**Model Evaluation and Refinement**

# **Model Evaluation and Refinement**

1. **Model Evaluation**:
   * **In-sample Evaluation**: Measures how well the model fits the training data.
   * **Out-of-sample Evaluation**: Assesses how well the model predicts new, unseen data.
2. **Data Splitting**:
   * Split the dataset into **training** and **testing** sets.
   * Common split: 70% training, 30% testing.
3. **Generalization Error**:
   * Indicates how well the model performs on unseen data.
   * The testing data provides an approximation of this error.
4. **Cross-Validation**:
   * A technique to improve model evaluation by splitting the dataset into **k** equal groups (folds).
   * Each fold is used for testing while the others are used for training.

Code Example: Using train\_test\_split

Here's how you can use the train\_test\_split function from the Scikit-learn library:

from sklearn.model\_selection import train\_test\_split

# Example dataset

X = [[1], [2], [3], [4], [5], [6], [7], [8], [9], [10]] # Predictor variables

y = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] # Target variable

# Splitting the dataset

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

print("Training data:", X\_train)

print("Testing data:", X\_test)

**Explanation**:

* X represents the predictor variables, and y is the target variable.
* train\_test\_split splits the data into training and testing sets.
* test\_size=0.3 indicates that 30% of the data will be used for testing.
* random\_state=42 ensures reproducibility of the split.

Code Example: Cross-Validation

Here's how to implement cross-validation using cross\_val\_score:

from sklearn.model\_selection import cross\_val\_score

from sklearn.linear\_model import LinearRegression

import numpy as np

# Example dataset

X = [[1], [2], [3], [4], [5], [6], [7], [8], [9], [10]]

y = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Initialize the model

model = LinearRegression()

# Perform cross-validation

scores = cross\_val\_score(model, X, y, cv=3) # 3-fold cross-validation

print("Cross-validation scores:", scores)

print("Average score:", np.mean(scores))

**Explanation**:

* cross\_val\_score evaluates the model using cross-validation.
* cv=3 means the dataset is split into 3 folds.
* The function returns an array of scores for each fold, which can be averaged to estimate the model's performance.

## **Overfitting – Underfitting and Model Evaluation**

1. **Objective**:
   * The goal is to determine the best polynomial order to estimate the function ( y(x) ) from training points that come from a polynomial function plus some noise.
2. **Underfitting**:
   * **Definition**: Occurs when the model is too simple to capture the underlying trend of the data.
   * **Example**: Fitting a linear function to a complex polynomial function results in high errors.
3. **Overfitting**:
   * **Definition**: Happens when the model is too complex and captures noise rather than the actual function.
   * **Example**: A 16th order polynomial may fit the training data perfectly but performs poorly on unseen data, especially where there is little training data.
4. **Mean Square Error (MSE)**:
   * A plot of MSE for training and testing sets shows:
     + Training error decreases with increasing polynomial order.
     + Test error decreases until reaching the optimal polynomial order, after which it starts to increase (indicating overfitting).
5. **Selecting the Best Polynomial Order**:
   * The optimal order minimizes the test error. In the example, the best order was found to be 8.
6. **Irreducible Error**:
   * This is the error due to noise in the data, which cannot be predicted or reduced.
7. **Model Assumptions**:
   * If the polynomial assumption is incorrect (e.g., data generated from a sine wave), the model will not fit well.

Practical Example with Horsepower Data

* **Data Points**:
  + Red points: Training data
  + Green points: Test data
* **Model Performance**:
  + Mean of data: Poor performance.
  + Linear function: Better fit.
  + Higher-order polynomials (2nd, 3rd, 4th) show varying performance, with the 4th order exhibiting erratic predictions.

R² Value Calculation

* **R² Value**: Indicates how well the model explains the variability of the data.
  + Closer to 1 means a better fit.
  + The R² value is optimal at a polynomial order of 3, and it decreases significantly at order 4.

Code Example for R² Calculation

Here’s a simplified code snippet to calculate R² values for different polynomial orders:

import numpy as np

import matplotlib.pyplot as plt

from sklearn.preprocessing import PolynomialFeatures

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import r2\_score

# Sample data

X\_train = np.array([...]) # Replace with actual training data

y\_train = np.array([...]) # Replace with actual training labels

X\_test = np.array([...]) # Replace with actual test data

y\_test = np.array([...]) # Replace with actual test labels

# List to store R² values

r2\_values = []

# Loop through polynomial orders

for order in range(1, 10): # Adjust range as needed

poly = PolynomialFeatures(degree=order)

X\_poly\_train = poly.fit\_transform(X\_train)

model = LinearRegression()

model.fit(X\_poly\_train, y\_train)

# Predict on test data

X\_poly\_test = poly.transform(X\_test)

y\_pred = model.predict(X\_poly\_test)

# Calculate R²

r2 = r2\_score(y\_test, y\_pred)

r2\_values.append(r2)

# Plot R² values

plt.plot(range(1, 10), r2\_values)

plt.xlabel('Polynomial Order')

plt.ylabel('R² Value')

plt.title('R² Value vs Polynomial Order')

plt.show()

Summary

* **Model Selection**: Choose the polynomial order that minimizes test error.
* **Underfitting vs Overfitting**: Balance complexity to avoid both underfitting and overfitting.
* **R² Value**: Use it to evaluate model performance.

Feel free to revisit these notes as you continue your learning journey! If you

**Introduction to Ridge Regression**

For models with multiple independent features and ones with polynomial feature extrapolation, it is common to have colinear combinations of features. Left unchecked, this multicollinearity of features can lead the model to overfit the training data. To control this, the feature sets are typically regularized using hyperparameters.

Ridge regression is the process of regularizing the feature set using the hyperparameter alpha. The upcoming video shows how Ridge regression can be utilized to regularize and reduce standard errors and avoid over-fitting while using a regression model.

## **Ridge Regression**

* **Purpose**: Ridge regression is used to prevent overfitting, especially when dealing with multiple independent variables or features.
* **Polynomial Regression**:
  + A fourth-order polynomial can fit data well, but using a higher-order polynomial (like a 10th order) can lead to overfitting, especially with outliers.
* **Alpha Parameter**:
  + Ridge regression introduces a parameter called **Alpha** to control the magnitude of polynomial coefficients.
  + As **Alpha** increases, the coefficients decrease, which helps in reducing overfitting.
  + If **Alpha** is too large, it can lead to underfitting (coefficients approach zero).
* **Cross Validation**:
  + To select the optimal **Alpha**, cross-validation is used. This involves training the model with different values of **Alpha** and evaluating performance using metrics like R² or mean squared error.

Code Example

Here’s a simple code snippet to implement ridge regression using Python's scikit-learn library:

import numpy as np

import matplotlib.pyplot as plt

from sklearn.linear\_model import Ridge

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import r2\_score

# Generate synthetic data

np.random.seed(0)

X = np.random.rand(100, 1) \* 10

y = 2 \* (X \*\* 2) + np.random.randn(100, 1) \* 5

# Split the data into training and validation sets

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# List to store R² scores for different Alpha values

alpha\_values = [0.001, 0.01, 0.1, 1, 10]

r2\_scores = []

for alpha in alpha\_values:

model = Ridge(alpha=alpha)

model.fit(X\_train, y\_train)

y\_pred = model.predict(X\_val)

r2\_scores.append(r2\_score(y\_val, y\_pred))

# Plotting R² scores

plt.plot(alpha\_values, r2\_scores, marker='o')

plt.xscale('log')

plt.xlabel('Alpha')

plt.ylabel('R² Score')

plt.title('Ridge Regression: R² Score vs Alpha')

plt.grid()

plt.show()

Explanation of the Code

* **Data Generation**: Synthetic data is created using a quadratic function with added noise.
* **Train-Test Split**: The data is split into training and validation sets.
* **Ridge Regression**: A loop iterates over different values of **Alpha**, fitting a ridge regression model for each and calculating the R² score.
* **Plotting**: The R² scores are plotted against the logarithmic scale of **Alpha** to visualize the effect of different **Alpha** values on model performance.

## **Grid Search**

* **Grid Search**: A method to optimize hyperparameters of a model by systematically working through multiple combinations of parameter values.
* **Hyperparameters**: Parameters that are not learned from the data but are set before the training process (e.g., alpha in Ridge Regression).
* **Cross-Validation**: A technique to assess how the results of a statistical analysis will generalize to an independent dataset.

Steps in Grid Search

1. **Define the Model**: Choose the model you want to optimize (e.g., Ridge Regression).
2. **Set Hyperparameters**: Create a dictionary of hyperparameters and their possible values.
3. **Grid Search**: Use GridSearchCV from Scikit-learn to perform the search.
4. **Fit the Model**: Train the model using the training data.
5. **Evaluate**: Use metrics like Mean Squared Error (MSE) or R-squared to evaluate model performance.

Example Code

Here’s a simple example using Ridge Regression with Grid Search:

from sklearn.model\_selection import GridSearchCV

from sklearn.linear\_model import Ridge

import numpy as np

# Sample data

X = np.random.rand(100, 10) # 100 samples, 10 features

y = np.random.rand(100) # 100 target values

# Define the model

ridge = Ridge()

# Define the hyperparameters and their values

param\_grid = {

'alpha': [0.1, 1.0, 10.0], # Different values for alpha

'normalize': [True, False] # Normalize option

}

# Create the GridSearchCV object

grid\_search = GridSearchCV(estimator=ridge, param\_grid=param\_grid, scoring='r2', cv=5)

# Fit the model

grid\_search.fit(X, y)

# Get the best parameters

best\_params = grid\_search.best\_params\_

print("Best parameters:", best\_params)

# Get the best estimator

best\_model = grid\_search.best\_estimator\_

print("Best model:", best\_model)

Explanation of the Code

* **Import Libraries**: Import necessary libraries for model building and evaluation.
* **Sample Data**: Create random data for demonstration purposes.
* **Define the Model**: Initialize the Ridge regression model.
* **Hyperparameter Grid**: Create a dictionary with hyperparameters and their possible values.
* **GridSearchCV**: Initialize the Grid Search with the model, parameter grid, scoring method (R-squared), and number of cross-validation folds (cv=5).
* **Fit the Model**: Train the model using the fit method.
* **Best Parameters**: Retrieve the best hyperparameters found during the search.
* **Best Model**: Get the best model based on the hyperparameters.

This process allows you to efficiently find the optimal hyperparameters for your model, improving its performance on unseen data.

**Lesson Summary**

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

* How to split your data using the train\_test\_split() method into training and test sets. You use the training set to train a model, discover possible predictive relationships, and then use the test set to test your model to evaluate its performance.
* How to use the generalization error to measure how well your data does at predicting previously unseen data.
* How to use cross-validation by splitting the data into folds where you use some of the folds as a training set, which we use to train the model, and the remaining parts are used as a test set, which we use to test the model. You iterate through the folds until you use each partition for training and testing. At the end, you average results as the estimate of out-of-sample error.
* How to pick the best polynomial order and problems that arise when selecting the wrong order polynomial by analyzing models that underfit and overfit your data.
* Select the best order of a polynomial to fit your data by minimizing the test error using a graph comparing the mean square error to the order of the fitted polynomials.
* You should use ridge regression when there is a strong relationship among the independent variables.
* That ridge regression prevents overfitting.
* Ridge regression controls the magnitude of polynomial coefficients by introducing a hyperparameter, alpha.
* To determine alpha, you divide your data into training  and validation data. Starting with a small value for alpha, you train the model, make a prediction using the validation data, then calculate the R-squared and store the values. You repeat the value for a larger value of alpha. You repeat the process for different alpha values, training the model, and making a prediction. You select the value of alpha that maximizes R-squared.
* That grid search allows you to scan through multiple hyperparameters using the Scikit-learn library, which iterates over these parameters using cross-validation. Based on the results of the grid search method, you select optimum hyperparameter values.
* The GridSearchCV() method takes in a dictionary as its argument where the key is the name of the hyperparameter, and the values are the hyperparameter values you wish to iterate over.