

✓ Polynomial Regression

Agenda:

- Introduction to Polynomial Regression
 - What is Polynomial Regression?
 - Why Polynomial Regression?
 - Applications of Polynomial Regression
 - How Does Polynomial Regression Handle Non-linear Data
- Polynomial Regression vs Linear Regression
- Exploring Data and Simulation for Polynomial Regression
- Generalization
- Bias and Variance Tradeoff
 - What are Bias and variance?
 - Underfitting, Overfitting Models
 - Why is Bias Variance Tradeoff?
 - Simulation for Bias Variance Tradeoff
- Validation in Model Training
 - Validation Techniques
 - Cross Validation
 - Implementation of K-Fold Cross Validation
- Controlling Overfitting
 - Introduction to Regularization
 - Types of Regularization
 - Shrinkage Parameter(λ)
 - Simulation for Regularization

✓ Introduction to Polynomial Regression

✓ What is Polynomial Regression?

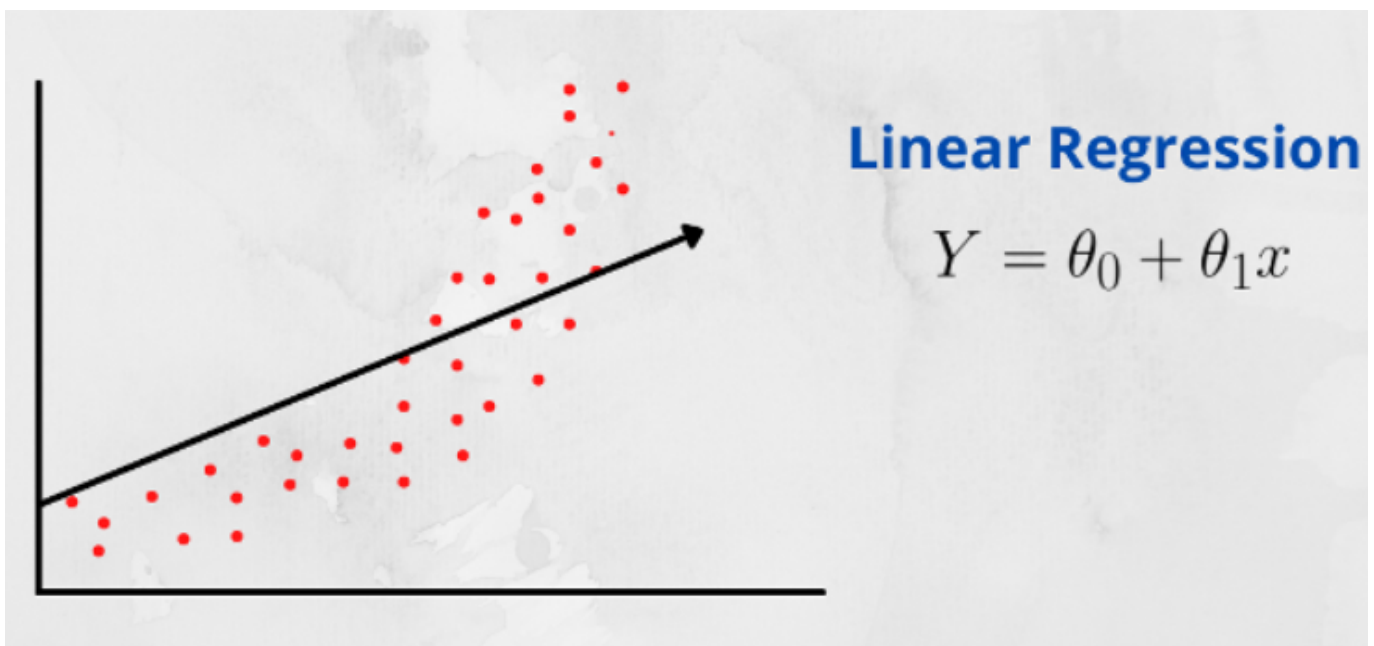
Polynomial regression is a type of regression analysis used to model the relationship between the independent variable (X) and the dependent variable (Y) as an nth degree polynomial.

It helps in capturing the non-linear relationships between the variables, which cannot be modeled accurately using simple linear regression.

Polynomial regression is particularly useful when the relationship between X and Y is not linear but follows a curve.

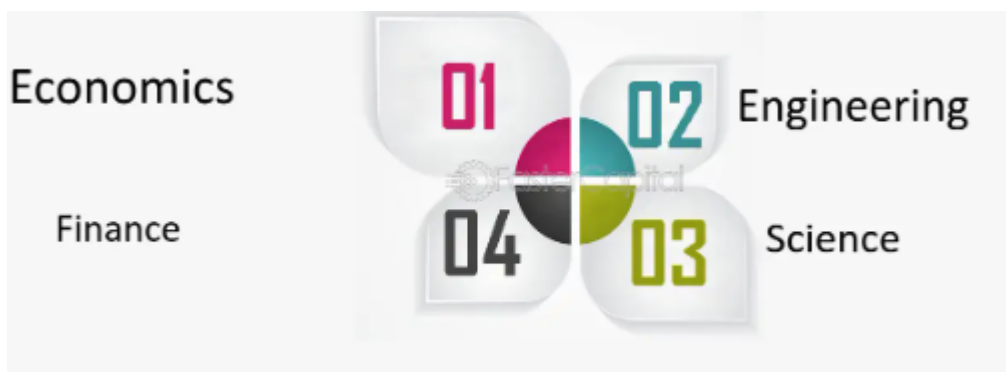
✓ Why Polynomial Regression?

A simple linear regression algorithm only works when the relationship between the data is linear. But suppose we have non-linear data, then linear regression will not be able to draw a best-fit line. Simple regression analysis fails in such conditions. Consider the below diagram, which has a non-linear relationship, and you can see the linear regression results on it, which does not perform well, meaning it does not come close to reality. Hence, we introduce polynomial regression to overcome this problem, which helps identify the curvilinear relationship between independent and dependent variables.



✓ Applications of Polynomial Regression

- Polynomial regression has a wide range of applications, particularly where relationships between the independent and dependent variables are non-linear. Below are some key areas where polynomial regression is commonly used:



1. Economics and Finance

- Stock Price Prediction: Polynomial regression is used to model the relationship between time and stock prices or between multiple financial indicators and stock prices.
- Market Trends: To capture non-linear patterns in market data, such as predicting the future value of a financial asset based on past performance.
- Risk Management: Modeling risk factors and predicting potential financial losses or gains over time when the relationship is not strictly linear.

2. Engineering and Physics

- Motion Analysis: In physics, polynomial regression is used to describe the motion of objects, especially when acceleration is not constant. For example, predicting the trajectory of a projectile or the speed of an object over time.
- Material Science: To model stress-strain relationships in materials, where the behavior of materials under load might follow non-linear patterns.
- Signal Processing: Polynomial regression can be used to filter signals or model noise in communication systems.

3. Environmental Science

- Temperature Prediction: Polynomial regression can be applied to model temperature changes over time or predict future temperatures based on historical data. This is useful in climate modeling and weather forecasting.
- Pollution Analysis: Modeling the concentration of pollutants in the environment over time or in relation to other factors like industrial activities or vehicle emissions.
- Population Growth: In ecological studies, polynomial regression helps in modeling non-linear population growth or species behavior over time.

✓ How Does Polynomial Regression Handle Non-Linear Data?

Polynomial regression is a form of Linear regression where only due to the Non-linear relationship between dependent and independent variables, we add some polynomial terms to linear regression to convert it into Polynomial regression.

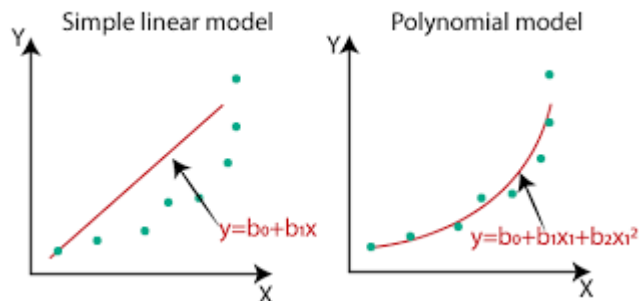
In polynomial regression, the relationship between the dependent variable and the independent variable is modeled as an n th-degree polynomial function. When the polynomial is of degree 2, it is called a quadratic model; when the degree of a polynomial is 3, it is called a cubic model, and so on.

Suppose we have a dataset where variable X represents the Independent data and Y is the dependent data. Before feeding data to a model in the preprocessing stage, we convert the input variables into polynomial terms using some degree.

The equation of polynomials becomes something like this:

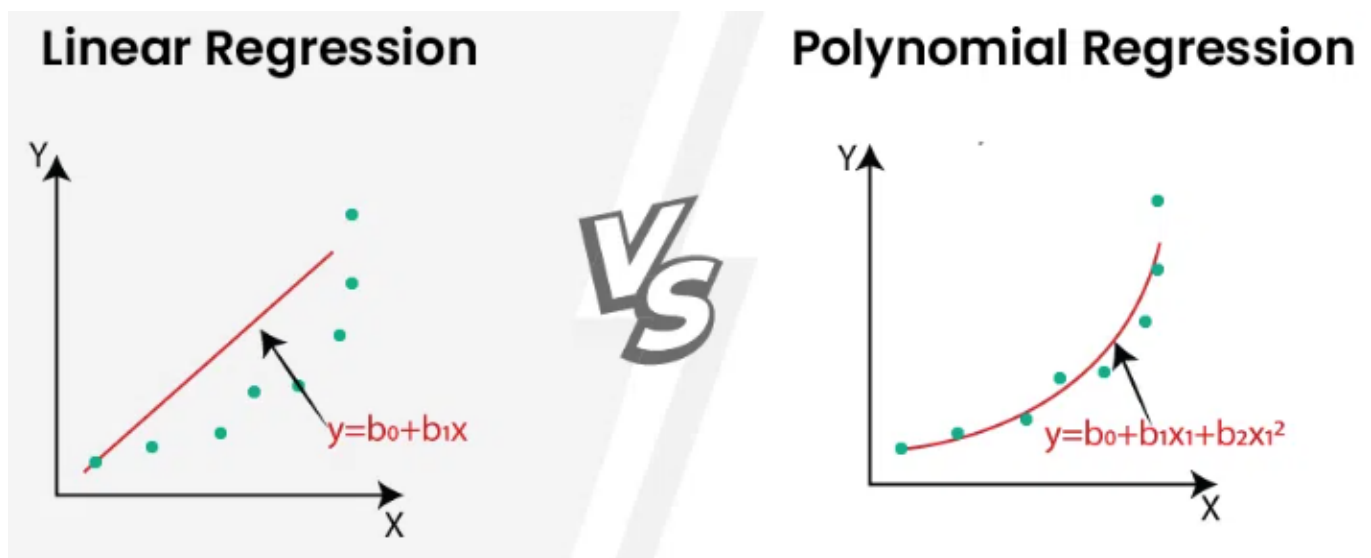
$$y = a_0 + a_1x_1 + a_2x_1^2 + \dots + a_nx_1^n$$

The degree of order which to use is a Hyperparameter, and we need to choose it wisely. But using a high degree of polynomial tries to overfit the data, and for smaller values of degree, the model tries to underfit, so we need to find the optimum value of a degree. Polynomial Regression models are usually fitted with the method of least squares.



✓ Polynomial Regression vs Linear Regression

Polynomial and linear regression are two types of regression models used for predicting a continuous target variable. While both fall under the umbrella of supervised learning and share some similarities, they have key differences in how they model the relationship between the independent and dependent variables.



- Linear Regression:
 - Simplicity: Linear regression is easier to interpret and works well when the relationship between variables is linear.
 - Limitations: It struggles to model non-linear relationships, often leading to underfitting when data shows curvature.
 - Use Cases:

- Simple relationships where changes in one variable consistently correspond to changes in the target.
- Predicting trends in business, financial forecasting, and simple cause-effect relationships.
- Polynomial Regression:
 - Flexibility: Can model both linear and non-linear relationships. Higher-degree polynomials offer more flexibility.
 - Limitations: High-degree polynomials can lead to overfitting if the model starts capturing noise in the data, and it becomes harder to interpret.
 - Use Cases:
 - Situations where the relationship between independent and dependent variables follows a curve (e.g., quadratic, cubic, etc.).
 - Fields such as physics, biology, and economics where outcomes may increase/decrease at varying rates.

✓ Polynomial Regression Pipeline

- A Polynomial Regression pipeline refers to a structured process for preparing data, applying polynomial transformations, training a model, and making predictions. Pipelines help streamline the workflow, making it easier to automate tasks and ensure that the same sequence of steps is applied consistently to both training and test data. In Polynomial Regression, the process includes transforming the input features into polynomial features and then fitting a linear model to these transformed features.

✓ Exploring Data

```
import numpy as np # Importing NumPy library for numerical computing
import pandas as pd # Importing Pandas library for data manipulation
import seaborn as sns # Importing Seaborn library for statistical data visualiza
import matplotlib.pyplot as plt # Importing Matplotlib library for creating plot
```

✓ Generating Sample Data

- Random data points for X are generated using numpy's random number generator.
- The relationship between X and Y is defined as a polynomial equation with noise added to simulate real-world data.

```
import numpy as np # Importing NumPy library for numerical computing
import matplotlib.pyplot as plt # Importing Matplotlib library for creating plot

# Set a seed for reproducibility
np.random.seed(1)

# Generate random data for x
X = np.random.rand(50, 1)

# Define a polynomial relationship between x and y with some noise
y = 0.7*(X**5) - \
    2.1*(X**4) + \
    2.3*(X**3) + \
    0.2*(X**2) + \
    0.3*(X) + \
    0.4*np.random.rand(50, 1) # Adding noise to the data
```

In above code block, we're generating random data for the independent variable x using NumPy's random.rand function and creating a polynomial relationship with some added noise for the dependent variable y. We're using NumPy to handle array operations efficiently. Additionally, we've set a seed using np.random.seed(1) for reproducibility of the random data generation.

✓ Visualizing the Data

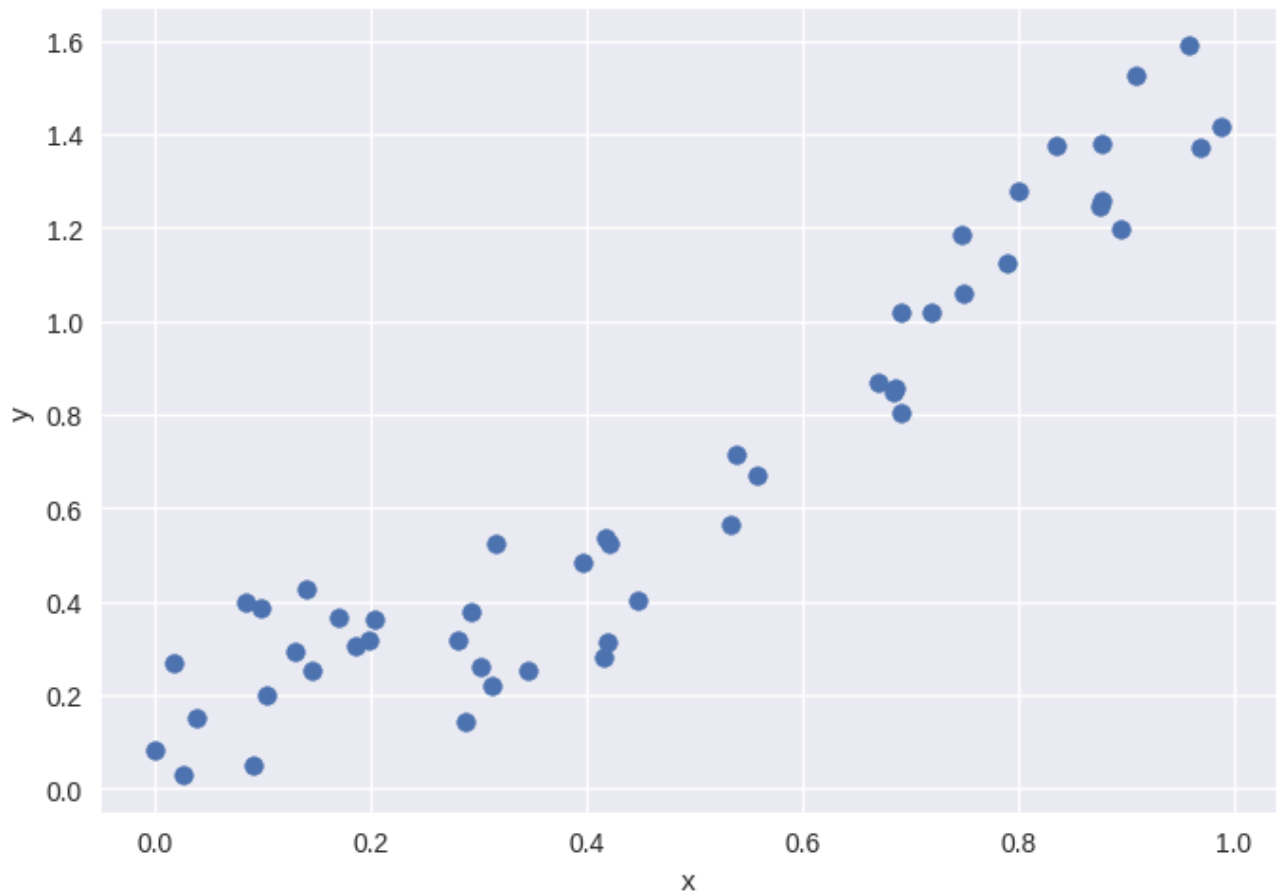
- Scatter plot is used to visualize the relationship between X and Y.

```
# Create a figure to plot the data
fig = plt.figure()

# Scatter plot the data points
plt.scatter(X, y)

# Label x and y axes
plt.xlabel("x")
plt.ylabel("y")

# Show the plot
plt.show()
```



In above code block, we're creating a scatter plot to visualize the relationship between the independent variable x and the dependent variable y . We use Matplotlib's scatter function to create the plot and label the x and y axes using `xlabel` and `ylabel` respectively. Finally, we display the plot using `plt.show()`.

Q. Do you think x and y exhibit linear relationship?

✓ Observation

- **Non-linearity:** When we plot the data points of X and Y on a graph, we notice that the points do not fall along a straight line. Instead, they seem to follow a curve or a pattern that is not linear.
- **Curve Fit:** Upon closer inspection, we observe that a curved line, rather than a straight line, better aligns with the distribution of the data points. This means that a polynomial function, which can represent curves, might be more suitable for describing the relationship between X and Y .

✓ Implication

- **Limitation of Linear Regression:** Simple linear regression assumes a linear relationship between the independent variable (X) and the dependent variable (Y). However, in this case, the data does not adhere to this assumption. Thus, using a linear regression model may not adequately capture the underlying trend in the data.
- **Advantage of Polynomial Regression:** Polynomial regression, on the other hand, allows for the modeling of non-linear relationships by introducing higher-order terms of the independent variable (X). By incorporating terms like X^2 , X^3 , and so on, polynomial regression can better capture the curvature observed in the data.

The non-linear nature of the relationship between X and Y and understanding the limitations of linear regression, we can make an informed decision to employ polynomial regression to better model the data. This approach enables us to capture the underlying trend and variability in the data more effectively, leading to more accurate predictions and insights.

✓ Simulation

```
from sklearn.linear_model import LinearRegression # Importing LinearRegression m
from sklearn.preprocessing import StandardScaler # Importing StandardScaler from
```

In above code block, we're importing the LinearRegression model and the StandardScaler from scikit-learn. We'll likely use the LinearRegression model to fit a linear regression to our data, and the StandardScaler for feature scaling, which is often important in machine learning algorithms.

✓ Linear Regression

```
model = LinearRegression() # Creating an instance of LinearRegression model

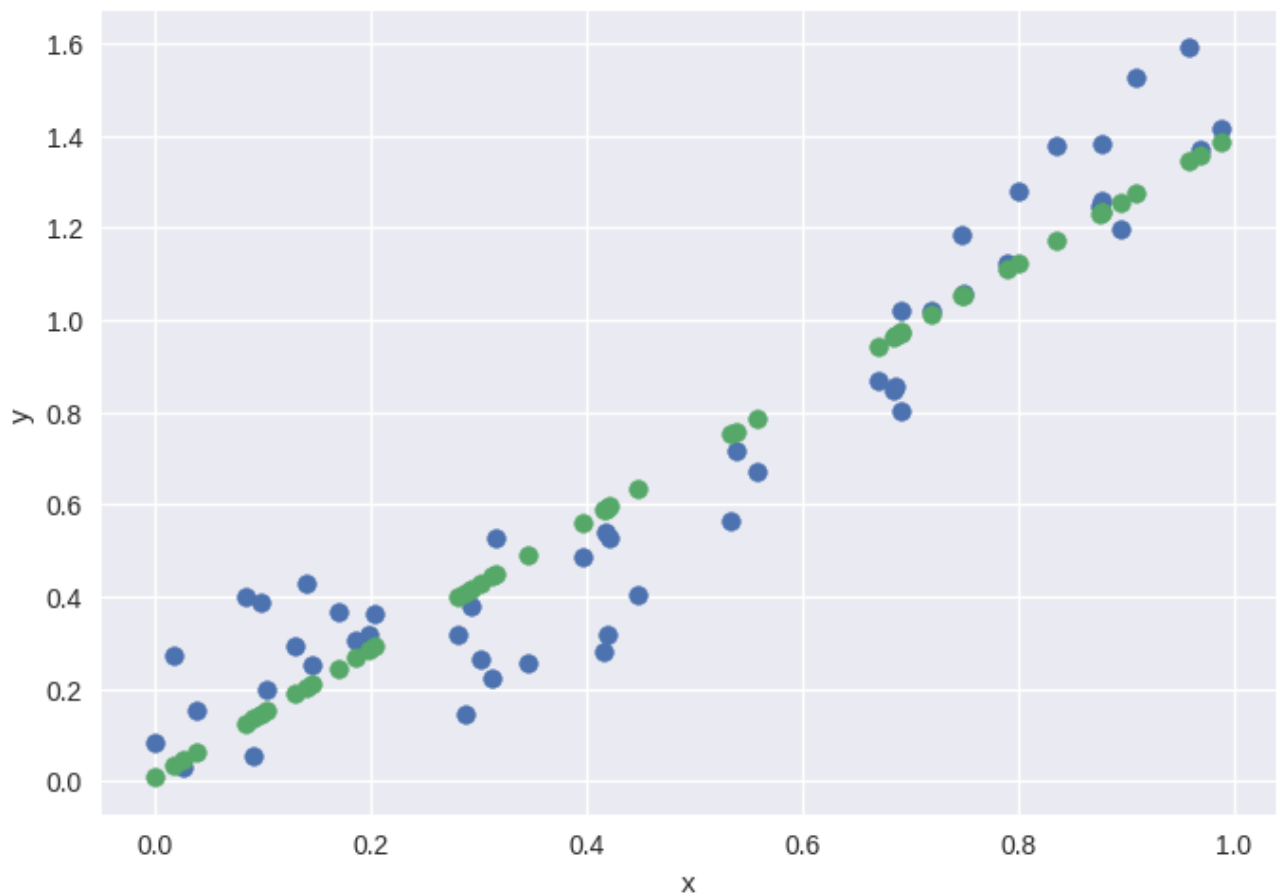
model.fit(X, y) # Fitting the model to the data

output = model.predict(X) # Making predictions using the trained model

fig = plt.figure() # Creating a figure for plotting

# Plotting the original data and the model predictions
plt.scatter(X, y, label="Data")
plt.scatter(X, output, label="Predictions")
plt.xlabel("x")
plt.ylabel("y")
plt.show() # Showing the plot

model.score(x, y) # Calculating the R^2 score of the model
```

0.8919326382437994

In above code block, we're fitting a linear regression model to our data using the `LinearRegression` model from `scikit-learn`. We fit the model to the data using the `fit` method, make predictions using the `predict` method, and then visualize the original data and the model predictions using a scatter plot. Finally, we calculate the R^2 score of the model using the `score` method, which measures the goodness of fit of the model to the data.

The R^2 score of approximately 0.892 indicates that our linear regression model explains about 89.2% of the variance in the dependent variable y based on the independent variable x . This suggests that the model fits the data quite well, as a higher R^2 score closer to 1 indicates better explanatory power of the model.

✓ Polynomial Regression of Degree 2

```

X_deg2 = np.hstack([X, X**2]) # Adding a quadratic feature to the independent va

model = LinearRegression() # Creating an instance of LinearRegression model

model.fit(X_deg2, y) # Fitting the model to the data with the quadratic feature

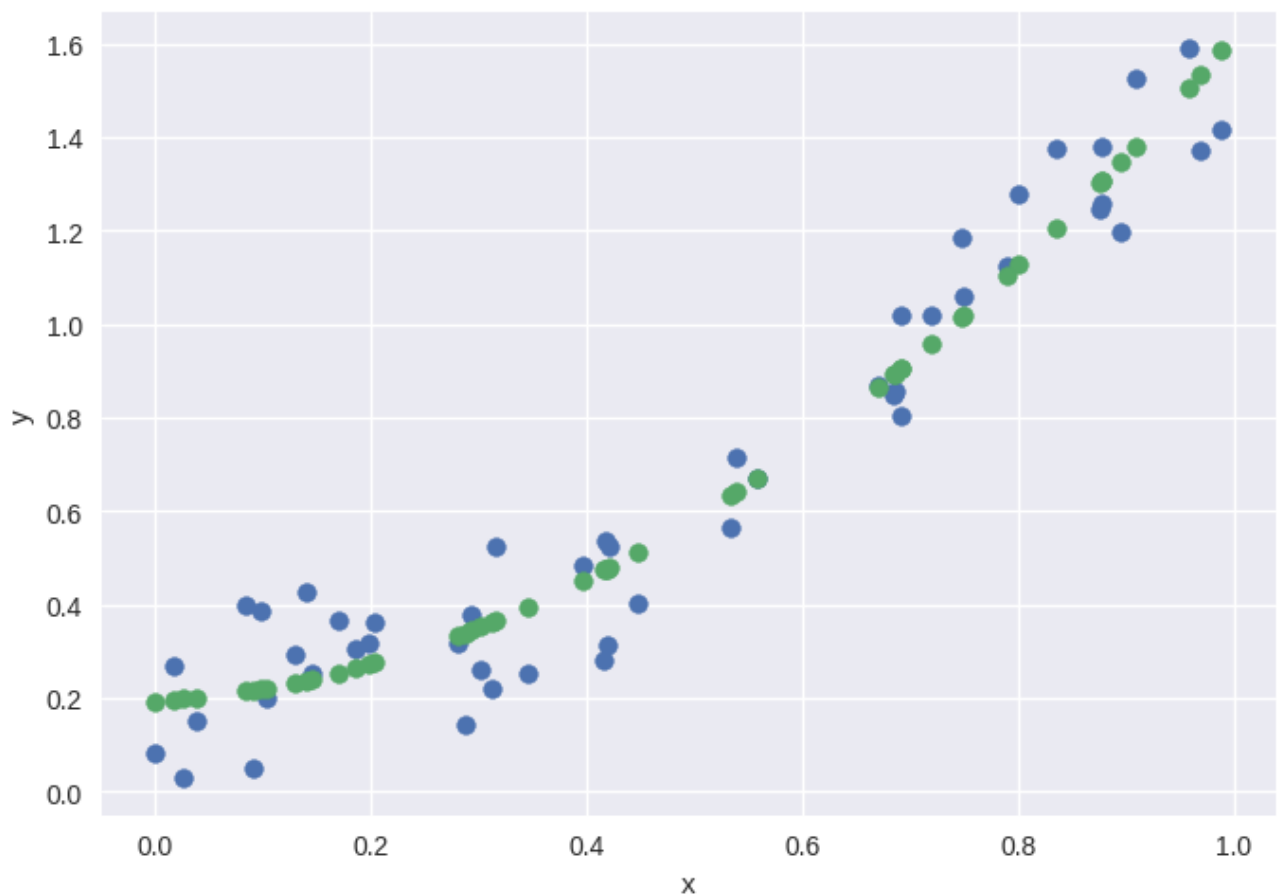
output = model.predict(X_deg2) # Making predictions using the trained model

fig = plt.figure() # Creating a figure for plotting

# Plotting the original data and the model predictions
plt.scatter(X, y, label="Data")
plt.scatter(X, output, label="Predictions")
plt.xlabel("x")
plt.ylabel("y")
plt.show() # Showing the plot

model.score(X_deg2, y) # Calculating the R^2 score of the model with the quadrat

```



0.937213227713278

The R^2 score of approximately 0.937 in the code block where we added a quadratic feature to the independent variable x indicates that the quadratic regression model explains about 93.7% of the variance in the dependent variable y based on the combined linear and quadratic features of x .

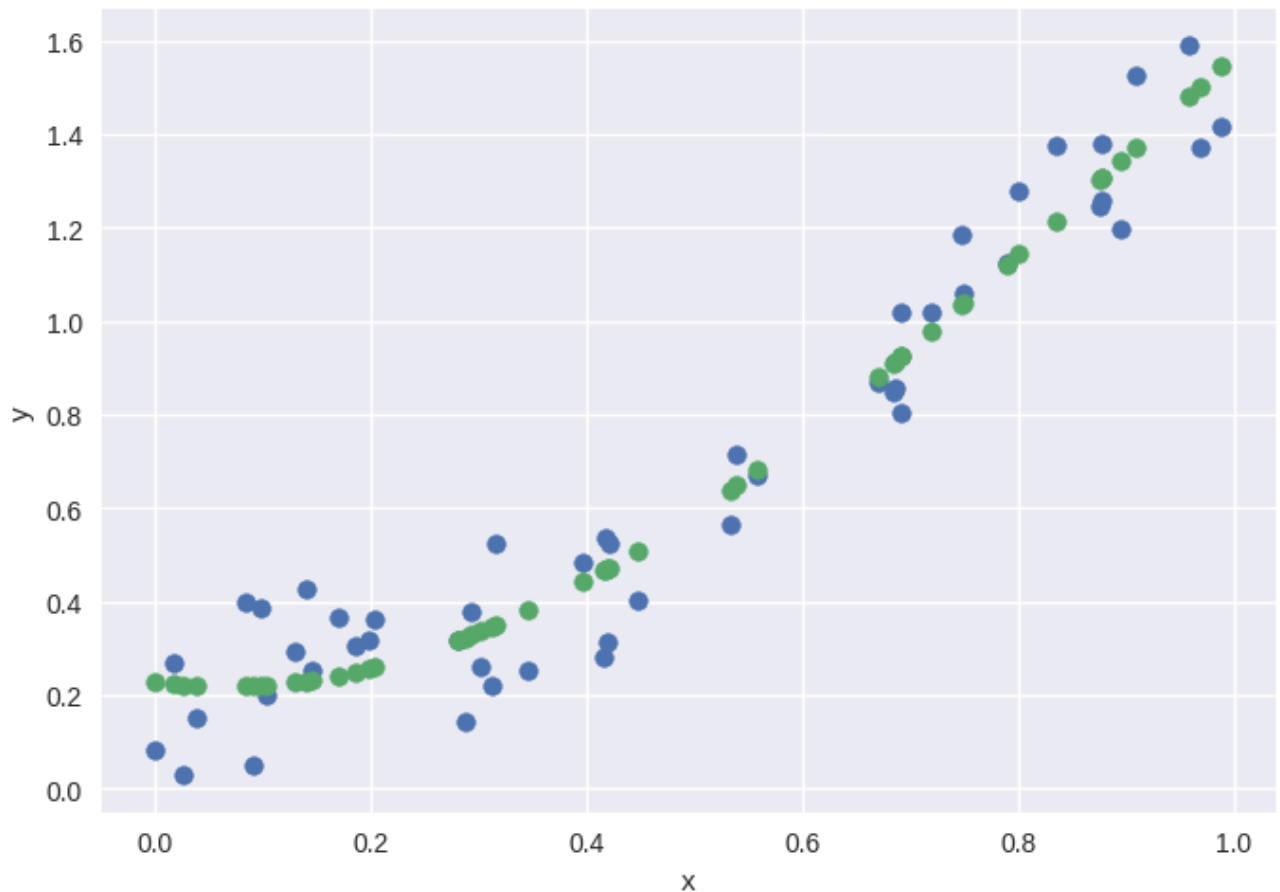
Comparing this R^2 score with the Linear Regression (0.892) where only linear features were used, we observe a significant improvement. This suggests that the quadratic model better captures the underlying relationship between x and y , resulting in a higher degree of explanation for the variance in y . Therefore, the quadratic regression model provides a better fit to the data compared to the linear regression model.

✓ Polynomial Regression of Degree 3

```
X_deg3 = np.hstack([X, X**2, X**3]) # Adding cubic features to the independent v
model = LinearRegression() # Creating an instance of LinearRegression model
model.fit(X_deg3, y) # Fitting the model to the data with cubic features
output = model.predict(X_deg3) # Making predictions using the trained model
fig = plt.figure() # Creating a figure for plotting

# Plotting the original data and the model predictions
plt.scatter(X, y, label="Data")
plt.scatter(X, output, label="Predictions")
plt.xlabel("x")
plt.ylabel("y")
plt.show() # Showing the plot

model.score(X_deg3, y) # Calculating the  $R^2$  score of the model with cubic featu
```



0.9384895307987051

✓ Using Built-in Methodologies for Polynomial Regression

```
from sklearn.preprocessing import PolynomialFeatures # Importing PolynomialFeatu

scaler = StandardScaler() # Creating an instance of StandardScaler for feature s
poly = PolynomialFeatures() # Creating an instance of PolynomialFeatures for gen

poly = PolynomialFeatures(2) # Creating an instance of PolynomialFeatures with d
X_poly = poly.fit_transform(X) # Generating polynomial features up to order 2 fr

X_poly_scaled = scaler.fit_transform(X_poly) # Scaling the polynomial features u

model = LinearRegression() # Creating an instance of LinearRegression model
model.fit(X_poly_scaled, y) # Fitting the model to the scaled polynomial feature

print(model.score(X_poly_scaled, y)) # Printing the R^2 score of the model
```



0.937213227713278

```
poly = PolynomialFeatures(3) # Creating an instance of PolynomialFeatures with d
X_poly = poly.fit_transform(X) # Generating polynomial features up to order 3 fr

X_poly_scaled = scaler.fit_transform(X_poly) # Scaling the polynomial features u

model = LinearRegression() # Creating an instance of LinearRegression model
model.fit(X_poly_scaled, y) # Fitting the model to the scaled polynomial feature

print(model.score(X_poly_scaled, y)) # Printing the R^2 score of the model
```

```
⇒ 0.9384895307987051
```

```
scores = [] # Initialize an empty list to store R^2 scores for each degree of po

# Iterate over degrees of polynomial features from 1 to 5
for i in range(1, 6):
    poly = PolynomialFeatures(i) # Create an instance of PolynomialFeatures with
    X_poly = poly.fit_transform(X) # Generate polynomial features up to degree i

    scaler = StandardScaler() # Create an instance of StandardScaler for feature
    scaler.fit(X_poly) # Fit the scaler to the polynomial features
    X_poly_scaled = scaler.fit_transform(X_poly) # Scale the polynomial features

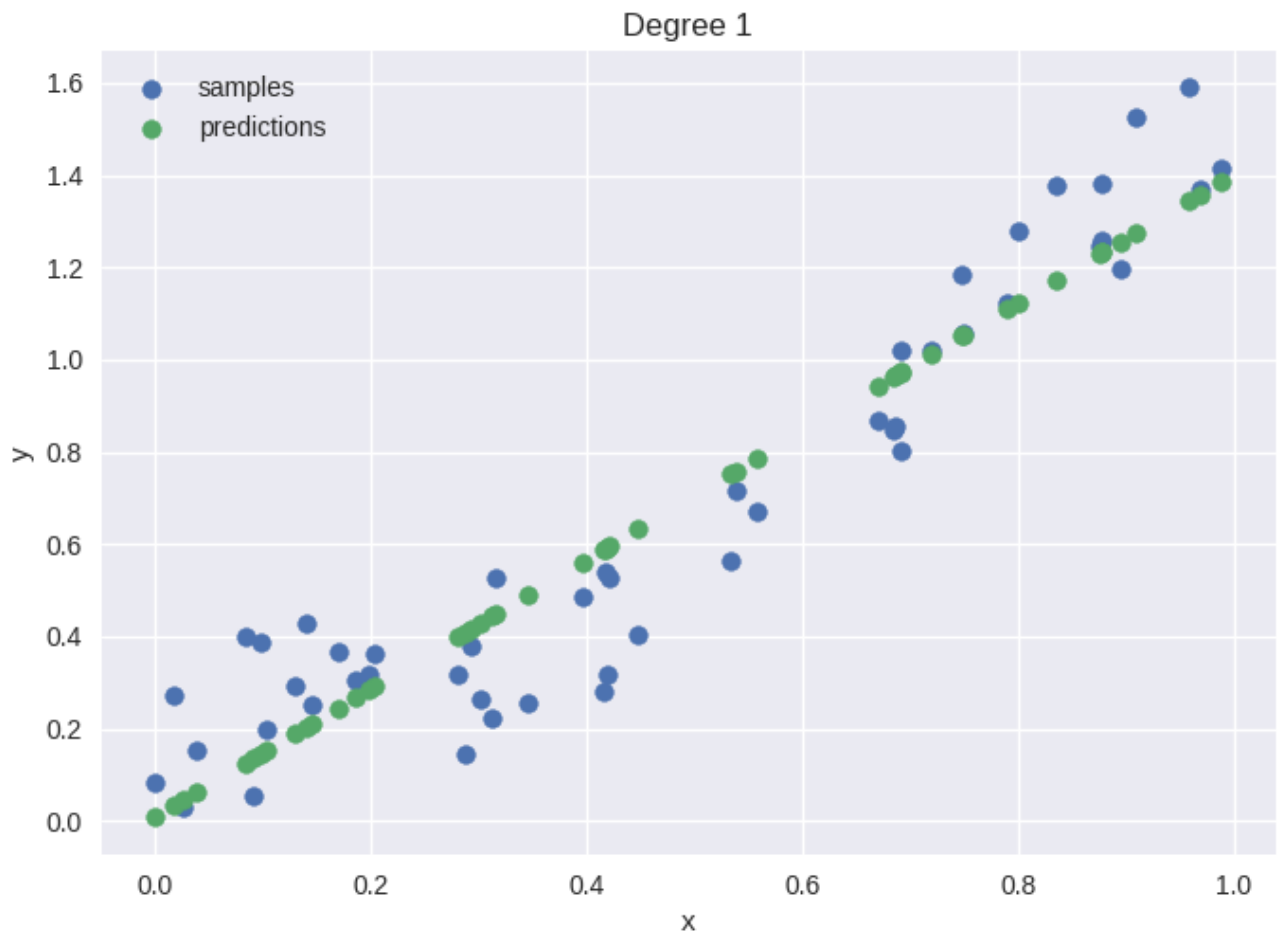
    model = LinearRegression() # Create an instance of LinearRegression model
    model.fit(X_poly_scaled, y) # Fit the model to the scaled polynomial feature

    output = model.predict(X_poly_scaled) # Make predictions using the trained m

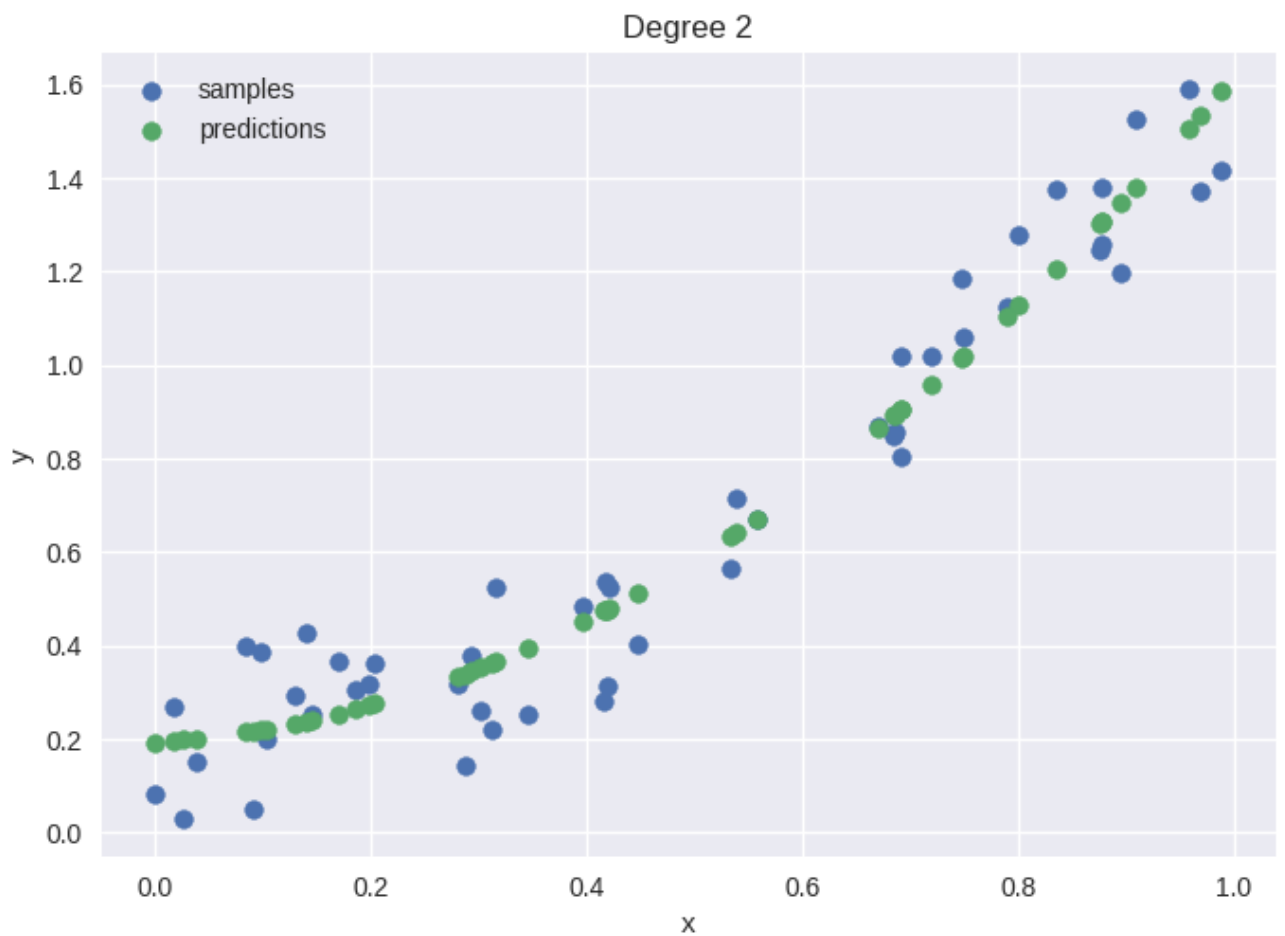
    # Plot the original data and the model predictions
    fig = plt.figure()
    plt.scatter(X, y, label="samples")
    plt.scatter(X, output, label="predictions")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.title(f"Degree {i}")
    plt.legend()
    plt.show()

    # Display the R^2 score of the model
    display(model.score(X_poly_scaled, y))

    # Append the R^2 score to the scores list
    scores.append(model.score(X_poly_scaled, y))
```



0.8919326382437994



0.937213227713278



As the degree of the polynomial increases, the R^2 score generally improves, indicating that the higher-degree polynomial models better fit the data. This is expected because higher-degree polynomials can capture more complex relationships between the features and the target variable. However, it's important to note that excessively high-degree polynomials can lead to overfitting, where the model learns to fit the noise in the data rather than the underlying pattern, resulting in poor generalization to new data. Therefore, it's essential to strike a balance between model complexity and performance. In this case, the improvement in R^2 starts to plateau around degree 4 and 5, suggesting that increasing the degree further may not significantly improve model performance.

```
scores = [] # Initialize an empty list to store R^2 scores for each degree of po

# Iterate over degrees of polynomial features from 1 to 39
for i in range(1, 40):
    poly = PolynomialFeatures(i) # Create an instance of PolynomialFeatures with
    X_poly = poly.fit_transform(X) # Generate polynomial features up to degree i

    scaler = StandardScaler() # Create an instance of StandardScaler for feature
    scaler.fit(X_poly) # Fit the scaler to the polynomial features
    X_poly_scaled = scaler.fit_transform(X_poly) # Scale the polynomial features

    model = LinearRegression() # Create an instance of LinearRegression model
    model.fit(X_poly_scaled, y) # Fit the model to the scaled polynomial feature

    output = model.predict(X_poly_scaled) # Make predictions using the trained m

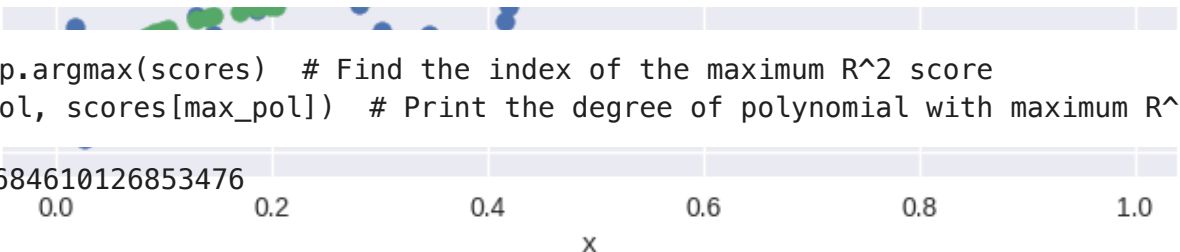
    # fig = plt.figure()
    # plt.scatter(X, y, label="samples")
    # plt.scatter(X, output, label="predictions")
    # plt.xlabel("x")
    # plt.ylabel("y")
    # plt.title(f"Degree {i}")
    # plt.show()

    # display(model.score(X_poly_scaled, y))

    scores.append(model.score(X_poly_scaled, y)) # Append the R^2 score to the s

max_pol = np.argmax(scores) # Find the index of the maximum R^2 score
print(max_pol, scores[max_pol]) # Print the degree of polynomial with maximum R^2
```

⇒ 30 0.9684610126853476



✓ Generalization

Generalization refers to the ability of a machine learning model to perform well on unseen or new data that it hasn't been trained on. In other words, a model is said to generalize well if it can

accurately make predictions on data that it hasn't encountered before.

When we train a machine learning model, we aim to capture the underlying patterns in the training data so that it can learn to make predictions. However, the ultimate goal is not just to perform well on the training data itself but to generalize well to new, unseen data. This is crucial because in real-world applications, the model will encounter data that it hasn't been trained on, and its performance on this unseen data is what matters.

Achieving good generalization involves finding the right balance between capturing the underlying patterns in the training data without fitting too closely to the noise or idiosyncrasies of the training set. Overfitting occurs when a model learns the training data too well, capturing noise or random fluctuations that are specific to the training set but do not represent the underlying patterns of the data. On the other hand, underfitting occurs when a model is too simple to capture the underlying patterns in the data.

```
# Set a seed for reproducibility
np.random.seed(1)

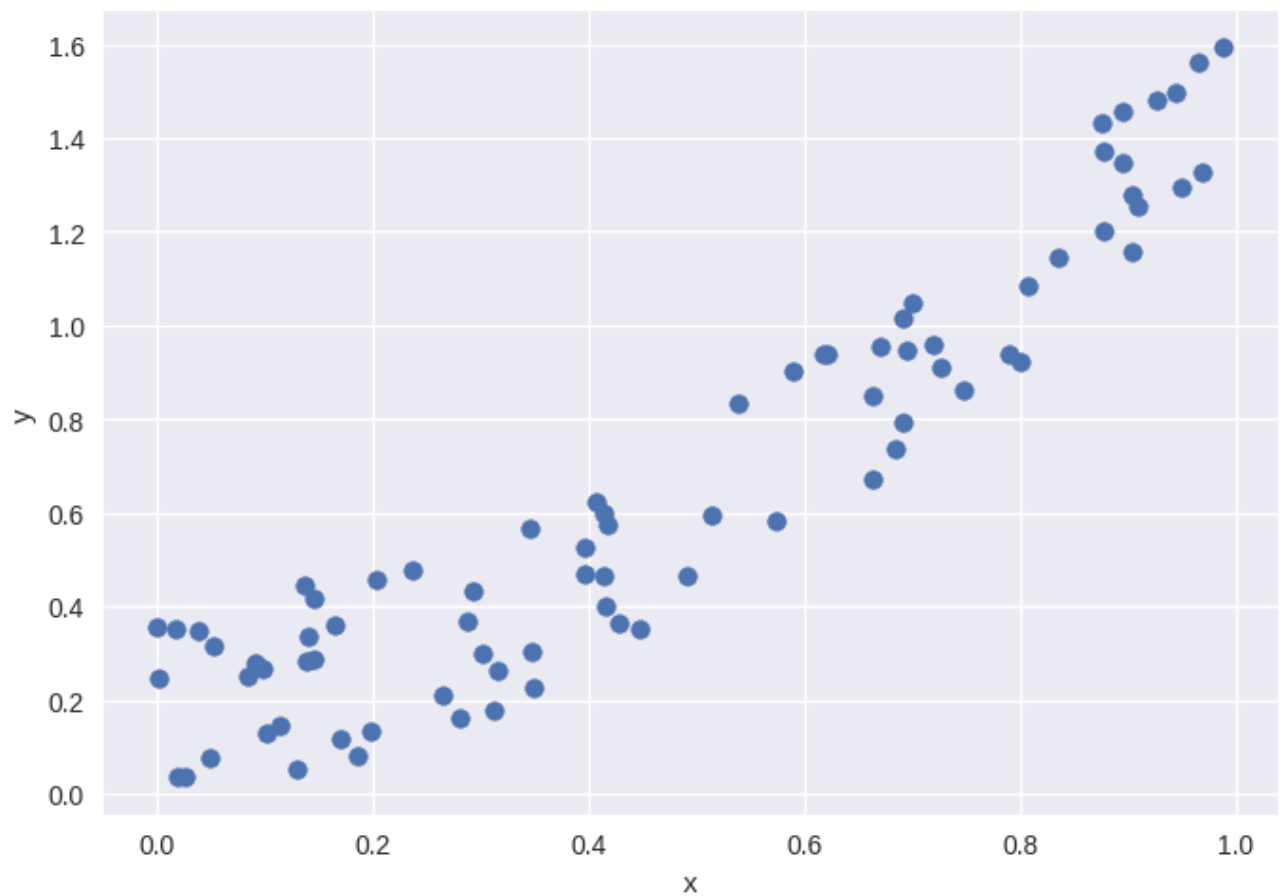
# Generate random data for x
X = np.random.rand(100, 1)

# Define a polynomial relationship between x and y with some noise
y = 0.7*(X**5) - \
    2.1*(X**4) + \
    2.3*(X**3) + \
    0.2*(X**2) + \
    0.3*(X) + \
    0.4*np.random.rand(100, 1) # Adding noise to the data

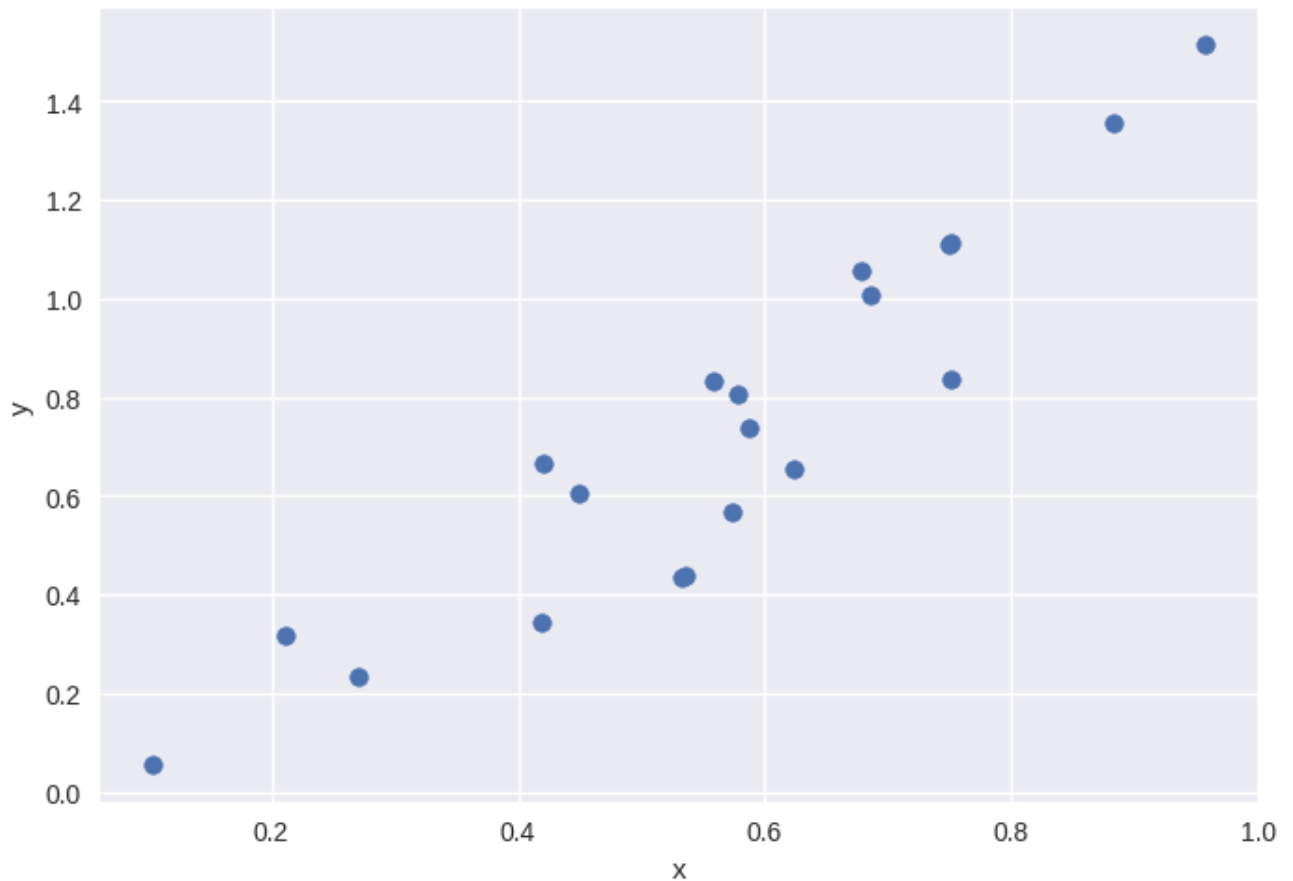
from sklearn.model_selection import train_test_split

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_s

plt.scatter(X_train, y_train)
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

```
plt.scatter(X_test, y_test)
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



```
# sequence: create polynomial feature, scale, train(f(t))
```

```
from sklearn.pipeline import make_pipeline
```

```
degree = 5 # Define the degree of the polynomial
polyreg_scaled = make_pipeline(PolynomialFeatures(degree), scaler, LinearRegressi
polyreg_scaled.fit(X_train, y_train)
```

```
# Display R^2 scores for training and testing data
display(polyreg_scaled.score(X_train, y_train))
display(polyreg_scaled.score(X_test, y_test))
```



```
0.9214962587812688
0.8656262710391975
```

```
# sequence: create polynomial feature, scale, train(f(t))
```

```
from sklearn.pipeline import make_pipeline
```

```
degree = 30 # Define a higher degree of the polynomial
polyreg_scaled = make_pipeline(PolynomialFeatures(degree), scaler, LinearRegressi
polyreg_scaled.fit(X_train, y_train)
```

```
# Display R^2 scores for training and testing data
display(polyreg_scaled.score(X_train, y_train))
display(polyreg_scaled.score(X_test, y_test))
```

```
0.9389724819985867  
0.8094469425937477
```

We repeat the process with a higher degree of polynomial (degree = 30) to observe how the model performs with a more complex polynomial function. The pipeline is used again to create polynomial features, scale the features, and fit a linear regression model. Finally, we display the R scores for both the training and testing data to evaluate the model's performance.

✓ Bias and Variance Tradeoff

Whenever we discuss model prediction, it's important to understand prediction errors (bias and variance). There is a tradeoff between a model's ability to minimize bias and variance. Gaining a proper understanding of these errors would help us not only to build accurate models but also to avoid the mistake of overfitting and underfitting.

So let's start with the basics and see how they make difference to our machine learning Models.

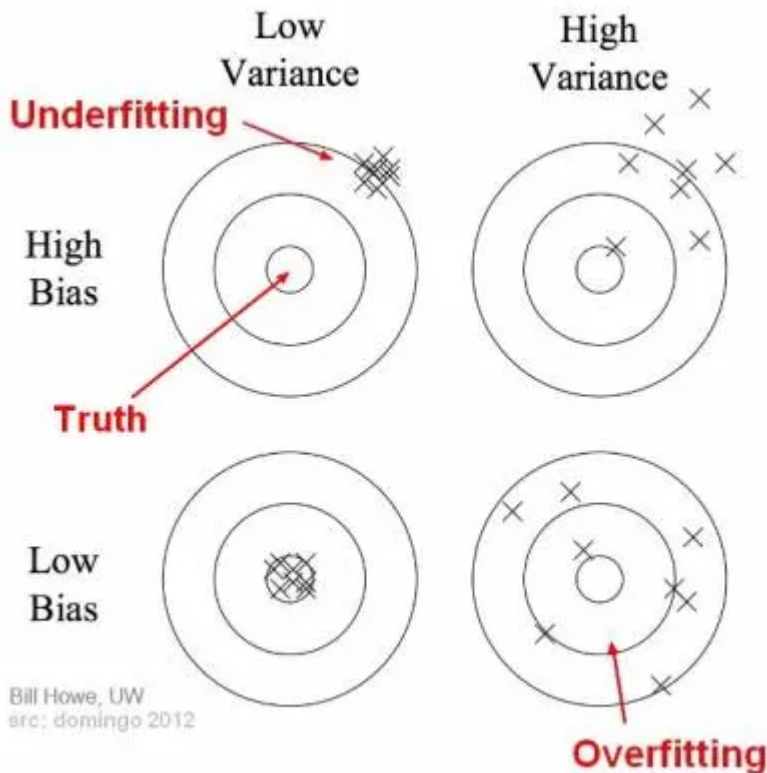
✓ What is bias?

Bias is the difference between the average prediction of our model and the correct value which we are trying to predict. Model with high bias pays very little attention to the training data and oversimplifies the model. It always leads to high error on training and test data.

✓ What is variance?

Variance is the variability of model prediction for a given data point or a value which tells us spread of our data. Model with high variance pays a lot of attention to training data and does not generalize on the data which it hasn't seen before. As a result, such models perform very well on training data but has high error rates on test data.

✓ Bias and variance using bulls-eye diagram



In the above diagram, center of the target is a model that perfectly predicts correct values. As we move away from the bulls-eye our predictions become get worse and worse. We can repeat our process of model building to get separate hits on the target.

✓ Underfitting Model

In supervised learning, underfitting happens when a model unable to capture the underlying pattern of the data. These models usually have high bias and low variance. It happens when we have very less amount of data to build an accurate model or when we try to build a linear model with a nonlinear data. Also, these kind of models are very simple to capture the complex patterns in data like Linear and logistic regression.

✓ Overfitting Model

Overfitting occurs when a model learns from noise or irrelevant patterns in the data, resulting in poor performance on unseen data.

Overfitting of the model occurs when the model learns just 'too-well' on the train data. This would sound like an advantage but it is not. When a model is overtrained on training data, it performs worst on the test data or any new data provided.

Technically, the model learns the details as well as the noise of the train data. This would hinder the performance of any new data provided to the model as the learned details and noise cannot be applied to the new data. This is the case when we say the performance of the model is not adequate.

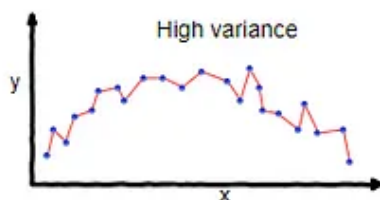
Model overfitting is a serious problem and can cause the model to produce misleading information.

A visual example of overfitting in regression:

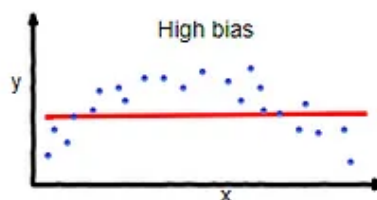
Below we see two scatter plots with the same data. We've chosen this to be a bit of an extreme example, just so you can visualize it.

On the left is a linear model for these points, and on the right is a model that fits the data pretty perfectly. The model on the right uses many more regression parameters and is overfit.

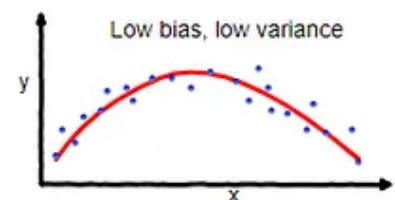
You can see why this model on the right looks great for this data set. But the only way it could work with another sample is if the points were in nearly the exact same places. It's too customized for the data in this sample.



overfitting



underfitting



Good balance

✓ Why is Bias Variance Tradeoff?

If our model is too simple and has very few parameters then it may have high bias and low variance. On the other hand if our model has large number of parameters then it's going to have high variance and low bias. So we need to find the right/good balance without overfitting and underfitting the data.

This tradeoff in complexity is why there is a tradeoff between bias and variance. An algorithm can't be more complex and less complex at the same time.

✓ Simulation for Bias Variance Tradeoff

```
# sequence: create polynomial feature, scale, train(f(t))

from sklearn.pipeline import make_pipeline # Importing make_pipeline function fr

degree = 30 # Define the degree of the polynomial features

# Creating a pipeline to streamline the process:
# 1. Generating polynomial features up to the specified degree
# 2. Scaling the features using StandardScaler
# 3. Fitting a linear regression model
polyreg_scaled = make_pipeline(PolynomialFeatures(degree), scaler, LinearRegressi

# Fitting the pipeline to the training data
polyreg_scaled.fit(X_train, y_train)

# Displaying the R^2 scores for the training and testing data to evaluate model p
display(polyreg_scaled.score(X_train, y_train)) # R^2 score for the training dat
display(polyreg_scaled.score(X_test, y_test))   # R^2 score for the testing data
```

⇒ 0.9389724819985867
0.8094469425937477

Above model is an overfitted model, because it is performing well on the training dataset but it is not performing well on the test dataset

```
# sequence: create polynomial feature, scale, train(f(t))

from sklearn.pipeline import make_pipeline # Importing make_pipeline function fr

degree = 5 # Define the degree of the polynomial features

# Creating a pipeline to streamline the process:
# 1. Generating polynomial features up to the specified degree
# 2. Scaling the features using StandardScaler
# 3. Fitting a linear regression model
polyreg_scaled = make_pipeline(PolynomialFeatures(degree), scaler, LinearRegressi

# Fitting the pipeline to the training data
polyreg_scaled.fit(X_train, y_train)

# Displaying the R^2 scores for the training and testing data to evaluate model p
display(polyreg_scaled.score(X_train, y_train)) # R^2 score for the training dat
display(polyreg_scaled.score(X_test, y_test))   # R^2 score for the testing data
```

⇒ 0.9214962587812688
0.8656262710391975

Above model is moderate fitted model.

```
from sklearn.pipeline import make_pipeline # Importing make_pipeline function fr

degree = 1 # Define the degree of the polynomial features

# Creating a pipeline to streamline the process:
# 1. Generating polynomial features up to the specified degree
# 2. Scaling the features using StandardScaler
# 3. Fitting a linear regression model
polyreg_scaled = make_pipeline(PolynomialFeatures(degree), scaler, LinearRegressi

# Fitting the pipeline to the training data
polyreg_scaled.fit(X_train, y_train)

# Displaying the R^2 scores for the training and testing data to evaluate model p
display(polyreg_scaled.score(X_train, y_train)) # R^2 score for the training dat
display(polyreg_scaled.score(X_test, y_test)) # R^2 score for the testing data

⇒ 0.8740205794555178
0.8073308228555811
```

Above model is underfitted model

```
from sklearn.preprocessing import PolynomialFeatures # Importing PolynomialFeatures
from sklearn.preprocessing import StandardScaler # Importing StandardScaler from
from sklearn.linear_model import LinearRegression # Importing LinearRegression f

degrees = 32 # Define the maximum degree of polynomial features to be considered

train_scores = [] # Initialize an empty list to store training R^2 scores for ea
test_scores = [] # Initialize an empty list to store testing R^2 scores for eac

# Loop over degrees of polynomial features from 1 to degrees-1
for degree in range(1, degrees):
    # Create a pipeline to streamline the process:
    # 1. Generating polynomial features up to the specified degree
    # 2. Scaling the features using StandardScaler
    # 3. Fitting a linear regression model
    polyreg_scaled = make_pipeline(PolynomialFeatures(degree), scaler, LinearRegr

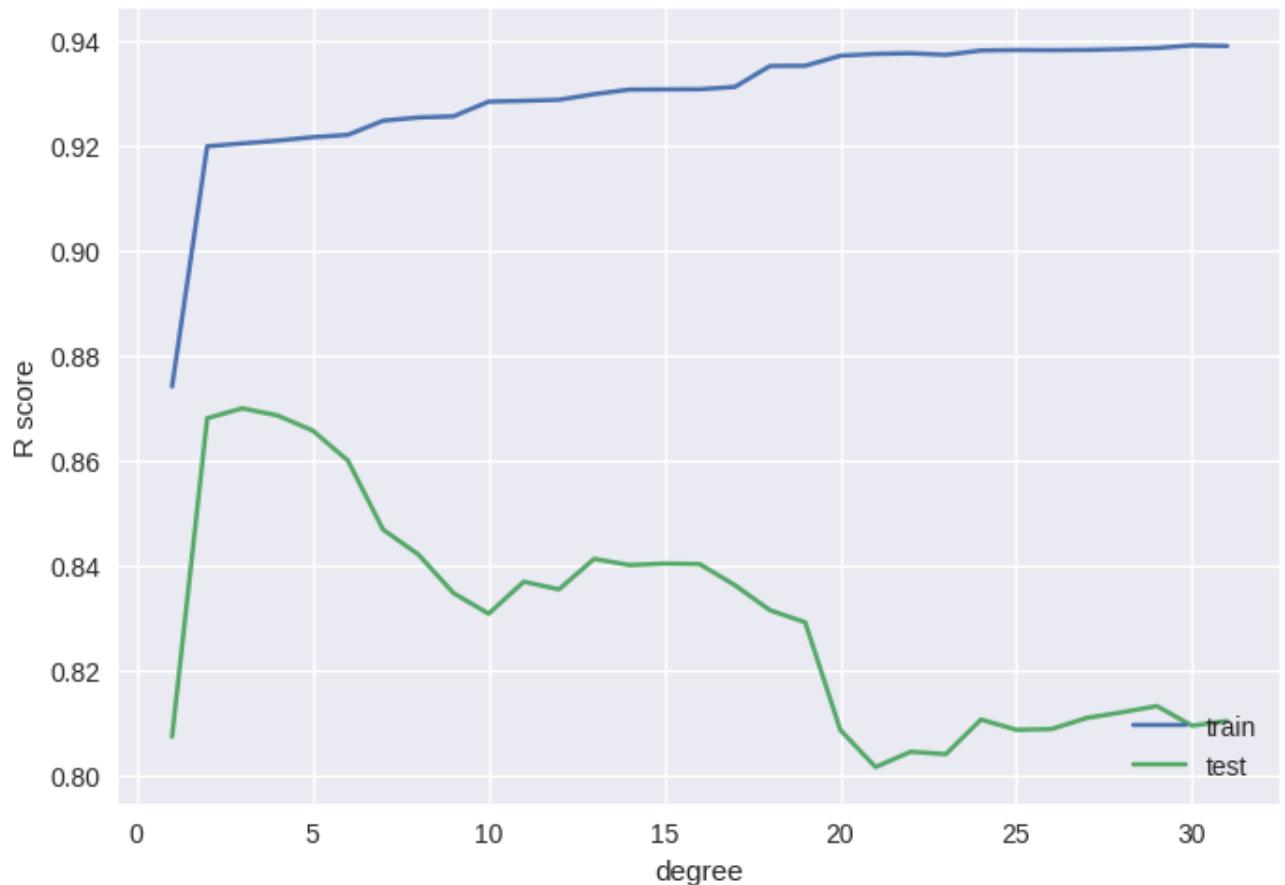
    # Fit the pipeline to the training data
    polyreg_scaled.fit(X_train, y_train)

    # Compute R^2 scores for training and testing data
    train_score = polyreg_scaled.score(X_train, y_train)
    test_score = polyreg_scaled.score(X_test, y_test)

    # Append the R^2 scores to the respective lists
    train_scores.append(train_score)
    test_scores.append(test_score)

plt.figure() # Create a new figure
plt.plot(list(range(1, 32)), train_scores, label="train") # Plot training R^2 sc
plt.plot(list(range(1, 32)), test_scores, label="test") # Plot testing R^2 scor

plt.legend(loc="lower right") # Add a legend to the plot
plt.xlabel("degree") # Label for x-axis
plt.ylabel("R score") # Label for y-axis
plt.show() # Display the plot
```

- The plot displays training (blue) and testing (orange) performance (R-squared) for different polynomial degrees.
- Initially, as the polynomial degree increases, the training performance (R-squared) tends to improve steadily. This suggests that higher polynomial degrees allow the model to capture more complex patterns within the training data, leading to better fitting.
- However, beyond a certain point, typically around degree 5 in this case, the testing performance begins to decline despite the continued increase in the polynomial degree. This decline indicates that the model's ability to generalize to unseen data diminishes as the complexity of the model increases. In other words, the model starts to overfit the training data, resulting in poor performance on new, unseen data.

✓ Validation in Model Training

In the process of training machine learning models, ensuring that the models generalize well is paramount. One crucial concept in achieving this is validation.

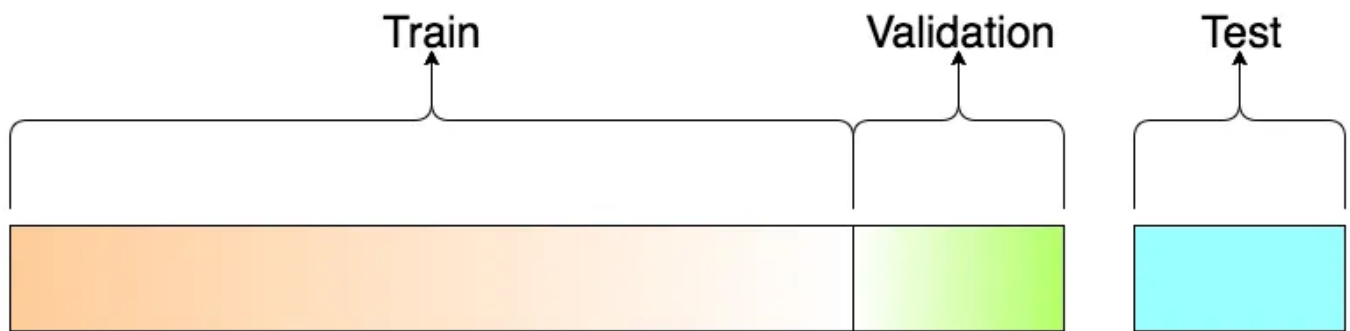
Problem with Direct Testing

Traditionally, a dataset is split into a training set and a test set. However, directly using the test set to evaluate and adjust the model during training can lead to biased results. This approach

forces the model to perform well on the test set, potentially compromising its generalization capability.

✓ Introduction of Validation Layer

To address above issue, an additional middle layer, known as the validation layer, is introduced. This layer acts as a buffer between the training and test sets, allowing for more unbiased model evaluation.



The validation dataset serves as unseen data during model training. By assessing the model's performance on the validation set, adjustments can be made to improve its generalization without biasing it towards the test set.

✓ Validation Techniques

✓ Basic Validation Split

Initially, the dataset is divided into training and test sets, typically in an 80-20 split. However, this approach can be problematic for smaller datasets, where insufficient training data may lead to biased models.

✓ Cross-Validation

To overcome the limitations of basic validation splits, techniques like k-fold cross-validation are employed. In k-fold cross-validation, the dataset is divided into k subsets (or "folds"). The model is trained k times, each time using a different fold as the validation set and the remaining data for training.

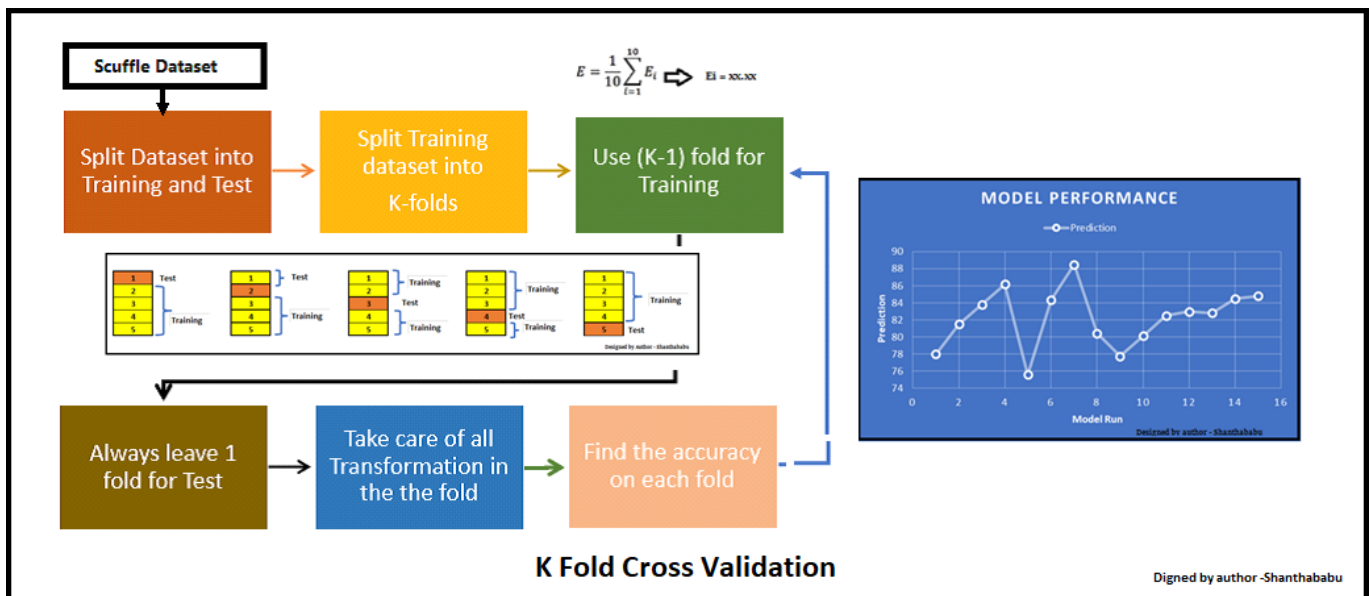
Advantages of Cross-Validation

- Provides a more robust estimate of model performance.
- Helps assess bias and variance by analyzing performance across multiple validation sets.

- Ensures consistent model performance across different training-validation splits, indicating good generalization.

✓ Implementation of K-Fold Cross-Validation

1. Data Splitting: The dataset is divided into training and test sets, with a portion reserved for validation
2. Fold Creation: The training set is further divided into k folds.
3. Model Training and Evaluation: The model is trained k times, each time using a different combination of folds for training and validation.
4. Performance Assessment: Model performance metrics, such as R-squared, are computed for each fold.
5. Generalization Evaluation: Consistency in performance across folds indicates good generalization.



```
from sklearn.model_selection import KFold
kf = KFold(n_splits=10, shuffle=True, random_state=1)

from sklearn.preprocessing import PolynomialFeatures
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression

degrees = 32 # number of data points

train_scores = []
test_scores = []

for degree in range(1, degrees):
    fold_train_scores = []
    fold_test_scores = []

    for train_index, test_index in kf.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        polyreg_scaled = make_pipeline(PolynomialFeatures(degree), scaler, LinearRegr
        polyreg_scaled.fit(X_train, y_train)

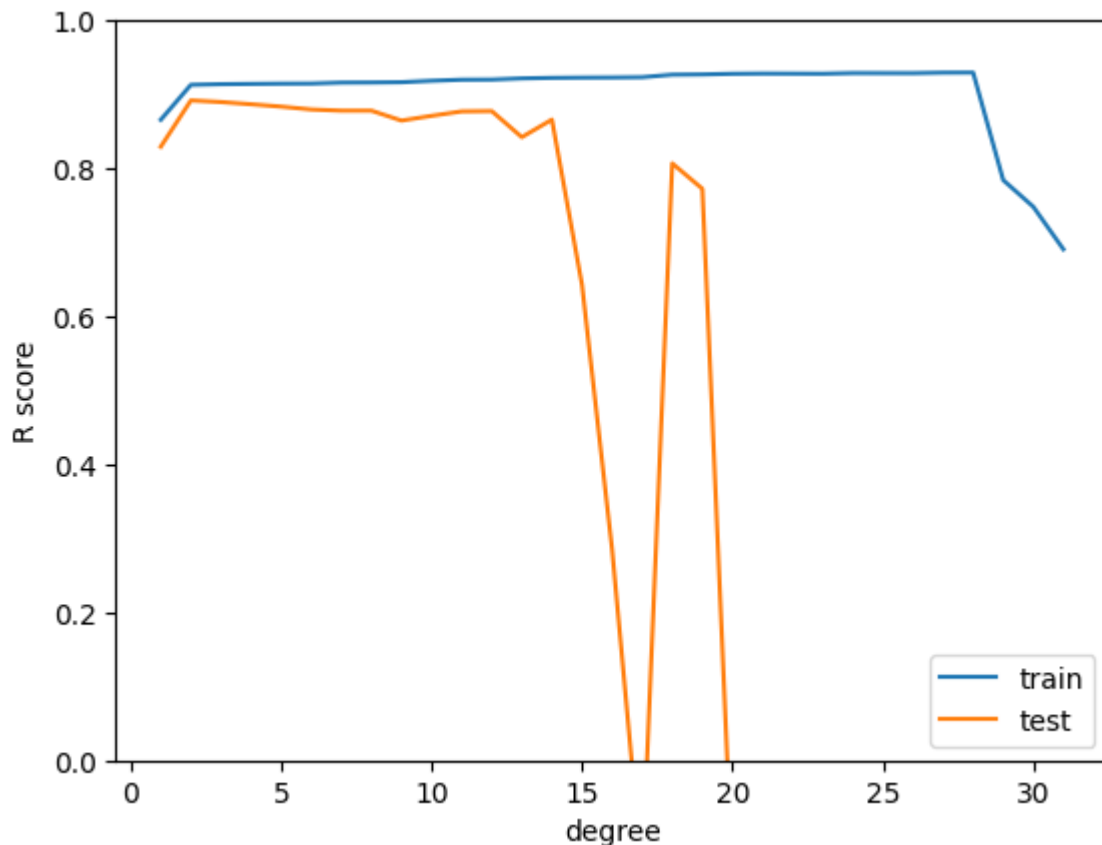
        train_score = polyreg_scaled.score(X_train, y_train)
        test_score = polyreg_scaled.score(X_test, y_test)

        fold_train_scores.append(train_score)
        fold_test_scores.append(test_score)

    train_score = np.mean(fold_train_scores)
    test_score = np.mean(fold_test_scores)
    train_scores.append(train_score)
    test_scores.append(test_score)

plt.figure()
plt.plot(list(range(1, 32)), train_scores, label="train")
plt.plot(list(range(1, 32)), test_scores, label="test")

plt.legend(loc="lower right")
plt.ylim(0, 1)
plt.xlabel("degree")
plt.ylabel("R score")
plt.show()
```



- Initially, as the degree of polynomial increases, both the training and testing scores tend to improve.
- This improvement is expected, as higher-degree polynomials can capture more complex relationships in the data.
- However, beyond a certain point, increasing the degree of polynomial may lead to overfitting, where the model learns noise in the data rather than the underlying pattern.
- This is indicated by a widening gap between the training and testing scores.
- The optimal degree of polynomial is the point where the testing score is maximized without a significant increase in the gap between training and testing scores.
- This point represents the best balance between bias and variance:
 - If the degree of polynomial is too low, the model may suffer from high bias, meaning it oversimplifies the underlying pattern in the data.
 - If the degree of polynomial is too high, the model may suffer from high variance, meaning it fits the noise in the data rather than the underlying pattern, leading to poor generalization to new data.

✓ Controlling Overfitting

✓ Introduction to Regularization

Regularization is a fundamental approach in machine learning to mitigate overfitting, particularly advantageous for linear models. By introducing a penalty term to the model's objective function, regularization aims to balance the trade-off between bias and variance.

Penalizing Coefficients:

- In regularization, the coefficients of the model are penalized to prevent them from growing too large. This penalty term discourages complex models with overly large coefficients, which are prone to overfitting. By imposing constraints on the coefficients, regularization promotes simpler models that generalize better to unseen data.

Reasons for Overfitting:

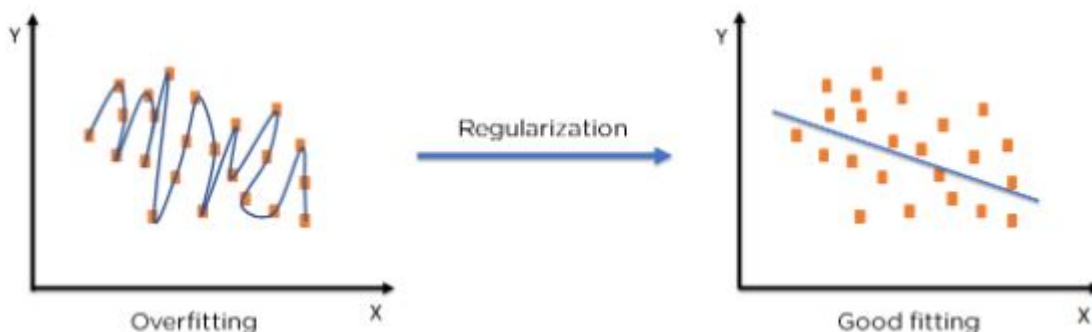
1. Unnecessary Coefficients:

- Overfitting can occur when the model includes unnecessary coefficients that do not contribute significantly to predicting the target variable. These unnecessary coefficients lead to the model capturing noise or irrelevant patterns in the training data, resulting in poor generalization performance.

2. Magnitude of Coefficients:

- Another cause of overfitting is the excessively high magnitude of coefficients. When certain features have disproportionately large coefficients, the model becomes overly sensitive to variations in those features, leading to overfitting. Regularization helps address this issue by penalizing large coefficient values, encouraging the model to prioritize simpler, more parsimonious solutions.

Regularization techniques play a crucial role in controlling overfitting by penalizing unnecessary or excessively large coefficients in the model. By promoting simplicity and generalization, regularization helps improve the robustness and performance of machine learning models, especially in the context of linear regression.



✓ Types of Regularization

1. Lasso (L1 Regularization):

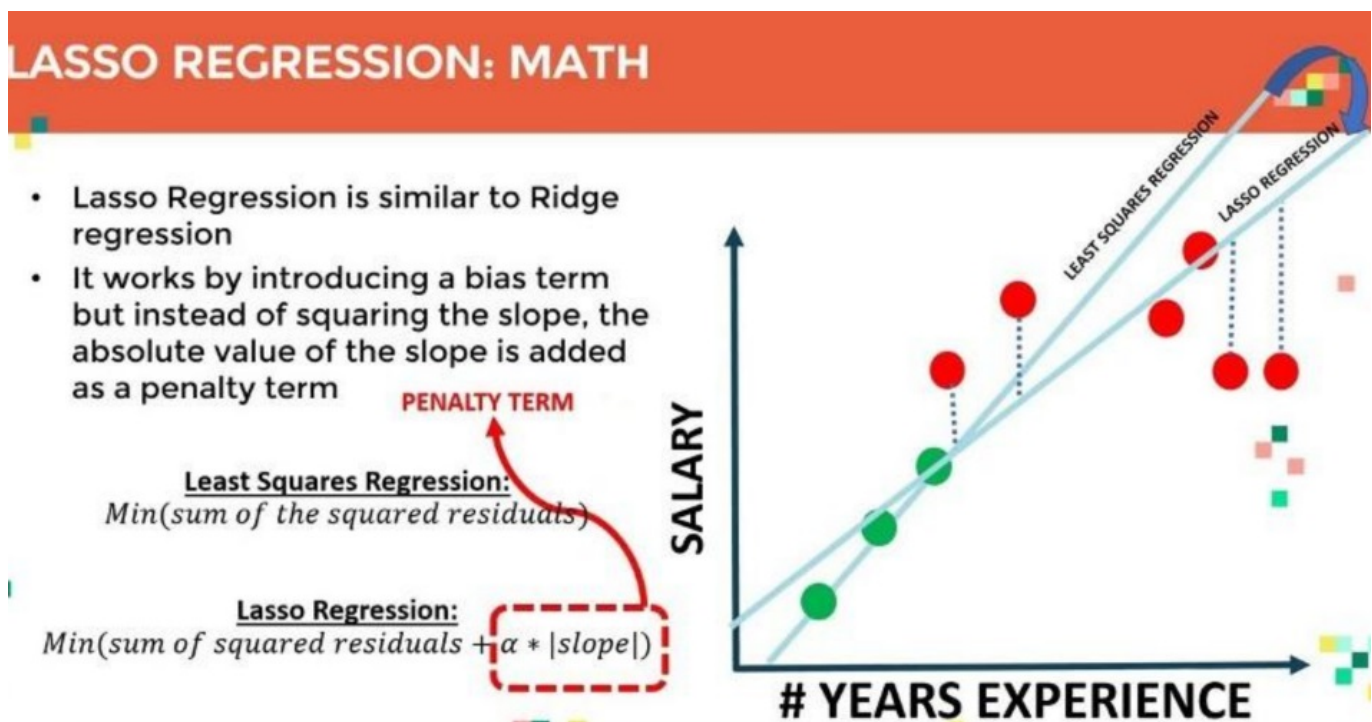
Modifies overfitted or under-fitted models by adding a penalty equivalent to the sum of the absolute values of the coefficients.

Lasso regression also performs coefficient minimization, but instead of squaring the magnitudes of the coefficients, it takes the actual values of the coefficients. This means that the sum of the coefficients can also be 0 because there are negative coefficients. Consider the cost function for the lasso regression.

Formula: The cost function with Lasso regularization is modified by adding a penalty term that is the sum of the absolute values of the coefficients multiplied by a regularization parameter λ :

$$\text{Cost Function (Lasso)} = \text{Original Cost Function} + \lambda * \sum |\theta_i|$$

Here, θ_i represents the coefficients of the model, and λ is the regularization parameter that controls the strength of regularization.



2. Ridge (L2 Regularization):

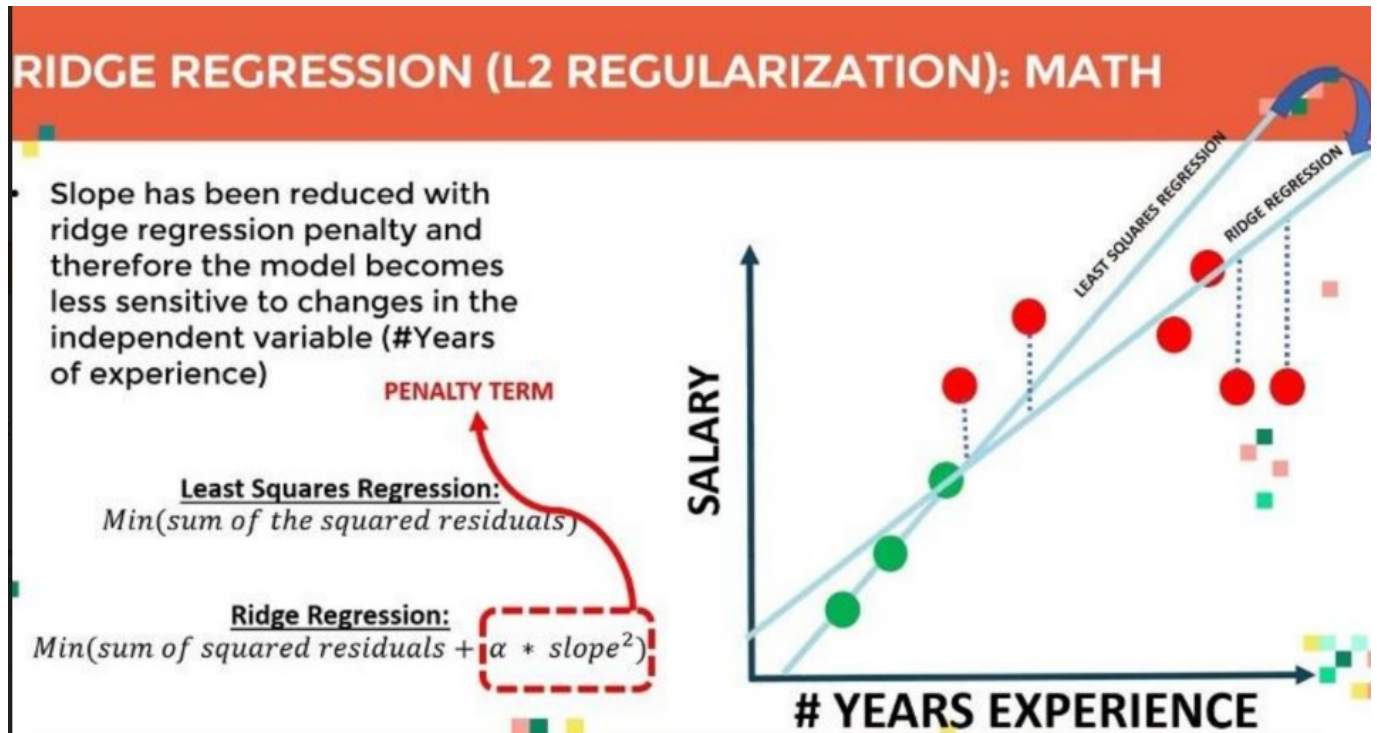
Also known as Ridge Regression, it adjusts models with overfitting or underfitting by adding a penalty equivalent to the sum of the squares of the magnitudes of the coefficients.

This means that the mathematical function representing our machine learning model is minimized and the coefficients are calculated. The size of the coefficients is multiplied and added. Ridge Regression performs regularization by reducing the coefficients present. The function shown below shows the cost function of the ridge regression.

Formula: The cost function with Ridge regularization is modified by adding a penalty term that is the sum of the squared values of the coefficients multiplied by a regularization parameter λ :

$$\text{Cost Function (Ridge)} = \text{Original Cost Function} + \lambda * \sum (\theta_i)^2$$

Again, θ_i represents the coefficients of the model, and λ controls the strength of regularization. Consider the graph illustrated below which represents Linear regression:



3. Elastic Net:

Elastic Net regularization is a hybrid technique that combines both Lasso and Ridge regularization. Formula: The cost function with Elastic Net regularization includes both the L1 and L2 penalty terms:

Cost Function (Elastic Net) = Original Cost Function + $\lambda_1 * \sum |\theta_i|$ + $\lambda_2 * \sum (\theta_i)^2$

Here, λ_1 and λ_2 are regularization parameters for L1 and L2 regularization, respectively.

- Elastic net combines L1 and L2 with the addition of an alpha parameter deciding the ratio between them:

$$\frac{\sum_{i=1}^n (y_i - x_i^T \hat{\beta})^2}{2n} + \lambda \left(\frac{1 - \alpha}{2} \sum_{j=1}^m \hat{\beta}_j^2 + \alpha \sum_{j=1}^m |\hat{\beta}_j| \right)$$

These regularization techniques offer flexible tools for controlling model complexity and improving generalization performance in machine learning models. The choice between Lasso, Ridge, or Elastic Net depends on the specific characteristics of the dataset and the desired balance between feature selection and coefficient shrinkage.

The shrinkage parameter, denoted as λ , determines the strength of the regularization. λ controls how strongly the coefficients are penalized in the regularization process.

✓ Linear Regression Simulation