast future outcomes, events, or behaviors

- 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

Predictive techniques: Supervised learning

- You have input features (X) and an output target 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**: the output variable (y) is a category, such as "disease" or "no disease", 0 or 1 (binary classification) and "red" or "blue" or "green" (multiclass classification). Values can be strings or discrete integers
    - Popular techniques: Logistic Regression, 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**: the output variable (y) is a continuous numerical value, such as "price" or "weight"
    - Popular techniques: Linear Regression, Polynomial Regression, Support Vector Regression (SVR), Random Forest Regression, XGBoost Regression, AdaBoost Regression

Predictive techniques: 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

Regression

- The process of estimating the relationship between a dependent (target) variable y which takes continuous numerical values and one or more independent (or input) variables X (features)
  - Example:
    - Goal: Predict house price (dependent variable)
    - Based on: house area in square meters (independent variable)
    - The house area is considered independent because we observe or measure it, but we cannot mathematically derive it from other variables.
    - However, we can predict the house price using the house area.
- Common regression algorithms:
  - Linear Regression – finds best fitting straight line (first-degree equation)
  - Polynomial Regression – fits higher-degree polynomial curves (2^{nd}, 3^{rd}, …)
  - Support Vector Regression (SVR)
  - Ensemble methods: Random Forest, Ada Boost, XGBoost

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):

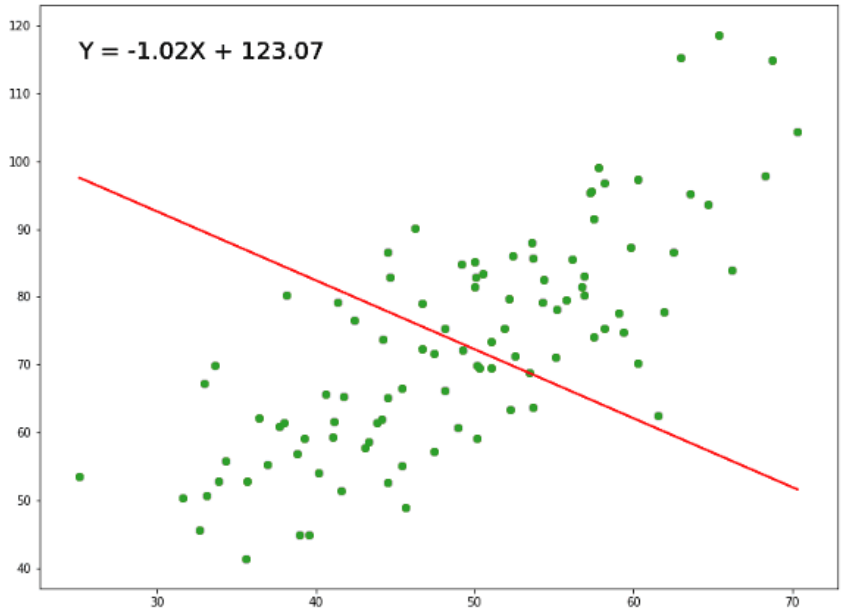
$$y = \beta_0 + \sum_{j=1}^{p} \beta_j * X_j = \beta_0 + \beta_1 * \mathrm{X}_1 + \beta_2 * \mathrm{X}_2 + \cdots$$

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

Linear Regression Methods

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

2. **Gradient descent** finds the linear model β coefficients iteratively



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

Main assumptions for using Linear Regression

- Linear relationships
  - between each independent variable (feature) and the dependent variable (target)
    - can best be tested with scatter plots / pair plots
- No or little multicollinearity
  - Low correlation between two or more independent variables – 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
    - PCA can also be used to reduce multicollinearity new features are uncorrelated
- 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: 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)

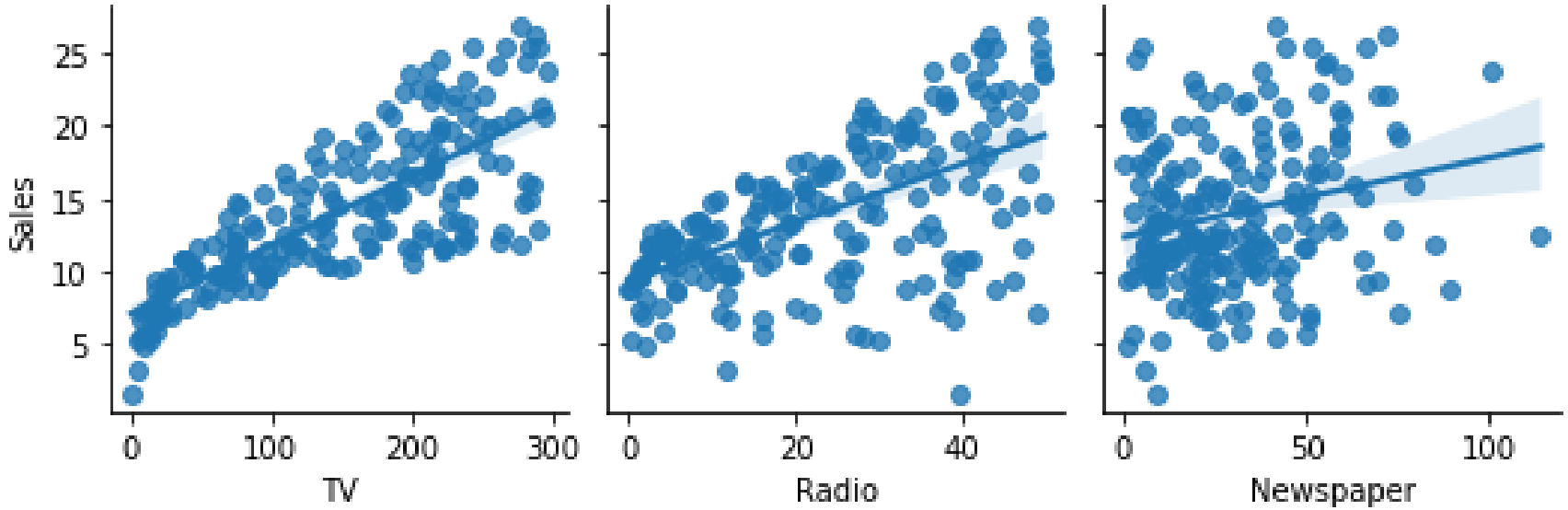
target variable

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.
Dataset is clean, without missing and erroneous values

Linear Regression: Testing assumptions

- Linearity

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

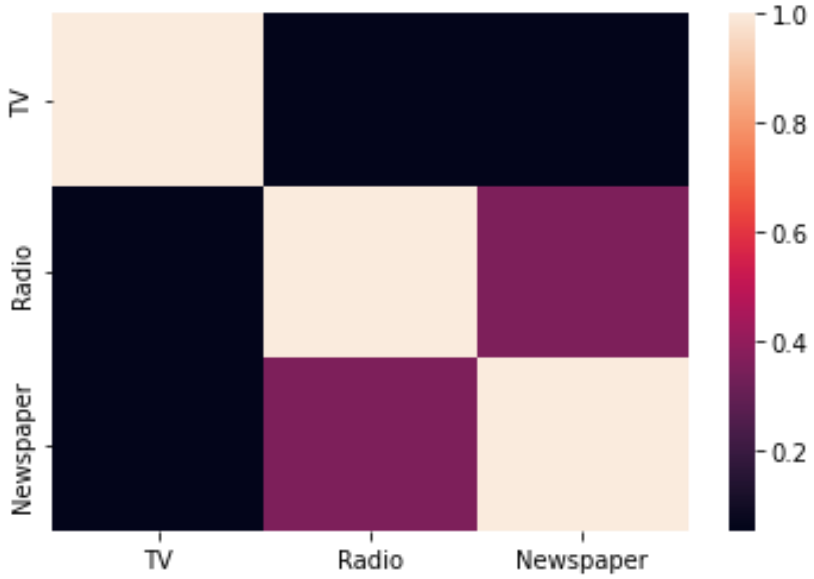


By looking at the plots we can see that none of the independent variables has an accurately linear relationship with Sales but TV and Radio do still better than Newspaper which seems to hardly have any specific shape. So, it shows that a linear regression fitting might not be the best model for it. A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

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

```
# extract the features (independent variables)
X = df.drop(columns=['Unnamed: 0','Sales'])
print(X[0:10])

# extract the dependent (target) variable
y = df['Sales']
print(y[0:10])
```

```
      TV  Radio  Newspaper
0  230.1   37.8       69.2
1   44.5   39.3       45.1
2   17.2   45.9       69.3
3  151.5   41.3       58.5
4  180.8   10.8       58.4
5    8.7   48.9       75.0
6   57.5   32.8       23.5
7  120.2   19.6       11.6
8    8.6    2.1        1.0
9  199.8    2.6       21.2
```

```
0    22.1
1    10.4
2     9.3
3    18.5
4    12.9
5     7.2
6    11.8
7    13.2
8     4.8
9    10.6
Name: Sales, dtype: float64
```

Linear Regression: Linear Regressors

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

# ALTERNATIVE REGRESSOR
from sklearn.linear_model import SGDRegressor
sgdr = SGDRegressor()
```

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

SGDRegression object uses stochastic gradient descent method

- SGDRegressor uses the iterative method gradient descent to estimate the coefficients
- The main reason why gradient descent could be preferred for linear regression instead of the 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 or samples (observations).

Non standardized independent values

Linear Regression: Model training

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

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

Training data size: 80%
Testing data size: 20%

```
# 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]
```

Model after training: $y = 2.9948 + 0.04458*x_1 + 0.19649*x_2 - 0.00278*x_3$

The coefficient 0.04458 for the first feature (TV ads) means that for each additional 1 thousand dollars spent, the sales increase by 0.04458 thousands units, holding the rest features constant.

Non standardized independent values

Linear Regression: Making prediction

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

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.80)


# 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_test)
# residuals is the differences between real y values (y_test) and predicted y values
residuals = y_test - y_pred
print("Residuals:", residuals[:10])
```

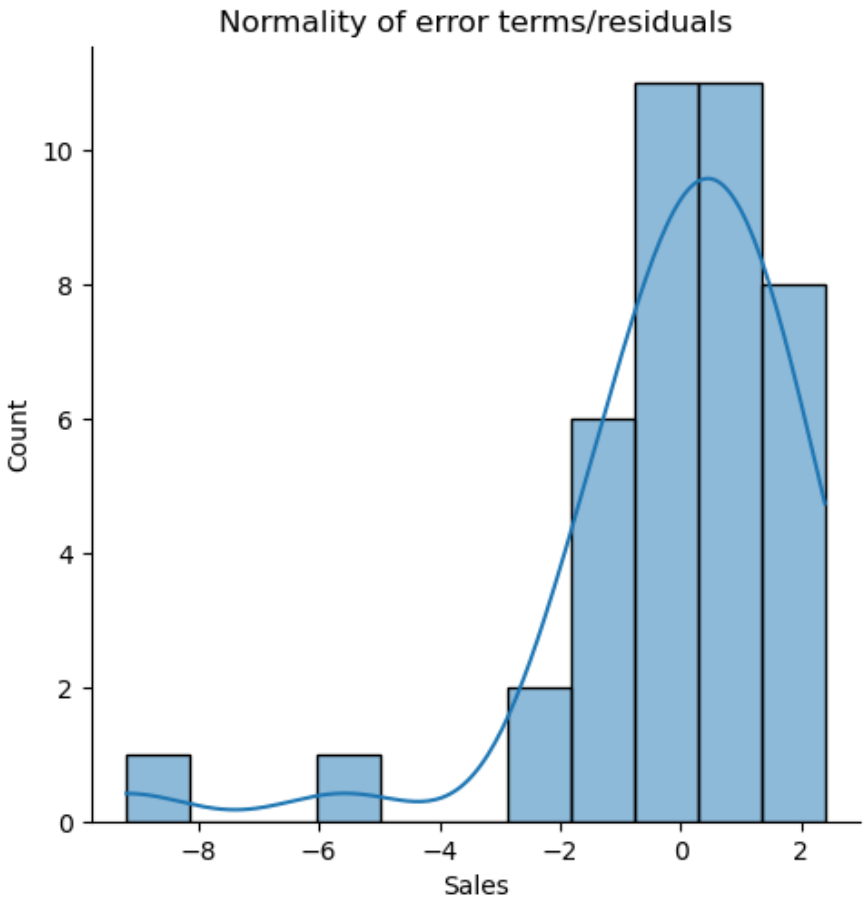
```
Residuals: [0.947719, 1.680292, 1.319703, -0.31786
3, 2.162061, -5.582869, -1.909746, -1.569740, 0.25
6420]
```

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

```
# computing the p-value for the null-hypothesis
that this distribution is a normal distribution
from scipy import stats
_, p = stats.normaltest(residuals)
# p-value of 0.05 or greater means that the
distribution is a normal distribution
print(p) # => 3.602868648223426e-09, residuals
deviate from normal distribution
```



Data scaling/standardization

- The values of β coefficients represent the influence of each input feature on the target variable: $y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 + \cdots \beta_n X_n + \in$
  - When regression is used for explaining a phenomenon, i.e. how input features influence the output y, the values of β coefficients can shed light
    - E.g. if β1 > β2 one might say X1 has higher impact than X2 on y since a small change in X1 results in a comparably large effect on y
  - BUT we cannot directly compare the size of the various β coefficients if the input variables are measured on different scales
- By scaling/standardizing variables, coefficients become directly comparable to one another, with the largest coefficient indicating which independent variable has the greatest influence on the dependent variable
  - We can scale input features using MaxMinScaler, StandardScaler, RobustScaler shown in Lab 4

When is feature scaling needed in LR?

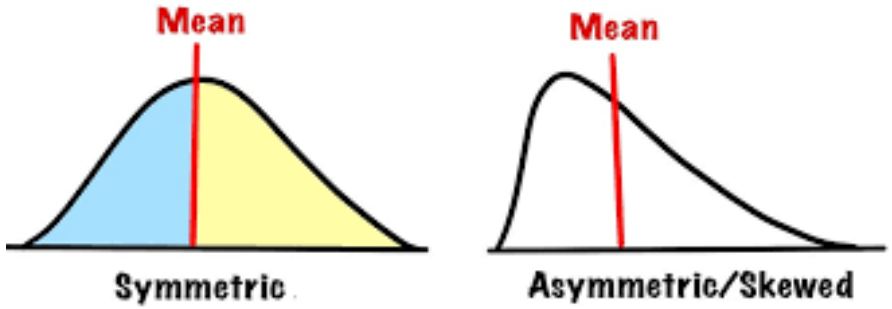
- When linear regression is used for making β coefficients directly comparable to one another and reveal the influence of each feature on target thus making it easy to present effects to non-statisticians

- Technically, feature scaling does not make a difference in linear regression, however, when gradient descent-based algorithms (such as SGDRegressor) are used for Linear Regression, feature scaling is needed to speed up the process of convergence (see more details here)

When is feature/target unskewing needed in LR?

- Unskewing transformations attempt to make long-tail distribution more symmetric, ideally resembling a Gaussian (bell-shaped) distribution
  - Unskewing transformations: BoxCox, Yeo-Johnson, Sqrt, Log

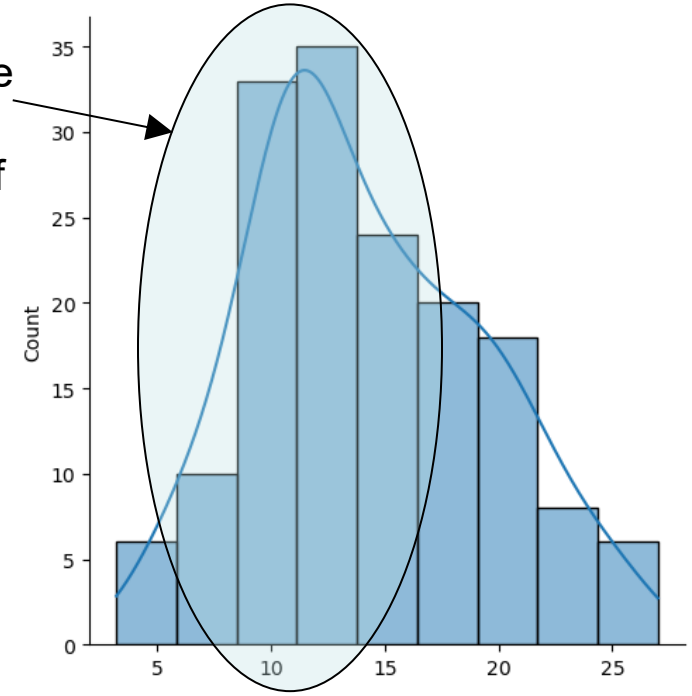
- Linear regression (OLS method) does not require feature and target variable distributions to be normal
  - But, in the presence of highly **skewed target variable**, the trained predictive model **is "biased" towards the majority of data**.



Linear Regression: Target variable distribution

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

We have more products with low number of sales




```
import seaborn as sns
# distribution plot of the target variable
sns.displot(y_train, 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_train)
# p-value of 0.05 or greater means that the distribution
is a normal distribution
print(p) # => 0.03975020925593686 4, not normal distrib.
```

- Solution: Unskew target variable using techniques of Lab4

Linear Regression: Standardize X / unskew y

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

# train scaler & apply transformation on training
X_train_scaled = sc.fit_transform(X_train)
print(X_train_scaled[0:10])
# apply scaler on test set
X_test_scaled = sc.transform(X_test)

# Unskew the target variable values
# Apply box-cox on training dataset
pt_bc = PowerTransformer(method='box-cox')
y_train_unskewed = pt_bc.fit_transform(y_train.to_frame()).ravel()
print(y_train_unskewed[0:10])
# apply transformation on y validation & test
y_test_unskewed = pt_bc.transform(y_test.to_frame()).ravel()
```

```
[[-1.34155345  1.0355176   1.65941078]
 [-1.4053143   0.08249594 -1.30629738]
 [-0.08995151  0.40243892 -0.81980897]
 [ 0.69761311 -0.18979597 -0.90868666]
 [ 0.76609699  0.01442296  1.28518893]
 [-0.56461564  0.42286082 -1.01627544]
 [-1.67570755 -1.44914602 -1.36243065]
 [-1.57770476  1.38268978  2.77272078]
 [-0.29304164  0.91979354  2.29558792]
 [-0.54218127 -1.20408331  0.19994556]]

[-0.6210696  -0.91685477  0.23555867  0
.40379751  0.62134625  0.04299781
 -2.19197316 -1.10834948  0.40379751 -0
.71021751]
```

Train the box-cox transformer on the training data set, then use the trained transformer to apply the transformation to the test data set to avoid data leakage.
.ravel() converts the resulting 2D array to 1D array which is required by ML predictive techniques.
The trained transformer will also used in reverse box-cox transformation.

Standardized independent values

Linear Regression

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

# train model (Fit linear model) and evaluate model β coefficients
model_scaled = lregr_scaled.fit(X_train_scaled, y_train_unskewed)
# 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_pred_unskewed = model_scaled.predict(X_val_scaled)
```

```
β0 = 2.56226937e-16
[β1,β2,β3] = [0.76046137  0.54112526 -0.00710177]
```

"TV", "Radio", "Newspaper"

- Standardization changes the interpretation of coefficients (at the same scale).
- 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.

Linear Regression: Model evaluation

- Model evaluation is a core part of building an effective ML 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:

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

minimize

  - MAE: Mean Absolute Error $MAE = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i|$
    - clear, interpretable sense of how far off, on average, predictions are from actual values
    - Does not penalize large prediction errors (caused by outliers)
  - MSE: Mean Squared Error $MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$
    - Average of squared differences: Large prediction errors are penalized (sensitive to outliers)
  - RMSE: Root Mean Squared Error $RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}$
    - error in the same units as the target variable, making it more interpretable than MSE

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)

Non standardized independent values

Linear Regression: Evaluate model

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

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

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

```
[10.05739563  7.4522807   7.0197076  24.08029725 12.01786259
6.5379385  12.78286918 15.10974587 10.76974013 16.34357951]
MSE: 4.402118291449682 , RMSE: 2.0981225634956795 ,
R2: 0.8601145185017869
```

86% of the variance in the target variable (Sales) is explained by the features in model.
86% fitting of the model on data points

Standardized independent values Unskewed target values

Linear Regression: Evaluate model

```
# prediction on validation data
# Model trained on standardized features and (box-cox) transformed target values
y_pred_unskewed = model_scaled.predict(X_test_scaled)
```

```
MSE_scaled = mean_squared_error(y_test_unskewed, y_pred_unskewed)
RMSE_scaled = np.sqrt(MSE_scaled)
r2 = r2_score(y_test_unskewed, y_pred_unskewed)
```

```
MSE: 0.2670809327659692 , RMSE: 0.5167987352596456 ,
R2: 0.7952680962371333
```

- Model performance in terms of R2 seems worse than without scaling but predicted values for Sales (y_pred_skewed) and test values for Sales (y_test_skewed) are box-cox transformed; not directly comparable with original values
- Revert to original scale using inverse box-cox and measure error

```
y_pred_inverse = pt_bc.inverse_transform(y_pred_unskewed.reshape(-1, 1))
```

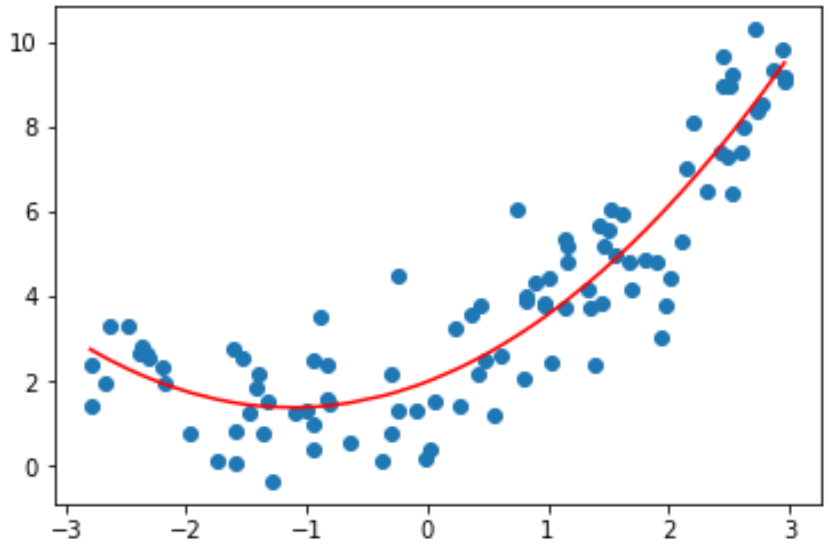
```
MSE_inverse = mean_squared_error(y_test, y_pred_inverse)
RMSE_inverse = np.sqrt(MSE_inverse)
r2 = r2_score(y_val, y_pred_inverse)
```

```
MSE: 3.51606112778633354 , RMSE: 1.8751162971363498 ,
R2: 0.8882706298935984
```

- R2 score is higher than before we had a model trained on non-transformed data – better performance with scaling and unskewing

Polynomial (or non-linear) regression

- When non-linear relationship (curve) is observed between independent (features) and dependent (target) variables
- Polynomial Regression comes to the play which predicts the best fit that follows the pattern (curve) of the data, as shown in the pic below:



Polynomial regression

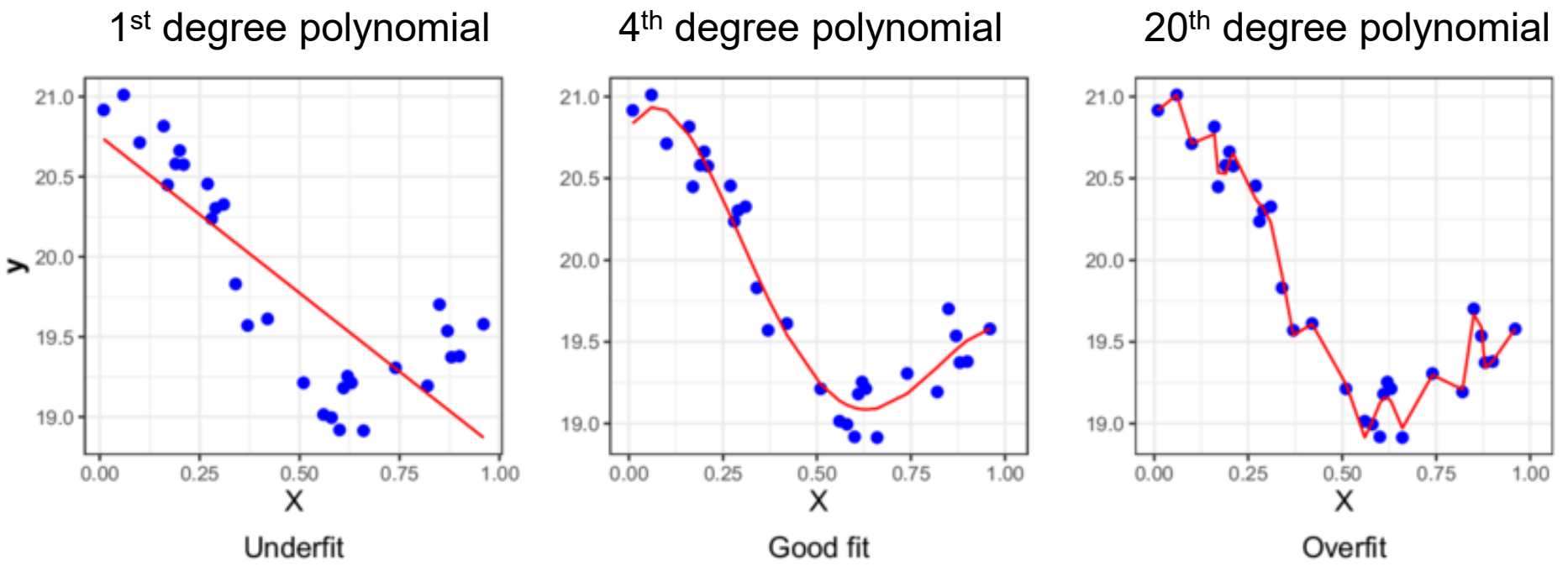
- Relationships between the independent variable(s) x and the dependent variable y are modelled as an n^{th} degree polynomial in x

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

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

Polynomial regression: of which degree?

- Increasing the degrees of the model also increases the risk of over-fitting the data (high accuracy on training data, not well generalization on unseen data)



Overfitting occurs when the model fits the "noise" in the data rather than the underlying trend.

- The degree of the polynomial to fit is a hyperparameter that cannot be inferred while fitting (training) model to the training set because it needs to be set in advance (before training)

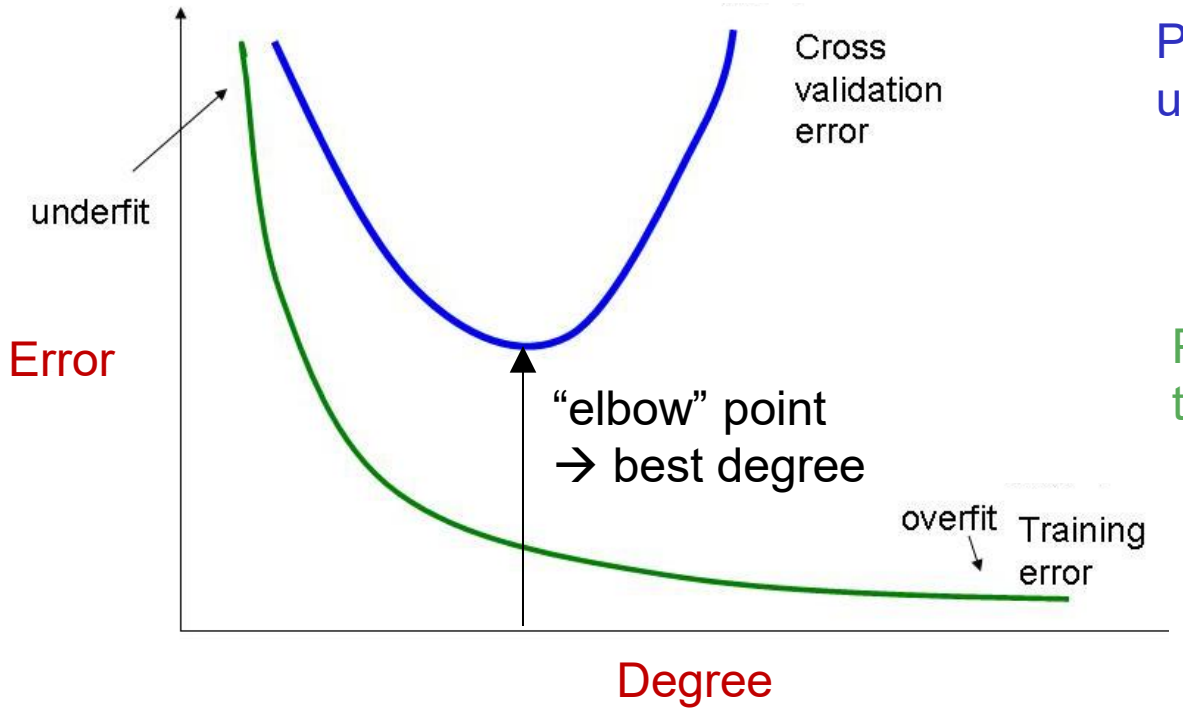
How to find the right degree of the equation?

- One approach to find the right degree while preventing over-fitting or under-fitting, is to start with a model of degree=1 and then gradually increase the model's degree until the performance (e.g. MSE, RMSE, R2) on unseen data stops getting better significantly → as increasing the degree beyond this point leads to overfitting
  - At each step:
    - **Train** the model using the ***training dataset***
    - **Predict** the target value using the ***validation dataset***
    - Evaluate model performance using any measure (e.g. MSE, RMSE, R2)

    It is strongly recommended to use Cross Validation (explained later)
  - At the end, when the best model is chosen, evaluate its **final performance** by predicting the target value using the **test dataset**.
- The whole process can be automated using GridSearchCV (see later)

How to find the right degree of the equation?

- One approach to find the right degree while preventing over-fitting or under-fitting, is to start with a model of degree=1 and then gradually increase the model's degree until the performance (e.g. MSE, RMSE, R2) on unseen data stops getting better significantly → as increasing the degree beyond this point leads to overfitting



Performance (error) on unseen data using CV

Performance (error) on training data

How is Polynomial regression implemented?

- 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. using 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 β_0, β_1, β_2, β_3
- When having more than one features, interaction terms appear
  - 2nd degree polynomial model : $y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1^2 + \beta_4 X_1 X_2 + \beta_5 X_2^2$ (Interaction term)
    - 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 β_0, β_1, β_2, β_3, β_4, β_5

When is scaling/unskewing needed in PR?

- Feature scaling is often needed in polynomial regression, and it can significantly impact the stability of the model:
  - Scaling facilitates faster convergence when gradient descent algorithms (e.g. SGDRegressor) will be used
  - Scaling makes beta coefficients comparable: When features are scaled, the coefficients reflect the relative importance more accurately and the influence of each feature on target variable, making it easier to interpret the model
  - Scaling removes multicollinearity (high correlation among features) leading to stable coefficient estimates ➔ model insensitive to small changes in data

Polynomial Regression: Boston Housing Dataset

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

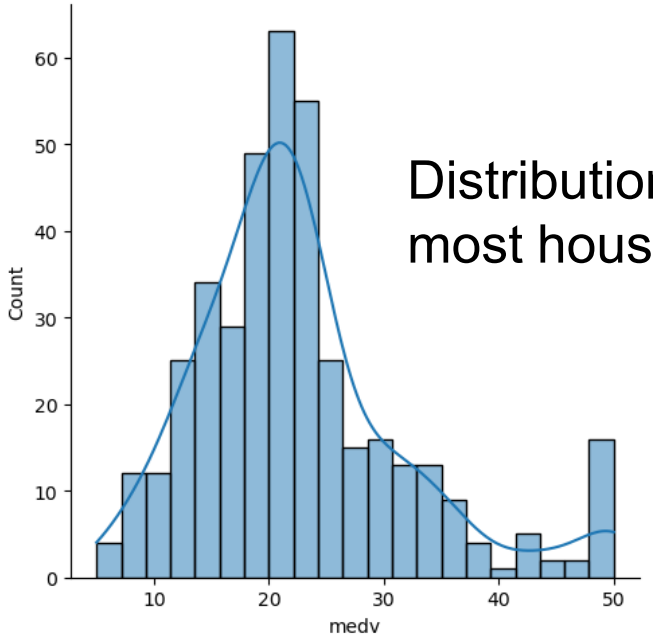
```
import numpy as np
import matplotlib.pyplot as plt

import pandas as pd
import seaborn as sns

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

# extract features and target variables
X = boston.drop(columns=['medv'])
y = boston['medv']

# split to training and test datasets (80% / 20%)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = 5, train_size = 0.8)
```

	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

Data transformation

- Feature scaling does not improve the predictive power of the model when using linear regressors; however useful when performing polynomial regression for the stability of the model
- Target variable transformations (such as Box Cox, Yeo-Johnson when skewness is apparent) can improve the model predictive power

```
# distribution of the target values
sns.displot(y_train, 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(y_train)
print(p) # => 1.52 e-16
```

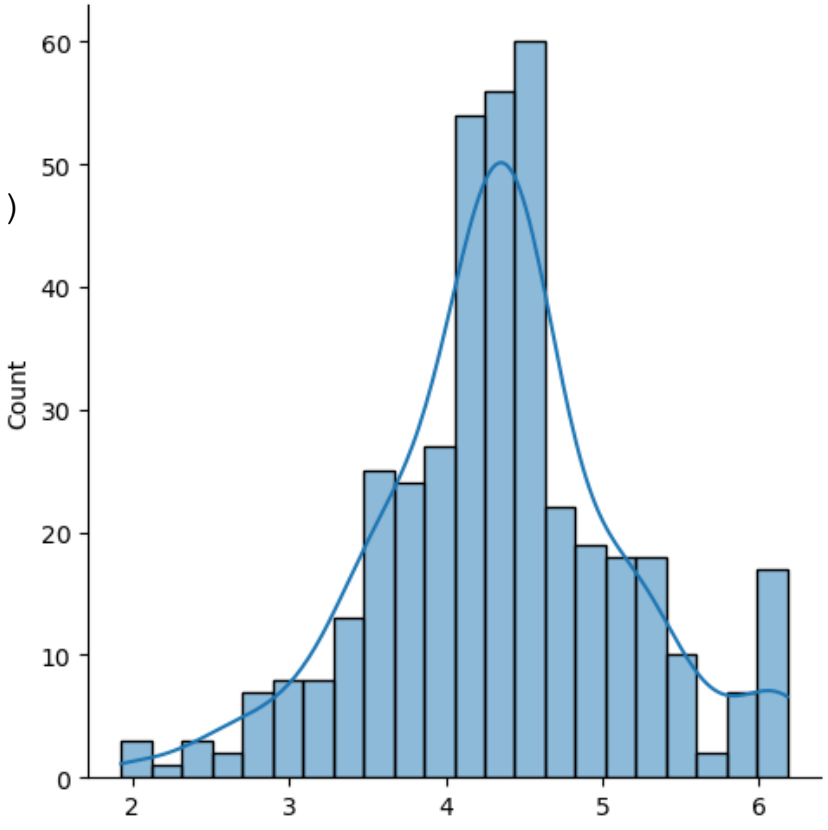


Distribution is skewed (not symmetrical): most houses have low prices

Data transformation

- Here, we use boxcox transformation

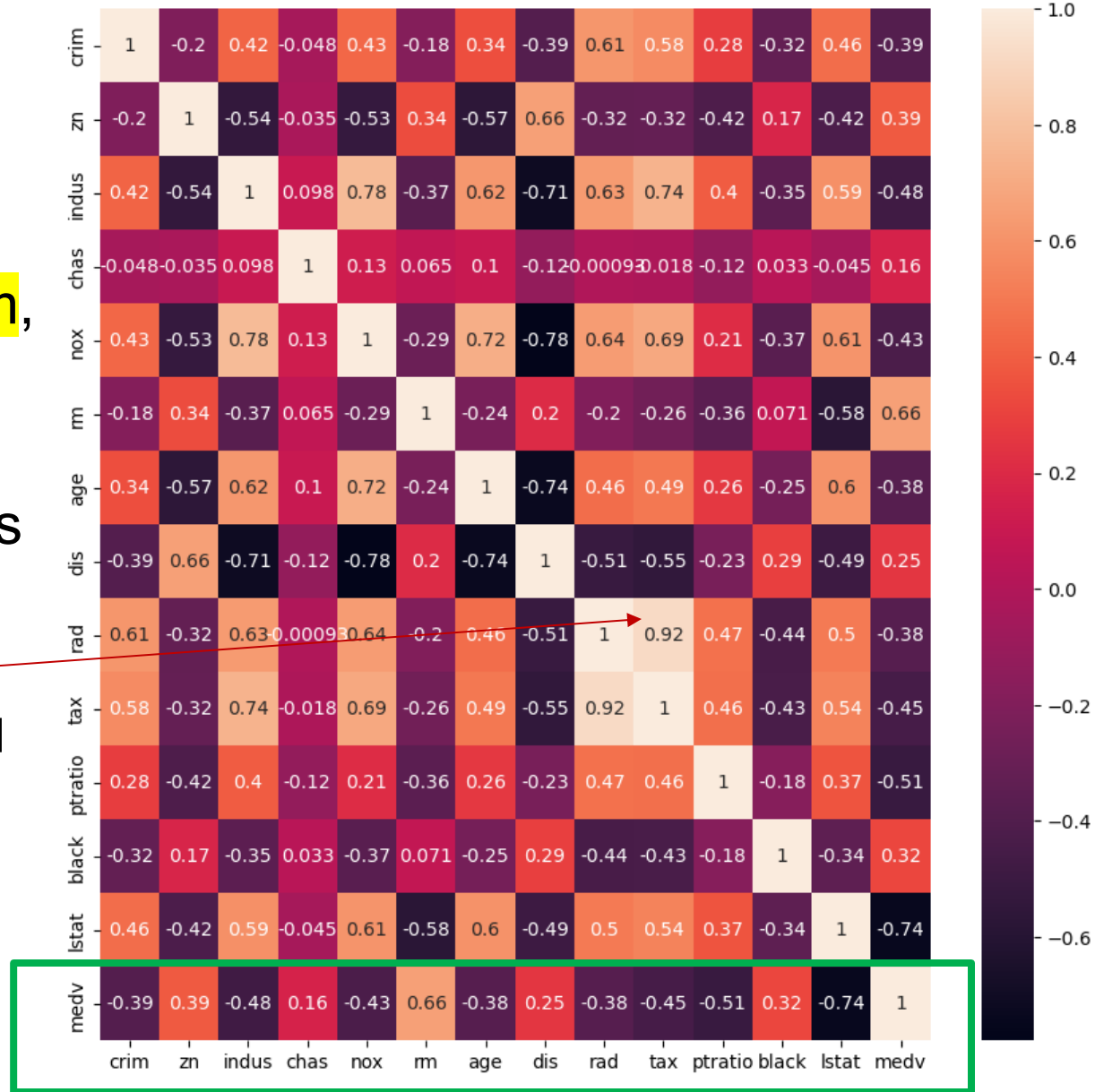
```
# y - transformation (box cox)
pt_bc = PowerTransformer(method='box-cox')
y_train_bc = pt_bc.fit_transform(y_train.to_frame()).ravel()
_, p = stats.normaltest(y_train_bc)
print(p) # => 0.13691573
sns.displot(y_train_bc, kde=True)
```



- Distribution of the transformed target variable
  - This distribution already looks quite similar to a normal distribution and achieves a p-value of 0.13, which is larger than 0.05. Therefore, we can say that the distribution approaches a normal distribution

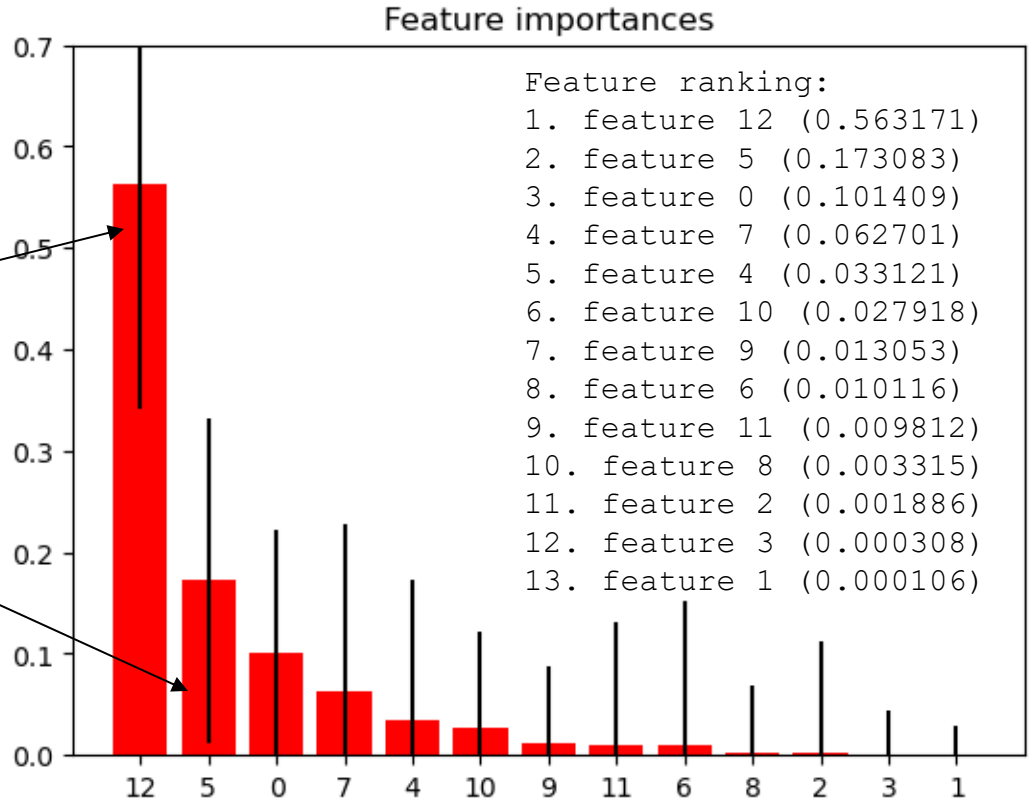
Feature Selection – Correlation matrix

- Create correlation matrix on the training dataset
- 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



Feature Selection – Importance

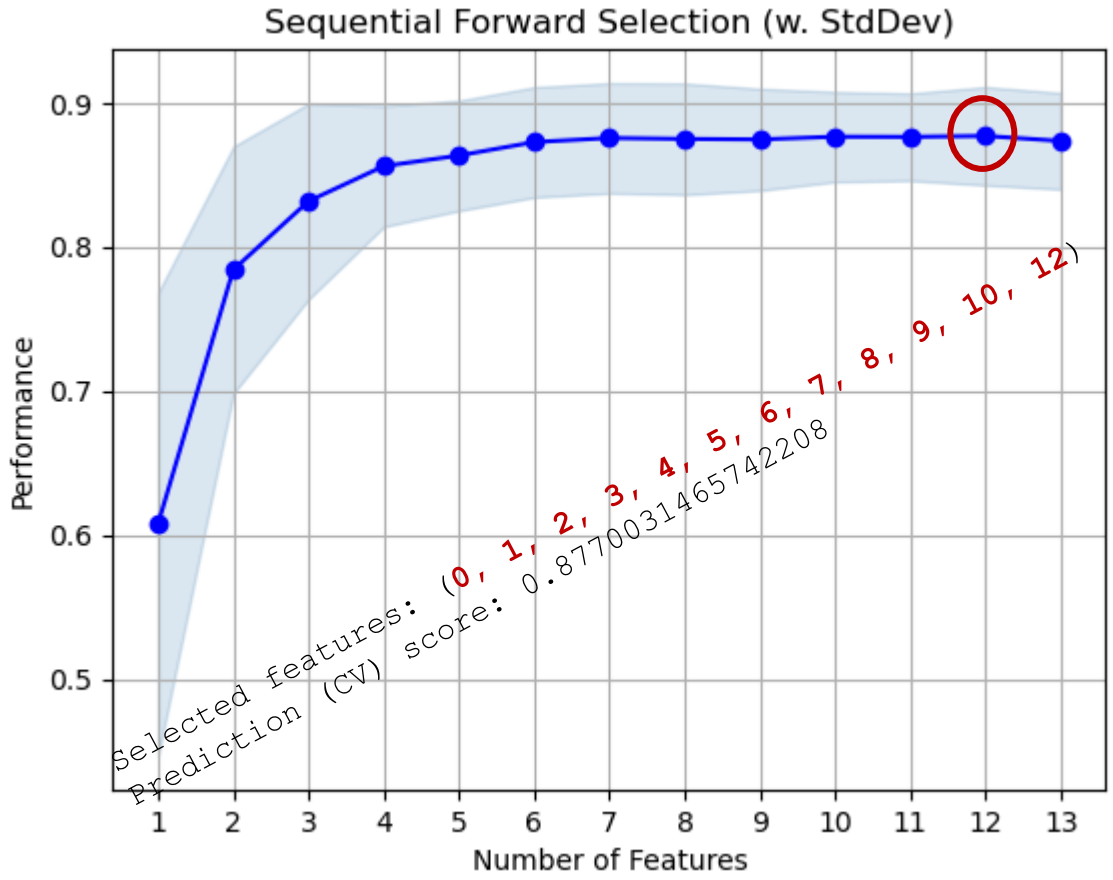
```
# Feature Importance using ExtraTreeClassifier
from sklearn.ensemble import GradientBoostingRegressor
# Build an estimator and compute the feature importances
estimator = GradientBoostingRegressor(n_estimators=100, random_state=0)

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



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

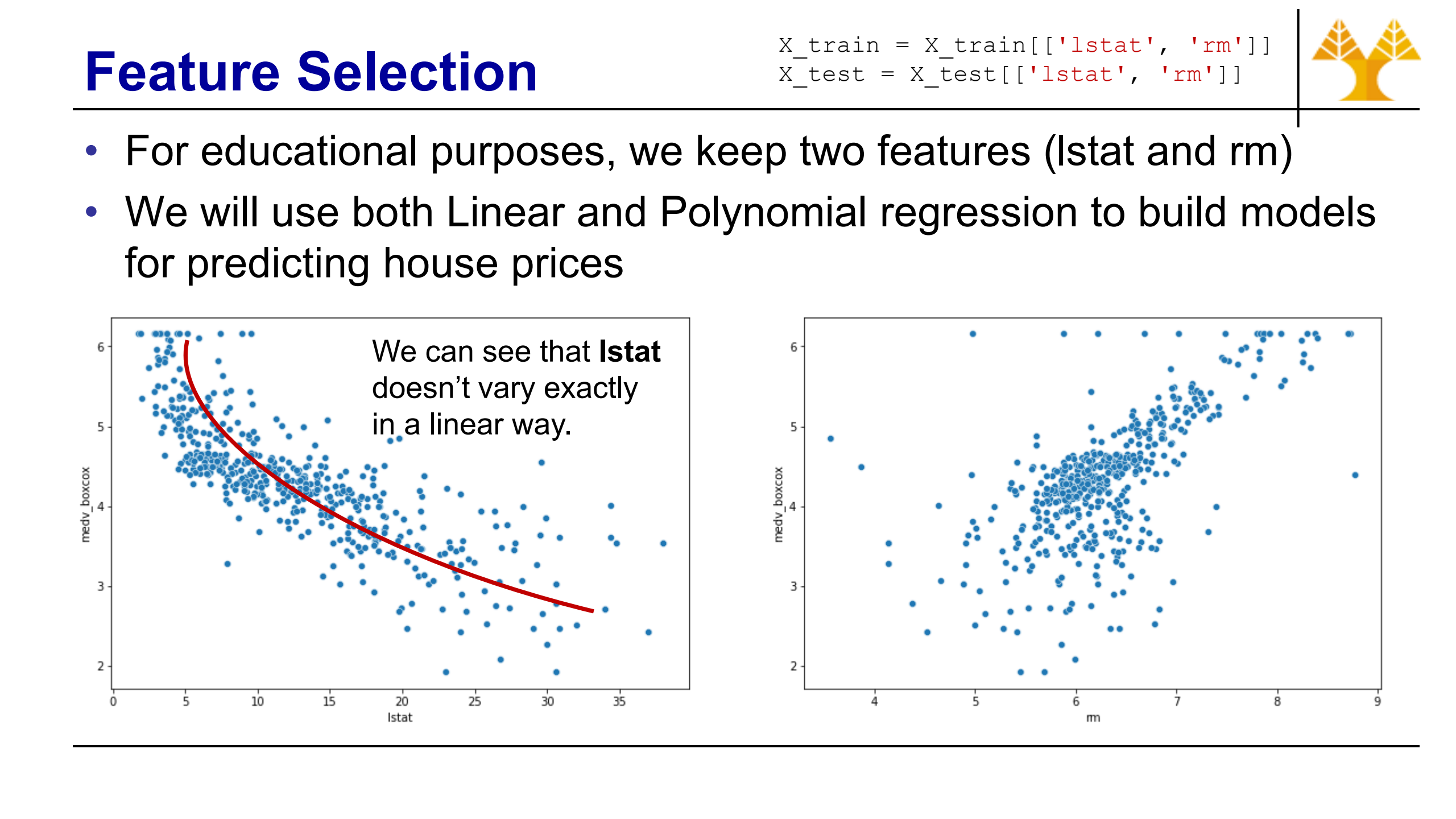
Feature Selection – Sequential Forward Selec

```
from mlxtend.feature_selection import SequentialFeatureSelector as SFS
from mlxtend.plotting import plot_sequential_feature_selection as plot_sfs

sfs = SFS(estimator,
          k_features=(2,13),
          forward=True,
          floating=False,
          scoring='r2',
          cv=10)

sfs = sfs.fit(X_train, y_train_bc)

plot_sfs(sfs.get_metric_dict(), kind='std_dev')

plt.title('Sequential Forward Selection')
plt.grid()
plt.show()
print('Selected features:',sfs.k_feature_idx_)
print('Prediction (CV) score:',sfs.k_score_)
```



Feature Selection

```
X_train = X_train[['lstat', 'rm']]
X_test = X_test[['lstat', 'rm']]
```



- For educational purposes, we keep two features (lstat and rm)
- We will use both Linear and Polynomial regression to build models for predicting house prices

Linear Regression on Boston dataset

- Results using the initial dataset without transformations (feature scaling, target unskewing)

```
Model performance on test dataset
----------------------------------
RMSE is 5.137400784702911
R2 score is 0.6628996975186953
```

- Results (on original scale) using Box Cox transformation on target variable

```
Model performance on test dataset
----------------------------------
RMSE is 4.603290903028745
R2 score is 0.7293493412884403
```

- Better performance is experienced when unskewing transformation is applied on the target variable

Linear Regression (playing 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 value of Y when all X=0
  - Default value of `fit_intercept` is true, i.e. β_0 is part of the model
- Set `lr = LinearRegression(`**`fit_intercept=False`**`)` and follow the process (training, prediction on test dataset, model evaluation) using the transformed target variable
  - Slight deterioration of the model

```
Model performance on test dataset (without intercept term)
------------------------------------------------------------
RMSE is 4.934031693652707
R2 score is 0.6890603420569186
```

Polynomial Regression (degree = 2)

```
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_bc)
```

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	lstat²	lstat * rm	rm²
[[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.

```
# transform test set features to higher degree features
X_val_poly = poly_features.fit_transform(X_test)

# predicting on test dataset
y_test_bc_predict = poly_model.predict(X_test_poly)

# revert to original scale
y_test_predict_orig = pt_bc.inverse_transform(y_test_bc_predict.reshape(-1, 1))
```

Polynomial Regression (degree = 2)

```
# evaluating the model on validation dataset
rmse_test_orig = np.sqrt(mean_squared_error(y_test, y_test_predict_orig))
r2_test_orig = r2_score(y_test, y_test_predict_orig)

print("Model performance on test dataset (original scale")
print("----------------------------------------")
print("RMSE is {}".format(rmse_test_orig))
print("R2 score is {}".format(r2_test_orig))
```

```
The model performance for the test set
----------------------------------------
RMSE of training set is 3.863684659842964
R2 score of training set is 0.8093329733509804
```

We can observe that the RMSE error is lower and R2 is higher (thus better) when using polynomial regression as compared to linear regression. However, hyperparameter tuning needs to be performed to:

- **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**

Problem with dataset splitting

- Results shown thus far (in terms of RMSE, R2) depend on a particular choice (split) for train and test datasets to train and evaluate the model
  - Based on the model's performance on unknown/unseen data for a single split, we cannot determine if it is underfitting, overfitting, or "well-generalized"

- Solution: Repeat the process of randomly splitting data into subsets (training and validation) and average results from all iterations => **Cross Validation (CV)**

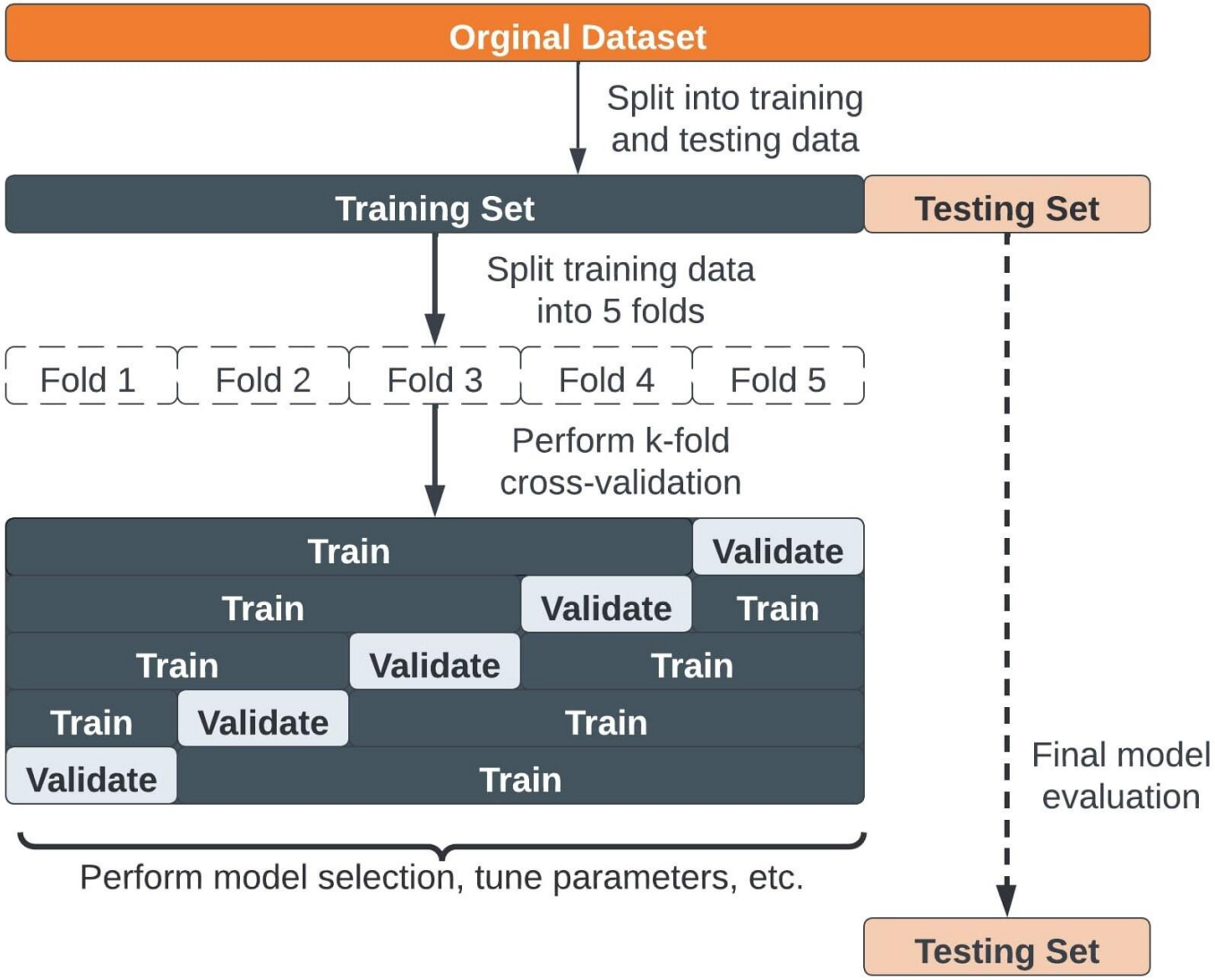
K-folds Cross Validation

- Prior running Cross-Validation, split initial dataset into train/test
- Further 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

Cross validation

- Modify model hyperparameters and re-run cross validation to find the best hyperparameter values
- Test dataset is used for the unbiased final evaluation of the model with the best model parameters and hyperparameters

k-folds Cross Validation



GridSearchCV

- Cross validation (CV) process creates a series of train and validation splits to train and measure the predictive power of the model
- During training (within CV process), best values for **model parameters are determined**
- Model **hyper-parameters cannot be directly learnt from the training phase**; thus, they need to be set before the CV process
  - When modifying a hyper-parameter, full CV process needs to be repeated
  - When multiple hyper-parameters are involved in a model, finding the best combination of hyper-parameter values is a hard job
- **Data encoding, transformation should be performed right after dataset splitting, within the CV process to avoid data leakage**
- Best strategy to implement all these steps: **GridSearchCV**

Exhaustive param search: GridSearchCV

- **GridSearchCV**: Exhaustive search over a specified hyper parameter combination for an estimator (classifier / regressor)
- Grid of hyper-parameter values is specified with the param_grid list
  - For example, for Polynomial Features estimator 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 will 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
- Evaluates model for each combination using CV for a scoring metric

```
grid = GridSearchCV(estimator, param_grid, cv=10, scoring = 'r2', n_jobs=-1)
grid.fit(X_train, y_train)
```

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

Pipeline

- Recall that polynomial regression process involves 2 sequential steps:
  - Create polynomial features
  - Apply linear regression

  } 2-step estimator

- It is possible to create a pipeline combining these two steps (PolynomialFeatures and LinearRegression)

- The pipeline is used as estimator in GridSearchCV

Polynomial regression: Pipeline with GridSearchCV

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

# split dataset to train/test 80% / 20%
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = 5, train_size = 0.8)

# 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 = 'r2', cv=10, n_jobs=-1)
search.fit(X_train, y_train)
```

Two-step pipeline (create polynomial features + apply regression) is the estimator.
We name each step ("poly" and "linear") so as to refer to its hyper-parameters, e.g. linear__fit_intercept is the fit_intercept thyperparameter of the linear regressor

The sklearn scoring API always maximizes the score, so metrics which need to be minimized like RMSE are negated (`"neg_root_mean_squared_error"`)

Polynomial regression: Pipeline with GridSearchCV

```
# print results
print(" Results from Grid Search " )
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_)

# Evaluate on the test set
best_model = search.best_estimator_
y_pred = best_model.predict(X_test)

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

# r-squared score of the model
r2 = r2_score(y_test, y_pred)

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

```
 Results from Grid Search

 The best score across ALL searched params:
 0.8239040045809777

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

Model performance on testing dataset
------------------------------------
RMSE is 3.220157338361434
R2 score is 0.8675577863835
```

Best performance achieved with Polynomial regression thus far

Pipelines

- A pipeline accepts a list of estimators, not only predictors but also data imputers, encoders, transformers to be applied prior training a predictor

```
im = SimpleImputer(strategy="mean")      # fill missing values
sc = StandardScaler()                    # scale features
preprocessing_pipeline = Pipeline([("imputer", im), ("scaler", sc)])
```

```
pf = PolynomialFeatures()                # create polynomial features
lr = LinearRegression()                  # linear regressor
training_pipeline = Pipeline([("poly", pf), ("linear", lr)])
```

```
# Pipelines can be attached to one another!
full_pipeline = Pipeline([("preprocessing", preprocessing_pipeline),
("training", training_pipeline)])
```

```
param_grid = [
    { "training__poly__degree": [1, 2, 3, 4, 5], "training__poly__interaction_only": [True, False],
"training__poly__include_bias": [True, False] },
    { "training__poly__degree": [1, 2, 3, 4], "training__poly__interaction_only": [True, False], "training__poly__include_bias":
[True, False], "training__linear__fit_intercept": [True, False] }
]
```

Pipelines with ColumnTransformer

- By default, transformations are applied to all columns of the dataset
- We can apply different transformations per column using `ColumnTransformer`. Example for applying different imputation strategy:
  - For int-based features (e.g. `chas` & `rad`) we will apply most_frequent imputation strategy
  - For `rm` and `age` we will apply mean imputation strategy followed by standard scaling
  - For the remainder features do nothing

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer
pipeline1 = Pipeline([('freq_imputer', SimpleImputer(strategy='most_frequent'))])
pipeline2 = Pipeline([('mean_imputer', SimpleImputer(strategy='mean')), ('scaler',
StandardScaler())])
preprocessing_pipeline = ColumnTransformer(transformers = [
    ('pipeline1', pipeline1, ['chas', 'rad']),
    ('pipeline2', pipeline2, ['rm', 'age']),
    # set remainder to passthrough to pass along all
    # the un-specified columns untouched to the next steps
], remainder='passthrough')
```

Pipelines with TranformedTargetRegressor

- Imputers, encoders and transformations are applied on input features
- Transformations (e.g. boxcox) on target variable can be applied using `TranformedTargetRegressor`

Pipeline **without** target transformation

```
training_pipeline = Pipeline([
    ("poly", PolynomialFeatures()),
    ("linear", LinearRegression())
])
```

Pipeline **with** target transformation

```
training_pipeline = Pipeline([
    ('poly', PolynomialFeatures()),
    ('linear', TransformedTargetRegressor(
        regressor=LinearRegression(),
        transformer=PowerTransformer(method='yeo-johnson')
    ))
])
```

- TransformedTargetRegressor is a meta-estimator that performs regression on a transformed target variable
- **Regressor** and **Transformer** are given as input
- PowerTransformer can be used to apply either boxcox or yeo-johnson transformations.

'yeo-johnson' → works with positive and negative values

'boxcox' → works with strictly positive values

Pipelines with TranformedTargetRegressor

- `TranformedTargetRegressor` with log transformation

Pipeline **without** target transformation

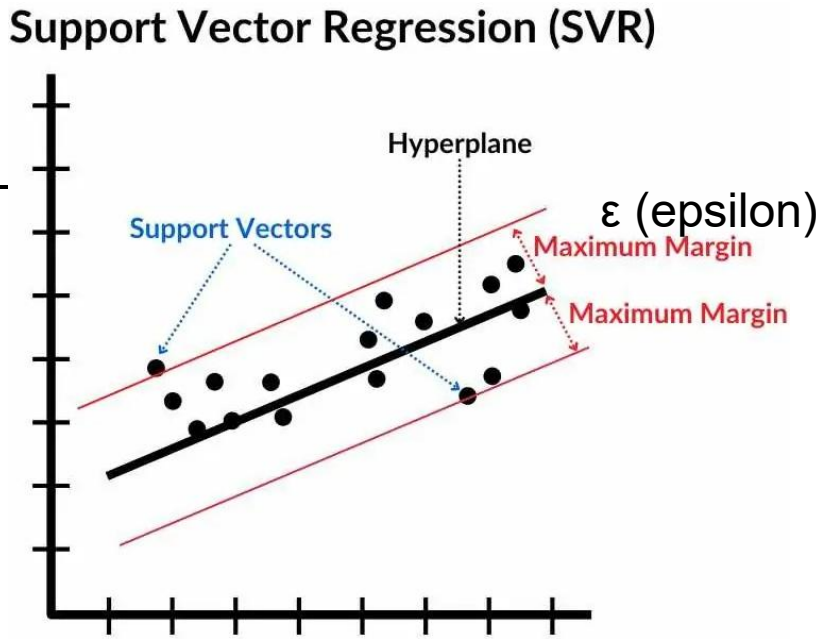
```
training_pipeline = Pipeline([
    ("poly", PolynomialFeatures()),
    ("linear", LinearRegression())
])
```

Pipeline **with** target transformation

```
training_pipeline = Pipeline([
    ('poly', PolynomialFeatures()),
    ('linear', TransformedTargetRegressor(
        regressor=LinearRegression(),
        func=np.log, inverse_func=np.exp
    ))
])
```

We can apply a transformer by setting the **func** parameter (transformation function) and **inverse_func** (reverse transformation function) instead of using the transformer parameter (see previous slide)

Support Vector Regression (SVR)



- Basic idea of support vector regression
  - Find optimal hyperplane that approximates the relationship between input features (X) and the target variable (y).
- **Hyperplane**: In the context of SVR, the hyperplane is a function (a line in 2D, or a plane in higher dimensions) that predicts the continuous output for a given set of input features
- **Epsilon (ε) Margin**: SVR introduces a margin of tolerance (ε) around the hyperplane.
  - Data points within this margin are considered "correctly predicted“ (no penalty is given to errors)
  - Data points that fall outside the margin or are on its boundary (called **support vectors**) contribute to the error term, and the goal is to minimize these errors

Support Vector Regression

- The objective of SVR is to minimize the following two key things:
  - **Prediction Error**: For points outside the epsilon margin, the error is proportional to how far the points deviate from the margin. The goal is to **minimize this error for points outside the margin**.
  - **Model Complexity**: SVR also tries to minimize the norm of the weights (w), which controls the flatness of the hyperplane. A **flatter hyperplane** is preferable because it helps generalize better (low error on unseen data).

Support Vector Regression hyperparameters

- **Epsilon**: defines a margin of tolerance around the hyperplane
  - Low epsilon → overfitting, High epsilon → underfitting | default value: 0.1
- **C** is a penalty parameter that controls the trade-off between the complexity of the model (flatness of the hyperplane) and the amount of allowed deviations (points falling outside the epsilon margin)
  - Low C → underfitting, High C → overfitting | default value: 1.0
- **Kernel**: a mathematical function that transforms data into a higher dimensional space (generally used for finding a better hyperplane)
  - The most widely used kernels include linear, polynomial (poly), radial basis function (rbf) and sigmoid | default value: RBF
- More details can be found in Appendix

Choosing the right values for hyperparams

- Epsilon:
  - Start Small: A smaller ϵ (e.g., 0.01) allows for a more sensitive model, capturing more details of the training data
  - Grid Search: Use a grid search to test different values for ϵ. Common ranges are between 0 and 1, depending on the scale of your data
- C:
  - Start with Defaults: Common default values are 1 or 10.
  - Logarithmic Scale in Grid Search: Use a logarithmic scale (e.g., 0.1, 1, 10, 100) to cover a wide range of values effectively
- Kernel:
  - Data Distribution: Start with a linear kernel if you suspect a linear relationship. For more complex relationships, consider the RBF kernel.
  - Experimentation in Grid Search: Try different kernels and see how they perform on your data
    - Hyperparameter Tuning: Each kernel may have specific hyperparameters (like σ for the RBF kernel, or degree for polynomial kernel) that also need tuning.

Scaling in Support Vector Regression (SVR)

- Support Vector Regressor is a distance-based regression algorithm that uses (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

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
pipe_svr = Pipeline(steps=[
        ("scaler", RobustScaler()),
        ("svr", TransformedTargetRegressor(regressor=SVR(),
         transformer=PowerTransformer(method='yeo-johnson')))
])
# parameter grid
parameter_grid = [
 {'svr__regressor__C': [1, 10, 100, 1000], 'svr__regressor__ kernel': ['linear']},
 {'svr__ regressor__ C': [1, 10, 100, 1000], 'svr__regressor__gamma': [0.001, 0.0001], 'svr__regressor__kernel': ['rbf']},
 {'svr__ regressor__C': [1, 10, 100, 1000], 'svr__regressor__degree': [1, 2, 3, 4, 5, 6], 'svr__regressor__ 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:
 0.7649632977483316

Model performance on validation dataset
--------------------------------------
RMSE is 2.9887655163221054
R2 score is 0.8859078053060818
```

SVR model outperforms the polynomial model. It achieves slightly higher R2 score.

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 which tends to make more accurate predictions than individual (weak) base models
- We have three kinds of ensemble methods using:
  - Sequential Homogeneous Learners (Boosting), e.g. AdaBoostRegressor, GradientBoostingRegressor, LightGBM (installation) XGBoost (installation), CatBoost (installation)
  - Parallel Homogeneous Learners (Bagging), e.g. BaggingRegressor, RandomForestRegressor
  - Parallel Heterogeneous Learners (Stacking), e.g. StackingRegressor

For more info please see the Appendix in the extended presentation

Is scaling/unskewing needed?

- Ensemble methods such as Random Forest, XGBoost, and AdaBoost generally **do not require feature scaling**, since they are **tree-based** and **do not rely on distance metrics** or the **magnitude of feature values**.
- A skewed dependent (target) variable is **not inherently problematic** for ensemble methods, as they **do not rely on assumptions** such as the normality of residuals or homoscedasticity, which are required in linear regression models. However, if the target distribution is **highly skewed**, a transformation (e.g., log or Box–Cox) can sometimes still improve model performance or interpretability.

RandomForestRegressor with GridSearchCV

Warning: This may run several minutes!!

```
from sklearn.ensemble import RandomForestRegressor
# Number of trees in random forest
n_estimators = [int(x) for x in np.linspace(start = 200, stop = 1000, num = 10)]
# 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 = {'rf__regressor__n_estimators': n_estimators,
                  'rf__regressor__max_depth': max_depth,
                  'rf__regressor__min_samples_split': min_samples_split,
                  'rf__regressor__min_samples_leaf': min_samples_leaf,
                  'rf__regressor__bootstrap': bootstrap}
pipe = Pipeline([("rf", TransformedTargetRegressor(regressor=RandomForestRegressor(),
transformer=PowerTransformer(method='yeo-johnson')))])

# make grid_RF object for GridSearchCV and fit the dataset
grid_RF = GridSearchCV(pipe, parameter_grid, scoring = 'r2', n_jobs=-1)
grid_RF.fit(X_train, y_train)
# print results
print(" Results from Grid Search " )
print("\n The best estimator across ALL searched params:\n", grid_RF.best_estimator_)
print("\n The best score across ALL searched params:\n", grid_RF.best_score_)
print("\n The best parameters across ALL searched params:\n", grid_SVR.best_params_)
```

```
The best parameters across ALL searched params:
 {'rf__regressor__bootstrap': True, 'rf__regresso
r__max_depth': 50, 'rf__regressor__min_samples_le
af': 1, 'rf__regressor__min_samples_split': 5, 'r
f__regressor__n_estimators': 200}

Model performance on testing dataset
------------------------------------
RMSE is 3.1597194793136025
R2 score is 0.8724826432366622
```

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

Tuning hyperparameters

- It is crucial to systematically study each model's hyperparameters and carefully select a range of values for each one.
- This helps ensure that you avoid excessive training times during the grid search process while still capturing the model's performance characteristics. Here's a step-by-step approach:
  1. Understand the Hyperparameters: Familiarize yourself with the hyperparameters of the specific ensemble models you are using (e.g., Random Forest, Gradient Boosting, etc.). Research their effects on model performance and interpretability.
  2. Identify Important Hyperparameters: Focus on the hyperparameters that most significantly impact model performance. For example:
     - For Random Forest: Number of trees (n_estimators), maximum depth of trees (max_depth), minimum samples per leaf (min_samples_leaf).
     - For Gradient Boosting: Learning rate (learning_rate), number of boosting stages (n_estimators), and maximum depth of individual trees (max_depth).

Tuning hyperparameters

3. Define Value Ranges: Based on your understanding of the model and the dataset, choose reasonable ranges for the hyperparameters. Start with broader ranges to explore their influence, then narrow them down as you gain insights after performing GridSearch. For instance:
   - n_estimators: 50, 100, 150, 200
   - max_depth: 3, 5, 7, 10
   - min_samples_leaf: 1, 2, 5
4. Utilize Logarithmic Scaling: For parameters that can vary widely, such as the learning rate in Gradient Boosting, consider using a logarithmic scale to cover a broader range more effectively. For example, you might test values like 0.01, 0.1, 1, and 10.
5. Start with Coarse Search: Begin with a coarser grid to get an overview of how the hyperparameters interact. This will help you identify promising regions in the hyperparameter space without extensive computation.
6. Implement Grid Search CV

Tuning hyperparameters

7. Analyze Results: After the grid search, review the results to identify the best-performing hyperparameter combinations. Use validation metrics to evaluate how well the model generalizes to unseen data.
8. Refine and Iterate: If needed, refine the hyperparameter ranges based on the results of the initial grid search. You can perform a second round of tuning with more focused ranges around the best parameters identified.