**Model Development**

In this module, you will learn how to define the explanatory variable and the response variable and understand the differences between the simple linear regression and multiple linear regression models. You will learn how to evaluate a model using visualization and learn about polynomial regression and pipelines. You will also learn how to interpret and use the R-squared and the mean square error measures to perform in-sample evaluations to numerically evaluate our model. And lastly, you will learn about prediction and decision making when determining if our model is correct.

**Learning Objectives**

* Evaluate model using visualizationtechniques in Python
* Apply polynomial regression techniques using Python
* Transform data into a polynomial, then use linear regression to fit the parameter
* Apply model evaluation using visualization in Python
* Apply polynomial regression techniques to Python
* Predict and make decisions based on to Python data models
* Describe the use of R-squared and MSE for in-sample evaluation
* Define the term "curvilinear relationship"

# **Model Development**

## **Model Development**

model development and predicting car prices:

* **Model Development**: Involves creating a mathematical equation to predict a value based on one or more independent variables (features).
* **Independent and Dependent Variables**:
  + **Independent Variable**: Example - Car's highway miles per gallon.
  + **Dependent Variable**: Example - Price of the car.
* **Importance of Data**:
  + More relevant data leads to more accurate models.
  + Example: If color is not included as a feature, the model may inaccurately predict prices for similar cars of different colors.
* **Types of Regression**:
  + **Simple Linear Regression**: One independent variable.
  + **Multiple Linear Regression**: Multiple independent variables.
  + **Polynomial Regression**: Models relationships that are not linear.
* **Model Evaluation**: Techniques like R-squared and Mean Squared Error (MSE) are used for evaluating model performance.

## **Linear Regression and Multiple Linear Regression**

1. **Simple Linear Regression (SLR)**:
   * **Definition**: A method to understand the relationship between two variables: one independent variable (predictor) and one dependent variable (target).
   * **Equation**: The relationship can be expressed as: [ y = b\_0 + b\_1 \cdot x ]
     + (b\_0): Intercept
     + (b\_1): Slope
2. **Prediction**:
   * You can predict the target variable using the model. For example, if you know the highway miles per gallon (20), you can predict the price of a car.
3. **Training the Model**:
   * You fit the model using training data points to determine the parameters (b\_0) and (b\_1).
4. **Noise**:
   * Real-world data often contains noise, which is a small random value added to the predictions. This accounts for variability not explained by the model.
5. **Multiple Linear Regression**:
   * **Definition**: Extends SLR to include multiple independent variables.
   * **Equation**: For multiple predictors: [ y = b\_0 + b\_1 \cdot x\_1 + b\_2 \cdot x\_2 + ... + b\_n \cdot x\_n ]
   * Each (b\_i) represents the coefficient for the corresponding predictor.

Code Example for Simple Linear Regression

Here’s how you can implement simple linear regression using Python's scikit-learn library:

import numpy as np

import pandas as pd

from sklearn.linear\_model import LinearRegression

# Sample data

data = {

'miles\_per\_gallon': [10, 15, 20, 25, 30],

'price': [30000, 25000, 22000, 20000, 18000]

}

# Create DataFrame

df = pd.DataFrame(data)

# Define predictor and target variables

X = df[['miles\_per\_gallon']] # Predictor

y = df['price'] # Target

# Create linear regression model

model = LinearRegression()

# Fit the model

model.fit(X, y)

# Make a prediction

predicted\_price = model.predict([[20]])

print(f"Predicted price for 20 MPG: ${predicted\_price[0]:.2f}")

# Model parameters

intercept = model.intercept\_

slope = model.coef\_[0]

print(f"Intercept (b0): {intercept}, Slope (b1): {slope}")

Explanation of the Code:

* **Data Preparation**: We create a DataFrame with sample data for miles per gallon and price.
* **Model Creation**: We instantiate a LinearRegression model.
* **Fitting the Model**: The fit method trains the model using the predictor and target variables.
* **Prediction**: We use the predict method to estimate the price for a car with 20 MPG.
* **Model Parameters**: The intercept and slope are printed to understand the linear relationship.

Code Example for Multiple Linear Regression

Here’s how to implement multiple linear regression:

# Sample data with multiple predictors

data = {

'miles\_per\_gallon': [10, 15, 20, 25, 30],

'age': [5, 3, 2, 1, 0],

'price': [30000, 25000, 22000, 20000, 18000]

}

# Create DataFrame

df = pd.DataFrame(data)

# Define predictor variables and target variable

X = df[['miles\_per\_gallon', 'age']] # Multiple predictors

y = df['price'] # Target

# Create linear regression model

model = LinearRegression()

# Fit the model

model.fit(X, y)

# Make a prediction

predicted\_price = model.predict([[20, 2]]) # 20 MPG, 2 years old

print(f"Predicted price for 20 MPG and 2 years old: ${predicted\_price[0]:.2f}")

# Model parameters

intercept = model.intercept\_

coefficients = model.coef\_

print(f"Intercept (b0): {intercept}, Coefficients (b1, b2): {coefficients}")

Explanation of the Code:

* **Data Preparation**: We create a DataFrame with multiple predictors (miles per gallon and age).
* **Model Creation**: Similar to SLR, we instantiate a LinearRegression model.
* **Fitting the Model**: The model is trained using both predictors.
* **Prediction**: We predict the price for a car with 20 MPG and 2 years old.
* **Model Parameters**: The intercept and coefficients for each predictor are printed.

Additional Concepts to Consider:

* **Assumptions of Linear Regression**:
  + Linearity: The relationship between predictors and the target should be linear.
  + Independence: Observations should be independent of each other.
  + Homoscedasticity: Constant variance of errors.
  + Normality: The residuals should be normally distributed.
* **Model Evaluation**:
  + Use metrics like R-squared, Mean Absolute Error (MAE), and Mean Squared Error (MSE) to evaluate model performance.

## **Model Evaluation using Visualization**

1. **Regression Plots**:
   * **Purpose**: To estimate the relationship between two variables (independent and dependent).
   * **Axes**:
     + **Horizontal Axis**: Independent variable (feature).
     + **Vertical Axis**: Dependent variable (target).
   * **Fitted Line**: Represents the predicted values based on the regression model.
2. **Creating a Regression Plot**:
   * You can use the regplot function from the Seaborn library.
   * **Code Example**:
   * import seaborn as sns
   * import pandas as pd
   * # Sample DataFrame
   * data = pd.DataFrame({
   * 'independent\_variable': [1, 2, 3, 4, 5],
   * 'dependent\_variable': [2, 3, 5, 7, 11]
   * })
   * # Creating a regression plot

sns.regplot(x='independent\_variable', y='dependent\_variable', data=data)

1. **Residual Plots**:
   * **Purpose**: To visualize the error between actual and predicted values.
   * **Interpretation**:
     + Ideally, residuals should be randomly distributed around zero.
     + Patterns in residuals can indicate model inadequacies (e.g., non-linearity).
2. **Creating a Residual Plot**:
   * Use the residplot function from Seaborn.
   * **Code Example**:
   * # Creating a residual plot

sns.residplot(x='independent\_variable', y='dependent\_variable', data=data)

1. **Distribution Plots**:
   * **Purpose**: To visualize the distribution of predicted vs. actual values.
   * **Use Case**: Useful for models with multiple independent variables.
   * **Code Example**:
   * # Creating a distribution plot
   * sns.kdeplot(data['dependent\_variable'], color='red', label='Actual', fill=True)

sns.kdeplot(data['predicted\_values'], color='blue', label='Predicted', fill=True)

Additional Concepts

* **Model Evaluation**: It's crucial to evaluate how well your model performs. This can be done using metrics like Mean Absolute Error (MAE), Mean Squared Error (MSE), and R-squared values.
* **Assumptions of Linear Regression**:
  + **Linearity**: The relationship between independent and dependent variables should be linear.
  + **Homoscedasticity**: The residuals should have constant variance.
  + **Independence**: Observations should be independent of each other.
  + **Normality**: Residuals should be normally distributed.

Summary

* **Regression plots** help visualize relationships between variables.
* **Residual plots** are essential for diagnosing model fit.
* **Distribution plots** provide insights into the accuracy of predictions.
* Understanding the **assumptions of linear regression** is vital for effective model evaluation.

## **Polynomial Regression and Pipelines**

1. **Polynomial Regression**:
   * **Definition**: A type of regression analysis where the relationship between the independent variable (x) and the dependent variable (y) is modeled as an (n)th degree polynomial.
   * **Curvilinear Relationships**: Polynomial regression is useful for modeling relationships that are not linear, such as quadratic or cubic relationships.
2. **Types of Polynomial Regression**:
   * **Quadratic Regression**: A second-order polynomial regression where the predictor variable is squared.
   * **Cubic Regression**: A third-order polynomial regression where the predictor variable is cubed.
   * Higher-order polynomials can be used if lower orders do not provide a good fit.
3. **Using Python for Polynomial Regression**:
   * The numpy.polyfit function is commonly used to fit polynomial regression models.

Code Example for Polynomial Regression

Here’s a simple example of how to implement a cubic polynomial regression using Python:

import numpy as np

import matplotlib.pyplot as plt

# Sample data

x = np.array([1, 2, 3, 4, 5])

y = np.array([2.2, 3.8, 7.1, 10.5, 15.2])

# Fit a cubic polynomial regression model

coefficients = np.polyfit(x, y, 3) # 3 indicates cubic

polynomial = np.poly1d(coefficients)

# Generate x values for plotting

x\_fit = np.linspace(1, 5, 100)

y\_fit = polynomial(x\_fit)

# Plotting the data and the polynomial fit

plt.scatter(x, y, color='red', label='Data Points')

plt.plot(x\_fit, y\_fit, label='Cubic Polynomial Fit', color='blue')

plt.legend()

plt.xlabel('X')

plt.ylabel('Y')

plt.title('Cubic Polynomial Regression')

plt.show()

Explanation of the Code:

* **Data Preparation**: We create sample data points for (x) and (y).
* **Fitting the Model**: np.polyfit(x, y, 3) fits a cubic polynomial to the data. The result is an array of coefficients.
* **Creating the Polynomial**: np.poly1d(coefficients) creates a polynomial function that can be used to compute (y) values for any (x).
* **Plotting**: We visualize the original data points and the fitted polynomial curve.

1. **Pipelines**:
   * **Definition**: A way to streamline the process of applying multiple transformations and modeling steps in machine learning.
   * **Benefits**: Simplifies code, reduces the risk of data leakage, and makes it easier to manage complex workflows.

Code Example for Pipelines

Here’s how to create a pipeline for polynomial regression and normalization:

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import PolynomialFeatures, StandardScaler

from sklearn.linear\_model import LinearRegression

# Sample data

X = np.array([[1], [2], [3], [4], [5]])

y = np.array([2.2, 3.8, 7.1, 10.5, 15.2])

# Create a pipeline

pipeline = Pipeline([

('poly\_features', PolynomialFeatures(degree=3)), # Cubic features

('scaler', StandardScaler()), # Normalize features

('lin\_reg', LinearRegression()) # Linear regression model

])

# Fit the pipeline

pipeline.fit(X, y)

# Make predictions

predictions = pipeline.predict(X)

print(predictions)

Explanation of the Pipeline Code:

* **Pipeline Creation**: We create a pipeline that first transforms the features into polynomial features, then normalizes them, and finally fits a linear regression model.
* **Fitting the Pipeline**: The fit method applies all transformations and fits the model in one step.
* **Making Predictions**: The predict method uses the fitted pipeline to make predictions on the input data.

Additional Concepts:

* **Normalization**: Important for ensuring that features contribute equally to the model. The StandardScaler standardizes features by removing the mean and scaling to unit variance.
* **Model Evaluation**: After fitting the model, it's crucial to evaluate its performance using metrics like R-squared, Mean Squared Error (MSE), etc.

## **Measures for In-Sample Evaluation**

1. **Mean Squared Error (MSE)**:
   * **Definition**: MSE measures the average of the squares of the errors—that is, the average squared difference between the actual and predicted values.
   * **Formula**: [ MSE = \frac{1}{n} \sum\_{i=1}^{n} (y\_i - \hat{y}\_i)^2 ] where (y\_i) is the actual value, (\hat{y}\_i) is the predicted value, and (n) is the number of samples.
   * **Python Code Example**:
   * from sklearn.metrics import mean\_squared\_error
   * # Actual and predicted values
   * actual = [150, 200, 250]
   * predicted = [50, 150, 300]
   * # Calculate MSE
   * mse = mean\_squared\_error(actual, predicted)

print("Mean Squared Error:", mse)

* + **Explanation**: In this code, we import the mean\_squared\_error function from sklearn.metrics. We define the actual and predicted values, then calculate the MSE using the function.

1. **R-squared (Coefficient of Determination)**:
   * **Definition**: R-squared indicates how well the independent variable(s) explain the variability of the dependent variable. It ranges from 0 to 1, where 1 indicates a perfect fit.
   * **Formula**: [ R^2 = 1 - \frac{MSE*{regression}}{MSE*{mean}} ] where (MSE*{regression}) is the MSE of the regression model and (MSE*{mean}) is the MSE of the mean of the data points.
   * **Python Code Example**:
   * from sklearn.linear\_model import LinearRegression
   * from sklearn.metrics import r2\_score
   * import numpy as np
   * # Sample data
   * X = np.array([[1], [2], [3], [4], [5]])
   * y = np.array([1, 2, 3, 4, 5])
   * # Create and fit the model
   * model = LinearRegression()
   * model.fit(X, y)
   * # Predict values
   * y\_pred = model.predict(X)
   * # Calculate R-squared
   * r\_squared = r2\_score(y, y\_pred)

print("R-squared:", r\_squared)

* + **Explanation**: Here, we create a simple linear regression model using LinearRegression from sklearn. After fitting the model to the data, we predict the values and calculate R-squared using the r2\_score function.

Additional Relevant Concepts

* **Overfitting**: This occurs when a model learns the noise in the training data instead of the actual pattern. It can lead to a high R-squared value on training data but poor performance on unseen data. This is important to consider when evaluating model performance.
* **Model Evaluation**: Besides MSE and R-squared, other metrics like **Mean Absolute Error (MAE)** and **Root Mean Squared Error (RMSE)** can also be useful for evaluating model performance.
* **Data Visualization**: Visualizing the residuals (the differences between actual and predicted values) can help identify patterns that indicate model fit issues.

Conclusion

Understanding MSE and R-squared is crucial for evaluating the performance of regression models. The provided code examples illustrate how to calculate these metrics in Python, enhancing your practical skills.

## **Prediction and Decision Making**

1. **Model Validation**:
   * Ensure that model results make sense through **visualization**, **numerical measures**, and **model comparison**.
2. **Prediction Example**:
   * Using a trained model to predict the price of a car based on its highway miles per gallon (MPG).
   * Example prediction: For a car with 30 MPG, the predicted price is **$13,771.30**.
3. **Coefficients Interpretation**:
   * The coefficient indicates how much the price changes with a one-unit increase in MPG.
   * Example: An increase of 1 unit in MPG results in a decrease of approximately **$821** in price.
4. **Model Limitations**:
   * If predictions yield unrealistic values (e.g., negative prices), it may indicate issues with the model or data.
5. **Generating Sequences**:
   * Use **NumPy** to create sequences for predictions.
6. import numpy as np

mpg\_values = np.arange(1, 101, 1) # Generates values from 1 to 100

1. **Visualization**:
   * Use regression plots to visualize data and model fit.
   * Residual plots can indicate non-linear behavior.
2. **Mean Square Error (MSE)**:
   * A common metric for evaluating model performance. Lower MSE indicates a better fit.
   * Example of MSE values:
     + MSE = 3,495 (good fit)
     + MSE = 12,870 (poor fit)
3. **R-squared Value**:
   * Indicates the proportion of variance explained by the model.
   * Values closer to 1 suggest a better fit.
   * Example R-squared values:
     + 0.9986 (excellent fit)
     + 0.806 (moderate fit)

Additional Concepts

* **Multiple Linear Regression (MLR)** vs. **Simple Linear Regression (SLR)**:
  + MLR includes multiple predictors, which can lead to a lower MSE compared to SLR.
* **Polynomial Regression**:
  + Useful for capturing non-linear relationships in data. It can also yield a lower MSE than linear regression.

Code Example for Model Evaluation

Here’s a simple code snippet to illustrate how to evaluate a linear regression model using MSE and R-squared:

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error, r2\_score

import numpy as np

# Sample data

X = np.array([[1], [2], [3], [4], [5]]) # Predictor (MPG)

y = np.array([20000, 18000, 16000, 14000, 12000]) # Target (Price)

# Create and fit the model

model = LinearRegression()

model.fit(X, y)

# Make predictions

predictions = model.predict(X)

# Calculate MSE and R-squared

mse = mean\_squared\_error(y, predictions)

r\_squared = r2\_score(y, predictions)

print(f'Mean Squared Error: {mse}')

print(f'R-squared: {r\_squared}')

Explanation of the Code:

* **LinearRegression**: A class from sklearn to create a linear regression model.
* **fit()**: Trains the model using the predictor and target data.
* **predict()**: Generates predictions based on the input data.
* **mean\_squared\_error()**: Computes the MSE between actual and predicted values.
* **r2\_score()**: Calculates the R-squared value to assess model fit.

**Lesson Summary**

Congratulations! You have completed this lesson. At this point in the course, you know:

* Linear regression refers to using one independent variable to make a prediction.
* You can use multiple linear regression to explain the relationship between one continuous target y variable and two or more predictor x variables.
* Simple linear regression, or SLR, is a method used to understand the relationship between two variables, the predictor independent *variable x* and the target dependent *variable y*.
* Use the **regplot** and **residplot** functions in the Seaborn library to create regression and residual plots, which help you identify the strength, direction, and linearity of the relationship between your independent and dependent variables.
* When using residual plots for model evaluation, residuals should ideally have zero mean, appear evenly distributed around the x-axis, and have consistent variance. If these conditions are not met, consider adjusting your model.
* Use distribution plots for models with multiple features: Learn to construct distribution plots to compare predicted and actual values, particularly when your model includes more than one independent variable. Know that this can offer deeper insights into the accuracy of your model across different ranges of values.
* The order of the polynomials affects the fit of the model to your data. Apply Python's **polyfit** function to develop polynomial regression models that suit your specific dataset.
* To prepare your data for more accurate modeling, use feature transformation techniques, particularly using the **preprocessing** library in scikit-learn, transform your data using polynomial features, and use the modules like **StandardScaler** to normalize the data.
* Pipelines allow you to simplify how you perform transformations and predictions sequentially, and you can use pipelines in scikit-learn to streamline your modeling process.
* You can construct and train a pipeline to automate tasks such as normalization, polynomial transformation, and making predictions.
* To determine the fit of your model, you can perform sample evaluations by using the Mean Square Error (MSE), using Python’s **mean\_squared\_error** function from scikit-learn, and using the score method to obtain the R-squared value.
* A model with a high R-squared value close to 1 and a low MSE is generally a good fit, whereas a model with a low R-squared and a high MSE may not be useful.
* Be alert to situations where your R-squared value might be negative, which can indicate overfitting.
* When evaluating models, use visualization and numerical measures and compare different models.
* The mean square error is perhaps the most intuitive numerical measure for determining whether a model is good.
* A distribution plot is a suitable method for multiple linear regression.
* An acceptable r-squared value depends on what you are studying and your use case.
* To evaluate your model’s fit, apply visualization, methods like regression and residual plots, and numerical measures such as the model's coefficients for sensibility:
* Use Mean Square Error (MSE) to measure the average of the squares of the errors between actual and predicted values and examine R-squared to understand the proportion of the variance in the dependent variable that is predictable from the independent variables.
* When analyzing residual plots, residuals should be randomly distributed around zero for a good model. In contrast, a residual plot curve or inaccuracies in certain ranges suggest non-linear behavior or the need for more data.