Introduction to Machine Learning
(DSAI_S2_CCO_DS_7976)

SESSION 2

Victor Planas-Bielsa

Contents

Important note: These notes are a detailed guide for the course, but they are not the full content. The course will be complemented with explanations on the board, exercises, discussions, and hands-on practice that we will do in class.

Lesson Learning Objectives.
The learning objectives for this lesson are as follows:

- Understand the Fundamentals of Regression
- Explore and Compare Regression Techniques
- Implement Regression Models Using Python
- Evaluate and Interpret Regression Models

Keep these objectives in mind while studying the material !

1 Regression Methods

1.1 Introduction to the Regression Problem

The regression problem is central to both statistics and machine learning, involving the prediction of a continuous dependent variable y based on one or more independent variables $x_1, x_2, \ldots, x_n$. This section provides a detailed exploration of the regression problem, encompassing its mathematical formulation, differentiation between linear and nonlinear regression, optimization of loss functions, and practical considerations for implementation.

At its core, regression seeks to establish a functional relationship between a dependent variable y and a set of independent variables $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}$. Formally, the regression model is expressed as:

$$y = f(\mathbf{x}) + \epsilon,$$

where:

- $f(\mathbf{x})$: The true underlying function that maps the input variables to the output variable.
- ϵ: A random noise term capturing variability not explained by $f(\mathbf{x})$.

The goal is to approximate $f(\mathbf{x})$ using a hypothesis function $\hat{f}(\mathbf{x}; \theta)$, where θ represents the model parameters. The approximation should minimize the error between predictions $\hat{y}$ and true values y.

Assumptions in Regression Models. For regression models to provide valid inferences and predictions, certain assumptions are typically made:

1. **Independence**: Observations are independent of each other.
2. **Homoscedasticity**: The variance of the error terms is constant across all levels of the independent variables.
3. **Normality**: The error terms are normally distributed.
4. **No Multicollinearity**: Independent variables are not highly correlated with each other.

Violations of these assumptions can lead to biased or inefficient estimates, necessitating diagnostic checks and potential model adjustments.

1.2 Types of Regression Problems

Regression problems can be categorized based on the functional form of the relationship between the dependent and independent variables, as well as the number of independent variables involved. The primary types include:

- Linear Regression
- Polynomial Regression

- Non-linear Regression
- Non-parametric Regression

Below, each regression type is elaborated with explanations and examples to illustrate their applications and distinctions.

Linear Regression

Linear regression models the relationship between a dependent variable y and one or more independent variables $\mathbf{x}$ by assuming that y is a linear combination of the predictors. This approach can be applied in two primary scenarios, *simple linear regression* and *multiple linear regression*, depending on the number of independent variables involved.

In *Simple Linear Regression*, the model includes a single independent variable, and the relationship is represented as:

$$y = \beta_0 + \beta_1 x + \epsilon,$$

where β_0 is the intercept, β_1 is the slope coefficient, and ϵ is the error term accounting for deviations not explained by the linear relationship. For example, consider predicting a student's exam score based solely on the number of hours they studied. Here, the independent variable (x) represents hours studied, and the dependent variable (y) represents the exam score. The model assumes a direct linear relationship between these two variables.

Multiple Linear Regression, on the other hand, generalizes this approach by incorporating two or more independent variables. The model can be expressed as:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p + \epsilon,$$

where $x_1, x_2, \ldots, x_p$ represent the independent variables. Each predictor contributes linearly to the dependent variable, with $\beta_1, \beta_2, \ldots, \beta_p$ capturing their respective effects. An illustrative example is predicting the price of a house based on its size, number of bedrooms, and age. In this case, the model assumes that each of these features contributes independently and linearly to the overall price of the house.

Through these formulations, linear regression provides a simple yet powerful framework for analyzing relationships between variables, enabling both explanation and prediction.

Polynomial Regression

Polynomial Regression is an extension of linear regression that models the relationship between the dependent variable y and the independent variables as an nth degree polynomial. While it retains linearity in the parameters, it allows the model to capture non-linear relationships between y and the predictors.

For a single predictor variable, the model takes the form:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_d x^d + \epsilon,$$

where d is the degree of the polynomial. Higher-order terms, such as x^2 or x^d, enable the model to fit curved relationships in the data.

In cases with multiple predictor variables, polynomial regression can also include **cross terms** (interaction terms) between predictors, as well as higher-order terms for individual variables. For example, for two predictors x_1 and x_2, a second-degree polynomial regression model could take the form:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1^2 + \beta_4 x_2^2 + \beta_5 x_1 x_2 + \epsilon.$$

Here, the term $x_1 x_2$ represents the interaction between x_1 and x_2, while x_1^2 and x_2^2 are the squared terms for the individual predictors. These interaction terms allow the model to account for dependencies between predictors, providing additional flexibility in capturing non-linear relationships.

Polynomial regression is very similar to linear regression in the sense that, after extending the features to include polynomial terms, the resulting model remains linear in the parameters. This means the fitting process uses the same methods as linear regression, such as solving a system of linear equations or using gradient descent.

However, there is a significant risk of overfitting if too many polynomial terms are added to the model, as this can lead to a solution that fits the noise in the training data rather than the underlying pattern. Therefore, careful selection of the polynomial degree and the use of regularization techniques or validation methods are important to ensure the model generalizes well to new data.

Polynomial regression is commonly used in scenarios where the relationship between the predictors and the response variable is non-linear, but the model must remain interpretable and computationally feasible.

Non-linear Regression

Non-Linear Regression extends regression analysis by modeling the relationship between the dependent variable y and one or more independent variables using non-linear functions. Unlike polynomial regression, which retains linearity in the parameters, non-linear regression allows for non-linear relationships in both the predictors and parameters, enabling it to capture a broader range of complex dynamics.

For a single independent variable, non-linear regression can model relationships that are inherently non-linear and cannot be adequately represented by linear or polynomial models. For instance, the model:

$$y = \beta_0 e^{\beta_1 x} + \epsilon$$

describes exponential growth or decay. A practical example is modeling the decay of a radioactive substance, where the amount remaining decreases exponentially over time. Here, the independent variable x represents time, and the dependent variable y represents the remaining quantity of the substance.

Non-linear regression can also accommodate multiple independent variables, allowing for the modeling of complex interactions between predictors. An example of such a model is:

$$y = \beta_0 + \beta_1 e^{\beta_2 x_1} + \beta_3 \ln(x_2) + \epsilon,$$

which combines exponential and logarithmic relationships. This is useful in scenarios such as predicting crop yield, where fertilizer amount (x_1) may have an exponential relationship with yield, while rainfall (x_2) follows a logarithmic relationship. Non-linear regression provides the flexibility needed to accurately model these diverse effects and interactions.

In some cases, proposing a suitable functional form for non-linear regression can be challenging, particularly when dealing with high-dimensional data. In lower dimensions (e.g., one or two independent variables), it may be possible to suggest a functional form through visual inspection of the data. However, as the dimensionality increases, this approach becomes infeasible. In such cases, domain knowledge or theoretical insights into the system under study are invaluable. For example, in physics, the behavior of a damped harmonic oscillator can be modeled using a combination of sinusoidal and exponential terms:

$$y = Ae^{-\gamma t}\cos(\omega t + \phi),$$

where A is the amplitude, γ is the damping coefficient, ω is the angular frequency, t is time, and ϕ is the phase. This prior knowledge of the system dynamics allows us to define a functional form that captures the underlying physical behavior accurately, illustrating the importance of combining empirical data analysis with a priori theoretical understanding.

Non-parametric Regression

Non-parametric Regression refers to a collection of regression techniques that do not assume a predetermined form for the relationship between the dependent variable and the independent variables. Unlike parametric regression models, which specify a fixed number of parameters and a specific functional form (e.g., linear, polynomial), non-parametric methods are more flexible and can adapt to the underlying structure of the data. This flexibility allows non-parametric regression to capture complex, nonlinear relationships without imposing strict assumptions on the model form.

Main Types of Non-parametric Regression

Non-parametric regression encompasses a variety of methods, each with its unique approach to modeling data. The primary types include:

1. **Spline Regression**: Spline regression fits piecewise polynomial functions, known as splines, between specified points called knots. This approach allows different sections of the data to be modeled with different polynomial degrees, ensuring smooth transitions at the knots.

2. **Local Polynomial Regression**: This method fits a low-degree polynomial (e.g., linear or quadratic) to a localized subset of the data around each point of interest. By doing so, it captures local trends while maintaining flexibility across the entire dataset.

3. **Regression Trees**: Regression trees partition the data into subsets based on the values of the independent variables. Each partition corresponds to a leaf node in the tree, which contains a simple prediction model, typically the mean of the dependent variable for observations in that node.

4. **k-Nearest Neighbors (k-NN) Regression**: k-NN regression predicts the value of a new observation based on the average (or weighted average) of the dependent variable values of its 'k' closest neighbors in the feature space.

5. **Kernel Regression**: Kernel regression estimates the relationship between variables using a weighted average of nearby observations. The weights are determined by a kernel function, which decreases with distance, giving more influence to points closer to the target observation.

6. **Neural Networks**: Neural networks are computational models inspired by the human brain, composed of interconnected layers of nodes or neurons. They are capable of learning complex, nonlinear relationships from data by adjusting the weights of connections through training algorithms such as backpropagation. Due to their flexibility and ability to approximate a wide variety of functions without assuming a predetermined functional form, neural networks are considered a non-parametric approach within regression analysis.

Comparison Between Parametric and Non-parametric Regression.

The following table summarizes the key differences between parametric and non-parametric regression methods. Note that non-parametric approaches are more closely aligned with machine learning paradigms due to their flexibility and ability to model complex, non-linear relationships.

Table 1: Comparison Between Parametric and Non-parametric Regression

Criterion	**Parametric Regression**	**Non-parametric Regression**
Flexibility	Limited flexibility due to the assumption of a specific functional form. Suitable for data where the relationship is well-understood and can be accurately captured by the chosen model.	Highly flexible, allowing the model to adapt to various data structures and capture complex, nonlinear relationships without predefined equations.
Assumptions	Relies on strong assumptions about the form of the relationship (e.g., linearity, normality of errors). Violations of these assumptions can lead to biased or inefficient estimates.	Makes fewer assumptions about the data, reducing the risk of model misspecification. However, this flexibility can make non-parametric methods more susceptible to overfitting if not properly regularized.
Interpretability	Models are generally more interpretable due to their simple and explicit functional forms. Coefficients have clear meanings (e.g., the effect of a predictor on the response variable).	Models can be less interpretable, especially for complex models like Regression Trees or Neural Networks, where the relationship is not easily summarized by simple parameters.
Computational Complexity	Typically computationally efficient, especially for models with fewer parameters.	Can be computationally intensive, particularly with large datasets or complex models that require extensive computation.
Data Requirements	Can perform well with smaller datasets if the model assumptions hold.	Generally require larger datasets to accurately capture the underlying relationships without overfitting.
Smoothness and Continuity	The smoothness of the fitted relationship is determined by the chosen functional form (e.g., linear vs. polynomial).	Offers greater control over smoothness through techniques like smoothing splines or bandwidth selection in kernel methods, allowing for more nuanced modeling of data trends.

1.3 Loss Functions

Loss functions are fundamental components in regression analysis and machine learning, serving as quantitative measures of the difference between predicted values and actual outcomes. By defining a specific loss function, we establish an objective for optimization algorithms to minimize, thereby enhancing the model's accuracy and generalizability. Selecting an appropriate loss function is crucial, as it directly influences the model's performance, robustness, and suitability for a given dataset.

Common Loss Functions in Regression

Understanding various loss functions is essential for selecting the most appropriate one based on the specific characteristics of the dataset and the objectives of the analysis. Below are some of the most commonly used loss functions in regression:

- Mean Squared Error (MSE). Mean Squared Error calculates the average of the squares of the errors between predicted and actual values. Its formula is given by:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

MSE is highly sensitive to outliers, making it suitable for scenarios where larger errors should be penalized more heavily.

- Mean Absolute Error (MAE) Mean Absolute Error computes the average of the absolute differences between predicted and actual values:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

MAE is more robust to outliers compared to MSE, providing a linear loss that treats all deviations equally.

- Huber Loss Huber Loss combines the properties of MSE and MAE, offering robustness to outliers while maintaining differentiability:

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \le \delta, \\ \delta(|a| - \frac{1}{2}\delta) & \text{otherwise.} \end{cases}$$

This loss function is particularly useful when a balance between sensitivity to outliers and smooth optimization is required.

- Quantile Loss Quantile Loss is employed for predicting specific quantiles and is defined as:

$$L_\tau(y, \hat{y}) = \max(\tau(y - \hat{y}), (\tau - 1)(y - \hat{y}))$$

It is asymmetric and useful in applications where predicting intervals or median values is necessary.

Selecting and Understanding the Impact of Loss Functions

The choice of loss function is central to achieving optimal model performance, as it shapes both the training dynamics and the model's ability to generalize to new data. This decision depends on several interrelated factors, starting with the nature of the dataset. For instance, datasets with significant outliers often require robust loss functions such as Mean Absolute Error (MAE) or Huber Loss. These functions reduce the influence of extreme data points, unlike Mean Squared Error (MSE), which amplifies large deviations due to its squared penalty. As a result, robust loss functions help the model maintain stability and performance in the presence of anomalies.

Loss functions also play a crucial role in determining the efficiency of the training process. Smooth and differentiable functions, like MSE, typically enable faster and more stable convergence when optimization relies on gradient-based methods. In contrast, non-differentiable functions such as MAE may lead to slower convergence or demand more specialized algorithms. Consequently, selecting a loss function often involves balancing computational efficiency with the nature of the problem being addressed.

The specific objectives of the analysis further influence the choice of loss function. MSE, for example, is particularly effective when minimizing large errors is critical, as its squared penalty emphasizes these deviations. On the other hand, applications requiring a more balanced treatment of errors may benefit from alternatives like MAE or Quantile Loss, which are better suited to asymmetric error distributions or interval predictions.

Generalization is another critical consideration. A well-chosen loss function contributes to a model's ability to perform consistently on unseen data by focusing optimization on the majority of the dataset rather than being skewed by outliers. This helps reduce overfitting, particularly in noisy datasets, and ensures the model captures meaningful patterns.

Lastly, interpretability and computational considerations can guide the selection process. MSE's differentiability and convexity make it computationally efficient for gradient-based optimization, while MAE offers a more intuitive interpretation of error magnitude. Both simplicity and alignment with the analysis's practical goals should inform the decision.

In summary, selecting a loss function is not merely a technical choice but a deliberate balancing act. Understanding how the loss function interacts with data characteristics, optimization dynamics, and model objectives ensures that the resulting model is robust, efficient, and well-suited to its intended purpose.

Regularization: Controlling Model Complexity

Regularization is a fundamental technique in machine learning and regression analysis to address the problem of overfitting. By adding a penalty term to the loss function, regularization discourages overly complex models and promotes simplicity, improving the model's ability to generalize to new, unseen data.

1. Why Regularization is Necessary Without regularization, models, particularly those with high capacity, may fit the noise in the training data rather than the underlying patterns. This overfitting results in poor performance on validation and test sets. Regularization works by imposing constraints on the model parameters, effectively controlling their magnitude or encouraging sparsity, which prevents the model from becoming excessively complex.

2. Types of Regularization There are several common types of regularization, each suited to specific scenarios:

- L1 Regularization (Lasso): L1 regularization adds the absolute value of the coefficients to the loss function:

$$\text{Loss}_{\text{L1}} = \text{Loss} + \lambda \sum_{j=1}^{p} |w_j|,$$

where λ is the regularization strength, w_j are the model coefficients, and p is the number of parameters. L1 regularization encourages sparsity, setting many coefficients to exactly zero. This makes it particularly useful for feature selection.

- L2 Regularization (Ridge): L2 regularization adds the squared values of the coefficients to the loss function:

$$\text{Loss}_{\text{L2}} = \text{Loss} + \lambda \sum_{j=1}^{p} w_j^2.$$

Unlike L1, L2 regularization shrinks coefficients towards zero without eliminating them, which helps prevent overfitting while retaining all features.

- Elastic Net Regularization: Elastic Net combines L1 and L2 penalties:

$$\text{Loss}_{\text{ElasticNet}} = \text{Loss} + \lambda_1 \sum_{j=1}^{p} |w_j| + \lambda_2 \sum_{j=1}^{p} w_j^2.$$

This approach balances sparsity and smoothness, making it suitable for datasets with highly correlated features.

3. Regularization in Practice The choice of regularization method and its strength (λ) depends on the dataset and the objectives of the analysis:

-*Tuning λ:* The regularization strength λ controls the tradeoff between the model's fit to the training data and its complexity. Small values of λ result in minimal regularization, while large values impose stronger penalties, simplifying the model. Techniques like cross-validation are commonly used to determine the optimal value of λ.

- *Feature Importance:* Regularization often alters the relative importance of features in the model. For instance, L1 regularization eliminates less relevant features, while L2 reduces their influence but retains all features.

4. Connections with Loss Functions and Model Selection Regularization is closely tied to loss functions, as the penalty term is incorporated into the optimization objective. For example, Ridge regression minimizes the Mean Squared Error (MSE) with an added L2 penalty, while Lasso minimizes MSE with an L1 penalty. Furthermore, regularization complements model selection criteria such as AIC and BIC by directly addressing model complexity during the training process.

5. Regularization Techniques Beyond Linear Models While the above methods are commonly used in linear regression, regularization is also applied in more complex models, such as:

- *Neural Networks:* Techniques like weight decay (analogous to L2 regularization) and dropout (randomly dropping units during training) are used to prevent overfitting.

- *Decision Trees and Ensemble Methods:* Regularization can take the form of limiting tree depth, pruning, or reducing the number of trees in ensemble models like Random Forests and Gradient Boosted Trees.

6. Benefits of Regularization Regularization not only prevents overfitting but also improves interpretability by simplifying the model. It helps address issues of multicollinearity, stabilizes parameter estimates, and ensures that the model generalizes well across different datasets.

1.4 Model Evaluation and Selection: Assessing Model Suitability

Determining whether a model is good enough or too complex, and deciding when it is appropriate to finalize a model, are fundamental aspects of regression and machine learning. This process involves a combination of diagnostic techniques, validation strategies, and model selection criteria.

Assessing Predictive Performance with Cross-Validation Cross-validation is an essential technique for evaluating a model's generalization capability. Among its variants, k-fold cross-validation is particularly robust. Here, the dataset is divided into k subsets (folds), and the model is trained on $k-1$ folds and validated on the remaining fold. This process is repeated k times, with each fold acting as the validation set once. The average performance metrics across all iterations, such as Mean Squared Error (MSE), Root Mean Squared Error (RMSE), or R-squared (R^2), provide an unbiased estimate of how well the model generalizes to unseen data.

If cross-validation indicates high variability in performance across folds, the model may be overly sensitive to specific data points, a sign of overfitting. Conversely, consistently poor performance across folds may indicate underfitting or an inadequate model structure.

Residual Analysis for Model Diagnostics Residuals, defined as the differences between predicted and observed values, provide critical insights into model fit. Proper residual analysis involves the following checks:

- *Random Distribution:* Residuals should be randomly distributed around zero, with no discernible patterns. Systematic patterns, such as trends or curves, suggest that the model fails to capture important features or relationships in the data.

- *Homoscedasticity:* The variance of residuals should be consistent across all levels of the predicted values. If the residual variance increases or decreases systematically, it indicates heteroscedasticity, which may require transformation of variables or adjustments to the model.

- *Absence of Outliers:* Significant outliers in residuals can unduly influence the model. Identifying and addressing these points—either by applying robust loss functions or re-evaluating data preprocessing—is crucial.

Evaluating Model Complexity: Bias-Variance Tradeoff A well-fitted model balances the bias-variance tradeoff:

- *High Bias (Underfitting):* The model is too simple and fails to capture the complexity of the data, leading to poor training and validation performance. This is often reflected in high bias, systematic prediction errors, and low R^2.

- *High Variance (Overfitting):* The model is overly complex and fits noise in the training data, resulting in excellent training performance but poor validation or test performance. Overfitting is often accompanied by low generalization capability and high variability in cross-validation scores.

Finding the right balance involves selecting a model that performs well on both training and validation data without significant disparity in their metrics.

Information Criteria for Model Comparison The Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC) are widely used metrics for model selection. Both criteria balance the goodness of fit with model complexity, helping to prevent overfitting by penalizing models with excessive parameters.

- **Akaike Information Criterion (AIC):** AIC estimates the relative information loss when a model is used to represent the data-generating process. It is calculated as:

$$\text{AIC} = 2k - 2\ln(\mathcal{L}),$$

where k is the number of parameters in the model, and $\ln(\mathcal{L})$ is the natural logarithm of the maximum likelihood estimate of the model. Lower AIC values indicate models with better trade-offs between fit and complexity. AIC is particularly useful when comparing models, as it quantifies the relative quality of each model, with no upper or lower bound.

- **Bayesian Information Criterion (BIC):** BIC is similar to AIC but introduces a stronger penalty for the number of parameters, making it more conservative in selecting complex models. The formula for BIC is:

$$\text{BIC} = \ln(n)k - 2\ln(\mathcal{L}),$$

where n is the number of observations in the dataset. Like AIC, lower BIC values are preferable, but its stricter penalty for complexity means it tends to favor simpler models when datasets are small.
Maximum Likelihood Estimation (MLE): The likelihood $\mathcal{L}$ represents the probability of observing the given data under the fitted model. Maximum likelihood estimation (MLE) finds the parameter values that maximize this likelihood. Libraries such as `statsmodels` or `scikit-learn` typically compute $\ln(\mathcal{L})$ during model fitting. For example, in `statsmodels`, the log-likelihood is stored as an attribute of the fitted model:

Example: Compute Likelihood in Python

```python
import statsmodels.api as sm
import numpy as np

# Example data
x = np.random.rand(100)  # Predictor
y = 3 * x + np.random.normal(0, 0.5, 100)  # Response with noise

# Add constant for the intercept
x_with_const = sm.add_constant(x)

# Fit linear regression model
model = sm.OLS(y, x_with_const).fit()

# Log-likelihood
log_likelihood = model.llf
print("Log-likelihood:", log_likelihood)
```

The log-likelihood value ($\ln(\mathcal{L})$) can then be used to compute AIC or BIC along with the number of parameters (k) and observations (n).

Domain Knowledge and Practical Considerations Ultimately, the appropriateness of a model depends on its alignment with the goals of the analysis and the characteristics of the data. Even if a model performs well statistically, it must be interpretable and practical for deployment in real-world applications.

When to Finalize a Model You can consider a model finalized when the following conditions are met:

1. **Generalization Capability:** Cross-validation and test set performance indicate that the model performs consistently well across datasets.
2. **Diagnostics Check:** Residual analysis shows no patterns, heteroscedasticity, or significant outliers that compromise the model's validity.
3. **Simplicity and Interpretability:** The model achieves a balance between predictive accuracy and interpretability, with no unnecessary complexity.
4. **Domain Suitability:** The model satisfies the objectives of the analysis and aligns with domain-specific requirements.

By applying these techniques and considerations, you can ensure that your model is both robust and appropriately tailored to the problem at hand.

How to Use AIC and BIC for Model (in practice)? When comparing multiple models, compute the AIC or BIC values for each model and select the one with the lowest value. This model represents the best trade-off between goodness of fit and complexity. It is important to interpret the differences between AIC or BIC values:

- Differences less than 2: Models have equivalent predictive power, and either can be chosen.

- Differences between 2 and 10: There is moderate evidence that the model with the lower value is better.
- Differences greater than 10: Strong evidence supports the model with the lower value.

For example, suppose you are comparing two models:

- Model A: AIC = 150, BIC = 155
- Model B: AIC = 152, BIC = 158

In this case, the difference in AIC (2) suggests that both models are equivalent in predictive power. However, the larger difference in BIC (3) slightly favors Model A, particularly if simplicity is a priority.

Summary of Best Practices AIC is often preferred when the goal is predictive accuracy, as it emphasizes minimizing information loss. BIC, with its stronger penalty for complexity, is more suitable for explanatory models where parsimony is desired. Both criteria should be used in conjunction with performance metrics (e.g., MSE, R^2) and validation techniques like cross-validation to ensure a comprehensive evaluation of model performance.

Are AIC and BIC Available for All Models in Python? AIC and BIC are widely used metrics for model selection, but they are not implemented for all types of models in Python, particularly for machine learning models such as neural networks and regression trees. This limitation arises from the assumptions and requirements of AIC and BIC.

Why AIC and BIC Are Not Always Available: Both AIC and BIC rely on the presence of a likelihood function ($\mathcal{L}$), which measures the probability of observing the data given the model. Statistical models, such as linear regression or generalized linear models (GLMs), explicitly define this likelihood function. However, many machine learning models, such as neural networks and regression trees, minimize alternative loss functions, such as Mean Squared Error (MSE) or cross-entropy, which are not directly tied to a likelihood.

Additionally, AIC and BIC require the number of model parameters (k) to compute the penalty for model complexity. While k is straightforward to determine in statistical models, it becomes ambiguous in models like neural networks (due to numerous weights and biases) or tree-based models (due to the number of splits or leaves).

When AIC and BIC Are Available: AIC and BIC are typically available for statistical models implemented in Python libraries such as `statsmodels`. Examples include:

- Linear Regression
- Generalized Linear Models (e.g., logistic regression, Poisson regression)
- Time-series models (e.g., ARIMA)

For example, AIC and BIC can be computed for a linear regression model as follows:

Example: Compute AIC and BIC in Python

```python
import statsmodels.api as sm

# Example with linear regression
x = sm.add_constant([1, 2, 3, 4, 5])  # Add constant for the
    intercept
y = [1.2, 2.1, 3.2, 4.1, 5.3]         # Response variable

# Fit the model
model = sm.OLS(y, x).fit()

# Access AIC and BIC
print("AIC:", model.aic)
print("BIC:", model.bic)
```

Alternatives for Machine Learning Models: For machine learning models, such as neural networks and regression trees, AIC and BIC are not natively available in Python because these models lack an explicit likelihood function. Instead, other approaches are commonly used to compare models:

- **Cross-Validation:** Evaluate model performance across multiple data splits to ensure generalization.
- **Hyperparameter Optimization:** Techniques like Grid Search or Random Search minimize validation loss directly.
- **Information-Theoretic Criteria:** Approximations such as the Minimum Description Length (MDL) or Bayesian methods may serve as alternatives for model comparison.

Best Practices for Model Comparison: When using machine learning models, rely on alternatives like cross-validation and evaluation metrics to compare model performance. For statistical models where AIC and BIC are available, these criteria can provide valuable insights into the trade-off between model complexity and goodness of fit. If you need likelihood-based comparisons for machine learning models, custom implementations or domain-specific approximations may be necessary.

1.5 Parameter Estimation and Optimization in Nonlinear Regression

In contrast to linear regression, where parameter estimation often relies on closed-form solutions, nonlinear regression requires iterative optimization algorithms to minimize a chosen loss function. These algorithms adjust model parameters step-by-step to reduce the discrepancy between observed data and the model's predictions. Gradient-based methods are particularly popular due to their effectiveness and versatility.

The Gradient Descent Algorithm

Gradient descent iteratively updates parameters by taking steps proportional to the negative gradient of the loss function. For a parameter θ, the update rule is:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta L(\theta),$$

where η is the learning rate and $\nabla_\theta L(\theta)$ represents the gradient of the loss function with respect to θ. The choice of η is critical: a value too large can cause divergence, while one too small can lead to slow convergence.

Example 1: Gradient Descent for Linear Regression (Univariate Case) Consider a dataset $\{(x_i, y_i)\}_{i=1}^n$ and a simple linear model $y = \theta_0 + \theta_1 x$. The loss function, using Mean Squared Error (MSE), is:

$$L(\theta_0, \theta_1) = \frac{1}{2n} \sum_{i=1}^n \left(y_i - (\theta_0 + \theta_1 x_i)\right)^2.$$

The gradients with respect to θ_0 and θ_1 are:

$$\frac{\partial L}{\partial \theta_0} = -\frac{1}{n} \sum_{i=1}^n \left(y_i - (\theta_0 + \theta_1 x_i)\right),$$

$$\frac{\partial L}{\partial \theta_1} = -\frac{1}{n} \sum_{i=1}^n x_i \left(y_i - (\theta_0 + \theta_1 x_i)\right).$$

Using these gradients, the parameters are updated iteratively:

$$\theta_0^{(t+1)} = \theta_0^{(t)} - \eta \frac{\partial L}{\partial \theta_0}, \quad \theta_1^{(t+1)} = \theta_1^{(t)} - \eta \frac{\partial L}{\partial \theta_1}.$$

Example 2: Gradient Descent for a Damped Harmonic Oscillator For the damped harmonic oscillator, the model is:

$$y(t) = Ae^{-\gamma t} \cos(\omega t),$$

where A, γ, and ω are the parameters to estimate. Given a dataset $\{(t_i, y_i)\}$, the loss function is:

$$L(A, \gamma, \omega) = \frac{1}{2n} \sum_{i=1}^n \left(y_i - Ae^{-\gamma t_i} \cos(\omega t_i)\right)^2.$$

The gradient with respect to γ is:

$$\frac{\partial L}{\partial \gamma} = -\frac{1}{n} \sum_{i=1}^n \left(y_i - Ae^{-\gamma t_i} \cos(\omega t_i)\right) \cdot \left(-At_i e^{-\gamma t_i} \cos(\omega t_i)\right).$$

This gradient is then used in the update rule:

$$\gamma^{(t+1)} = \gamma^{(t)} - \eta \frac{\partial L}{\partial \gamma}.$$

Similar expressions can be derived for A and ω, with the corresponding partial derivatives.

Advanced Optimization Techniques

While standard gradient descent is effective, it has limitations, particularly with noisy data or complex loss surfaces. Variants such as Stochastic Gradient Descent (SGD), Mini-Batch Gradient Descent, and adaptive methods like Adam improve performance by addressing issues like computational cost and sensitivity to learning rates. For instance, Adam combines momentum (to smooth updates) and adaptive learning rates (to adjust step sizes based on gradient magnitudes), making it robust for a wide range of problems.

Practical Considerations in Optimization

Convergence in optimization is typically determined by monitoring the loss function or the gradient norm. Adaptive learning rate schedules, such as exponential decay, can further enhance convergence efficiency. Ultimately, selecting the right algorithm and hyperparameters requires experimentation and validation, ensuring that the model achieves high accuracy without overfitting.

1.6 Where to Start: Selecting the Right Regression Technique

When faced with a new dataset, deciding on the most suitable regression model can be challenging. However, following a structured approach can simplify the process. Below are the key steps to guide you through model selection and implementation.

1. Understand the Nature of the Data

The first step is to thoroughly explore the dataset to understand its structure and characteristics. Begin with visualizations, such as scatter plots, to examine the relationships between the independent variables and the dependent variable. These plots can reveal whether the trends appear linear or non-linear, guiding your initial choice of models. Additionally, review summary statistics to identify any outliers or significant variability in the data that might affect your model.

Another critical aspect is determining the dimensionality of the data. For instance, datasets with a small number of predictors may be amenable to simpler approaches like linear regression. However, if the number of predictors is large relative to the number of observations, methods that handle high-dimensionality, such as regularization, might be required. Understanding these foundational aspects will help you narrow down the range of potential models.

2. Start Simple and Progressively Increase Complexity

Starting with simple models is often the best approach. Linear regression, for example, provides an excellent baseline when the relationship between the predictors and the dependent variable appears linear. Its simplicity and interpretability make it a valuable first step in understanding the data.

When the initial model does not adequately capture the data's behavior, consider extending it to polynomial regression. By introducing polynomial terms, the model can accommodate curved relationships, but it's important to limit the degree of the polynomial to avoid overfitting. For more complex datasets or when the relationship is inherently non-linear, exploring non-linear regression or non-parametric approaches, such as regression trees or splines, may be necessary. Always remember that simpler models should only be replaced when they fail to capture meaningful patterns in the data.

3. Use Model Selection and Validation Techniques

Choosing the best model requires a systematic evaluation of performance. Split the data into training and test sets to assess how well the model generalizes to unseen data. For a more robust evaluation, apply k-fold cross-validation, which reduces bias and variance by testing the model across multiple subsets of the data.

Model performance should be measured using appropriate metrics. For regression problems, metrics like Mean Squared Error (MSE), Mean Absolute Error (MAE), and R^2 are commonly used to quantify how well predictions match observed values. Comparing these metrics across different models helps identify the most effective one.

Additionally, consider using regularization techniques, such as Ridge or Lasso regression, when working with high-dimensional datasets or data prone to multicollinearity. These methods add penalties to the model parameters, reducing the likelihood of overfitting and improving generalization.

4. Incorporate Domain Knowledge

Domain knowledge can play a significant role in guiding model selection and feature engineering. For example, in physics, known functional relationships such as exponential decay or sinusoidal patterns can directly inform the choice of regression model. Similarly, in economics or biology, theoretical models can suggest interactions or transformations that enhance the analysis. By using these insights, you can create features or choose model forms that align with the underlying system being studied, often leading to more interpretable and accurate models.

5. Iterate and Refine

Modeling is rarely a one-step process. After selecting an initial model, validate its assumptions and review diagnostic checks to ensure it aligns with the data. For instance, verify whether residuals are randomly distributed, confirm homoscedasticity, and check for multicollinearity. If issues arise, adjust the model or experiment with alternative techniques.

Through this iterative process, you can gradually refine your model, improving both its fit and generalizability. Documenting your steps and findings is essential, as it not only helps justify your choices but also allows others to replicate or build upon your work.

Summary of Best Practices

To summarize:

- Begin with exploratory data analysis to understand the dataset's structure and relationships.
- Start with simple models and increase complexity only as needed.
- Use validation techniques like cross-validation and appropriate metrics to evaluate performance.

- Leverage domain knowledge to guide model selection and feature engineering.
- Iterate and refine the model through diagnostic checks and performance evaluations.

By following this structured approach, you can confidently select regression models that are well-suited to your data and objectives, balancing simplicity, accuracy, and interpretability.

Want to Learn More?

In some cases, datasets may contain outliers or influential points that can disproportionately affect model estimates and lead to misleading results. Detecting and addressing such points is an important step in data analysis. Several techniques can help identify outliers and influential observations, such as:

- **Leverage**: Measures how far an individual data point is from the average of all predictor values, indicating its potential to influence the model.
- **Cook's Distance**: Quantifies the effect of removing a given observation on the model's estimated coefficients, providing insight into the observation's overall impact.
- **DFFITS and DFBetas**: Evaluate the influence of individual data points on fitted values and regression coefficients, respectively.

When working with data that appears to contain outliers or unusual points, it is important to apply these methods to assess their impact. You are encouraged to consult resources on robust regression techniques and outlier diagnostics to learn more about identifying and handling such cases effectively.

1.7 Illustrative Example 1: Simple Linear Regression

In this example, we demonstrate how to perform simple linear regression to predict a dependent variable y from an independent variable x. We will generate synthetic data with a known underlying linear relationship, split the data into training and testing sets, implement linear regression using gradient descent, evaluate the model's performance, and compare the results with those obtained using a Python library.

Step 1: Generate Synthetic Data

We begin by generating a dataset that follows a linear relationship with added noise to simulate real-world observations. This allows us to have a ground truth for comparison.

Python Code: Generate Synthetic Data

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

# True parameters
theta_0_true = 2.0  # Intercept
theta_1_true = 3.0  # Slope

# Generate data
np.random.seed(7976)
x = np.linspace(0, 10, 100)  # Independent variable
y_true = theta_0_true + theta_1_true * x  # True linear relationship
noise = np.random.normal(scale=1.0, size=len(x))  # Random noise
y_noisy = y_true + noise  # Observed data

# Visualize the data
plt.scatter(x, y_noisy, label="Noisy Data", alpha=0.7)
plt.plot(x, y_true, 'r-', label="True Model")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.title("Synthetic Linear Data")
plt.show()
```

The plot below illustrates the generated noisy data points alongside the true linear relationship:

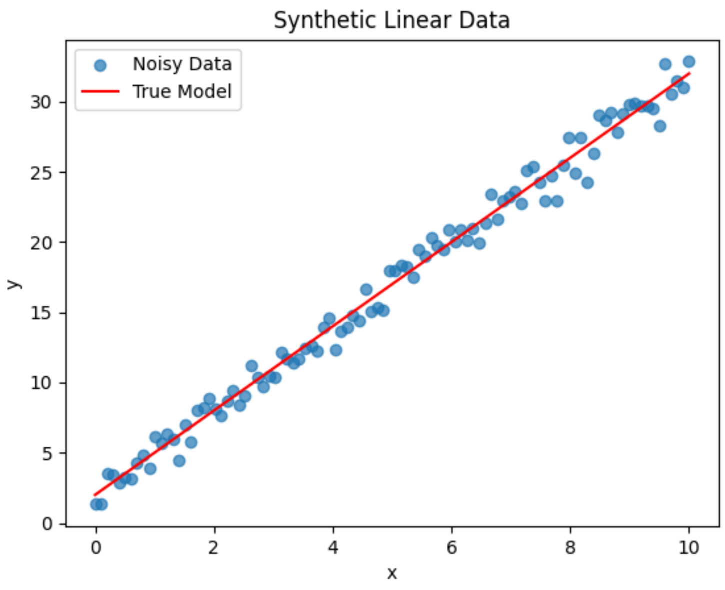

Figure 1: Synthetic Linear Data

Step 2: Split the Data

To evaluate our model's performance, we split the dataset into training and testing subsets. Typically, a common split ratio is 80% for training and 20% for testing.

Python Code: Split Data

```python
from sklearn.model_selection import train_test_split

# Reshape x for compatibility
x = x.reshape(-1, 1)
y_noisy = y_noisy.reshape(-1, 1)

# Split the data
X_train, X_test, y_train, y_test = train_test_split(
    x, y_noisy, test_size=0.2, random_state=7976
)
```

Step 3: Define the Model and Gradient Descent

We define the linear regression model $y = \theta_0 + \theta_1 x$ and implement the gradient descent algorithm to optimize the model parameters θ_0 and θ_1.

Loss Function The Mean Squared Error (MSE) is used as the loss function:

$$L(\theta_0, \theta_1) = \frac{1}{2n} \sum_{i=1}^{n} (y_i - (\theta_0 + \theta_1 x_i))^2 ,$$

where n is the number of training data points.

Gradients The gradients of the loss function with respect to θ_0 and θ_1 are:

$$\frac{\partial L}{\partial \theta_0} = -\frac{1}{n} \sum_{i=1}^{n} \left(y_i - (\theta_0 + \theta_1 x_i) \right),$$

$$\frac{\partial L}{\partial \theta_1} = -\frac{1}{n} \sum_{i=1}^{n} x_i \left(y_i - (\theta_0 + \theta_1 x_i) \right).$$

Parameter Updates Using gradient descent, the parameters are updated iteratively:

$$\theta_0^{(t+1)} = \theta_0^{(t)} - \eta \frac{\partial L}{\partial \theta_0},$$

$$\theta_1^{(t+1)} = \theta_1^{(t)} - \eta \frac{\partial L}{\partial \theta_1},$$

where η is the learning rate.

Step 4: Train the Model Using Gradient Descent

We implement the gradient descent algorithm to optimize the model parameters based on the training data.

Python Code: Gradient Descent Implementation

```python
# Initialize parameters
theta_0 = 1.0  # Intercept
theta_1 = 1.0  # Slope
learning_rate = 0.01
iterations = 1500

# Number of training samples
n_train = len(X_train)

# Lists to store parameter values
theta_0_values = []
theta_1_values = []


# Gradient Descent
for _ in range(iterations):
    y_pred = theta_0 + theta_1 * X_train.flatten()  # Predicted
        values
    error = y_train.flatten() - y_pred  # Residuals
    # Compute gradients
    grad_theta_0 = -np.sum(error) / n_train
    grad_theta_1 = -np.sum(error * X_train.flatten()) / n_train
    # Update parameters
    theta_0 -= learning_rate * grad_theta_0
    theta_1 -= learning_rate * grad_theta_1

    # Store parameter values
    theta_0_values.append(theta_0)
    theta_1_values.append(theta_1)

print(f"Estimated Parameters using Gradient Descent: theta_0 = {
    theta_0:.2f}, theta_1 = {theta_1:.2f}")
```

After running the gradient descent algorithm, we obtain the estimated parameters:

$$\theta_0 \approx 1.98 \quad \theta_1 \approx 3.00$$

These estimates are close to the true parameters ($\theta_0 = 2.0$, $\theta_1 = 3.0$), indicating that the gradient descent successfully approximated the underlying model. We can also visualize how the paraeters have ben learnt through the process.

Python Code: Visualization: Learning of Parameters

```python
# Plot parameter convergence
plt.figure(figsize=(10, 5))
plt.plot(range(iterations), theta_0_values, label='theta_0')
plt.plot(range(iterations), theta_1_values, label='theta_1')
plt.axhline(y=theta_0_true, linestyle='--', color='k', label='True
    theta_0')
plt.axhline(y=theta_1_true, linestyle='--', color='m', label='True
    theta_1')
plt.xlabel('Iteration')
plt.ylabel('Parameter Value')
plt.title('Convergence of Parameters')
plt.legend()
plt.savefig('images/lin_grad_descent.png')
plt.show()
```

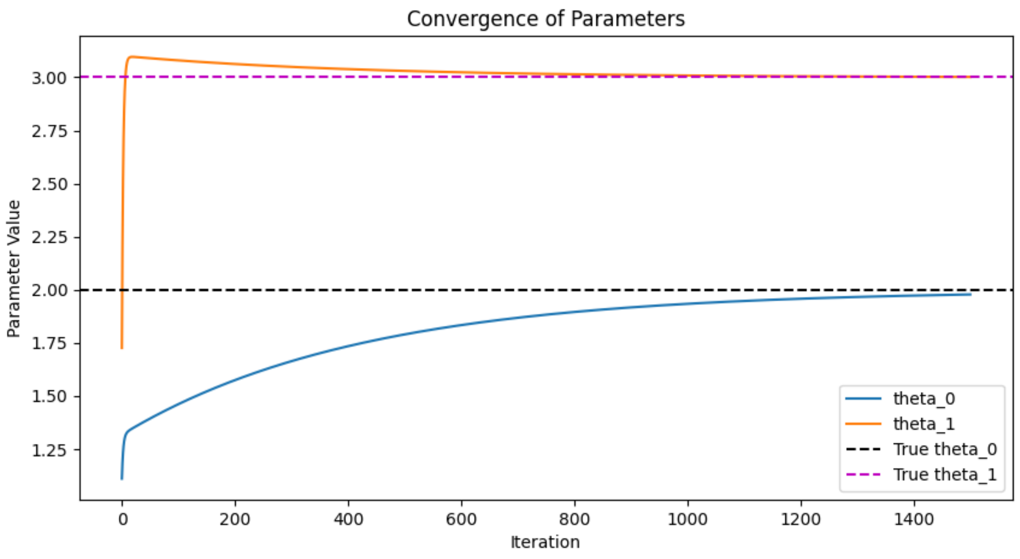

Figure 2: Parameter Evolution During Gradient Descent

Step 5: Evaluate Model Performance on Test Set

We assess the model's performance by computing the Mean Squared Error (MSE) and the r^2 score on the test set. More precisely, we compute these metrics throughout the learning process to observe how they converge toward optimal values. This type of graph helps determine whether the learning process is still ongoing or has *converged*.

Python Code: Model Evaluation

```python
def  mean_squared_error(y_true, y_pred):
    y_true = np.array(y_true)
    y_pred = np.array(y_pred)

    squared_errors = (y_true - y_pred) ** 2
    mse = np.mean(squared_errors)
    return mse

from sklearn.metrics import r2_score


iterations = len(theta_0_values)
mse_train_values = []
r2_train_values = []
mse_test_values = []
r2_test_values = []

for theta_0, theta_1 in zip( theta_0_values, theta_1_values):
    y_train_pred = theta_0 + theta_1 * X_train.flatten()
    y_test_pred = theta_0 + theta_1 * X_test.flatten()

    mse_train = mean_squared_error(y_train, y_train_pred)
    r2_train = r2_score(y_train, y_train_pred)
    mse_test = mean_squared_error(y_test, y_test_pred)
    r2_test = r2_score(y_test, y_test_pred)

    mse_train_values.append(mse_train)
    r2_train_values.append(r2_train)
    mse_test_values.append(mse_test)
    r2_test_values.append(r2_test)

print(f'MSE train final: {mse_train:.2f}', end='\n')
print(f'MSE test final: {mse_test:.2f}', end='\n\n')
print(f'r2 train final: {r2_train:.3f}', end='\n')
print(f'r2 train final: {r2_test:.3f}')
```

After running the gradient descent algorithm, we obtain the estimated parameters:

MSE (train, final):	148.51
MSE (test, final):	176.59
R^2 (train, final):	0.989
R^2 (test, final):	0.990

We can also visualize the evolution of the performance during the training.

Python Code: Visualization Evolution Performance Metrics

```python

warm_up = 5 # we do not show the first values, certainly too high.
plt.figure(figsize=(14, 6))

# --- Subplot for Mean Squared Error (MSE) ---
plt.subplot(1, 2, 1)
plt.plot(range(warm_up, iterations), mse_train_values[warm_up:],
    label='Train MSE', color='blue')
plt.plot(range(warm_up, iterations), mse_test_values[warm_up:],
    label='Test MSE', color='orange')
plt.xlabel('Iteration')
plt.ylabel('Mean Squared Error')
plt.title('MSE Over Iterations')
plt.legend()
plt.grid(True)

# --- Subplot for r2 Score ---
plt.subplot(1, 2, 2)
plt.plot(range(warm_up, iterations), r2_train_values[warm_up:],
    label='Train r2', color='green')
plt.plot(range(warm_up, iterations), r2_test_values[warm_up:],
    label='Test r2', color='red')
plt.xlabel('Iteration')
plt.ylabel('r2 Score')
plt.title('r2 Score Over Iterations')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.savefig('images/lin_evolution_performance.png')
plt.show()
```

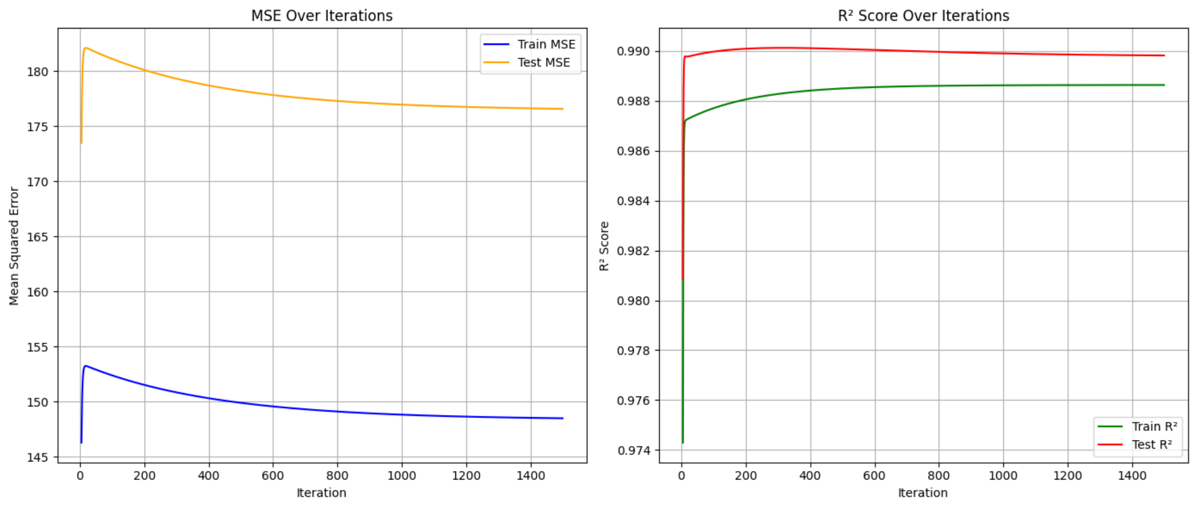

Figure 3: Performance Evolution During Gradient Descent

We could even visualize the training path within the parameter space, provided the dimensionality is low enough, as it is in this case.

Python Code: Visualization Evolution Performance Metrics

```python

# Plot the path in parameter space
plt.figure(figsize=(5, 4))
plt.plot(theta_0_values, theta_1_values, 'o-', markersize=3, label
    ="Parameter Path")
plt.scatter([theta_0_values[0]], [theta_1_values[0]], color='red',
     label="Start Point", zorder=5)
plt.scatter([theta_0_values[-1]], [theta_1_values[-1]], color='
    green', label="End Point (Optimal)", zorder=5)
plt.xlabel(r"$\theta_0$ (Intercept)")
plt.ylabel(r"$\theta_1$ (Slope)")
plt.title("Gradient Descent Path in Parameter Space")
plt.legend()
plt.grid()
plt.savefig('images/lin_path_param.png')
plt.show()
```

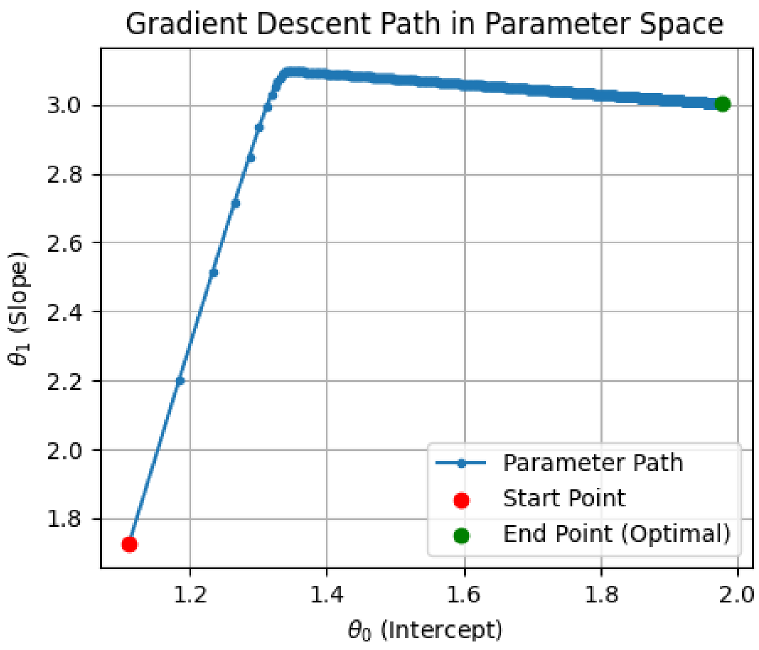

Figure 4: Performance Evolution During Gradient Descent

Step 6: Compare with Python Library Implementation

For comparison, we utilize the `LinearRegression` class from scikit-learn to perform linear regression and evaluate its performance.

Python Code: Scikit-Learn Comparison

```python
from sklearn.linear_model import LinearRegression

# Initialize and train the model
model = LinearRegression()
model.fit(X_train, y_train)

# Estimated parameters
theta_0_sklearn = model.intercept_[0]
theta_1_sklearn = model.coef_[0][0]

# Predictions on test set
y_test_pred_sklearn = model.predict(X_test)
mse_test_sklearn = mean_squared_error(y_test, y_test_pred_sklearn)


print("=== sklearn Values ===")
print(f"Estimated Parameters using Scikit-Learn: theta_0 = {
    theta_0_sklearn:.2f}, theta_1 = {theta_1_sklearn:.2f}")
print(f"Test Set MSE using Scikit-Learn: {mse_test_sklearn:.2f}")


print("\n=== Gradient Descent Values ===")
print(f"Estimated Parameters using Gradient Descent: theta_0 = {
    theta_0:.2f}, theta_1 = {theta_1:.2f}")
mse_test_gradient_descent =mse_test_values[-1]
print(f"Test Set MSE using Gradient Descent: {
    mse_test_gradient_descent:.2f}")
```

=== sklearn Values ===
Estimated Parameters using Scikit-Learn: $\theta_0 = 2.00, \theta_1 = 3.00$
Test Set MSE using Scikit-Learn: 0.92

=== Gradient Descent Values ===
Estimated Parameters using Gradient Descent: $\theta_0 = 1.98, \theta_1 = 3.00$
Test Set MSE using Gradient Descent: 176.59

Step 7: Visualize the Fitted Models

Finally, we visualize the fitted models from both implementations against the test data.

Python Code: Visualization

```python
# Plot the test data
plt.scatter(X_test, y_test, label="Test Data", alpha=0.7)

# Plot gradient descent fitted model
plt.plot(X_test, y_test_pred, 'g-', label="Gradient Descent Fit")

# Plot scikit-learn fitted model
plt.plot(X_test, y_test_pred_sklearn, 'b--', label="Scikit-Learn
    Fit")

# Plot true model
plt.plot(x, y_true, 'r--', label="True Model")

plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.title("Comparison of Linear Regression Models")
plt.savefig('images/lin_comp_models.png')
plt.show()
```

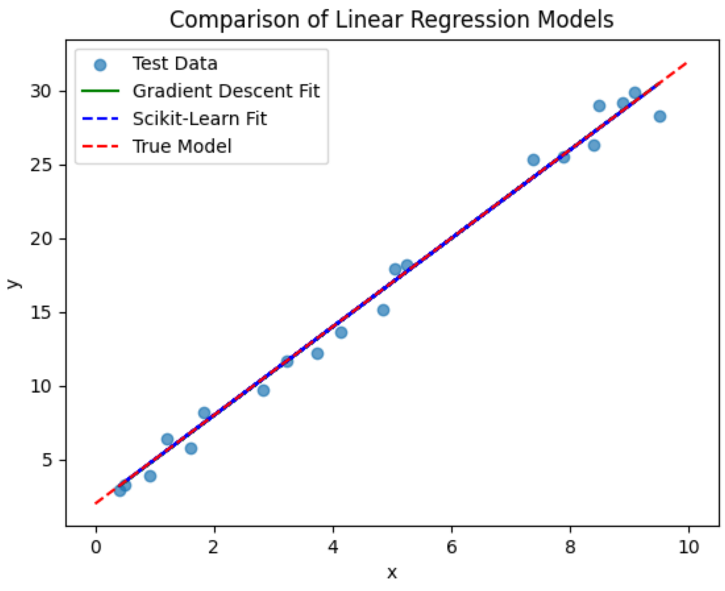

Figure 5: Comparison of Gradient Descent and Scikit-Learn Linear Regression Fits

The visualization confirms that both the gradient descent implementation and the scikit-learn model closely align with the true underlying linear relationship, validating the effectiveness of both approaches.

Step 8. Residual Analysis

Let's have a look at the residuals. We examine the residuals of the training dataset to check whether the implicit assumptions made during model training are valid. We can add some quick tests, such as identifying residuals greater than 3 standard deviations, to flag potential outliers. Note, however, that such values are not automatically outliers, as approximately 0.3% of normally distributed data is expected to fall outside this range.

Python Code: Visualization

```python
import seaborn as sns

# Compute residuals
residuals = y_train.flatten() - y_train_pred.flatten()

# Plot histogram of residuals using seaborn
plt.figure(figsize=(8, 5))
sns.histplot(residuals, kde=True, color='blue', edgecolor='k',
    alpha=0.7)
plt.title('Histogram of Residuals')
plt.xlabel('Residual')
plt.ylabel('Frequency')
plt.axvline(0, color='red', linestyle='--', linewidth=1)  #
    Vertical line at zero
plt.grid(True)
plt.savefig('images/lin_residuals.png')
plt.show()

# Identify potential outliers (e.g., residuals outside 3 standard
    deviations)
residual_std = np.std(residuals)
outliers = np.where(np.abs(residuals) > 3 * residual_std)

# Print information about outliers
if len(outliers[0]) > 0:
    print("Potential outliers detected at indices:", outliers[0])
    print("Residual values of outliers:", residuals[outliers])
else:
    print("No significant outliers detected.")
```

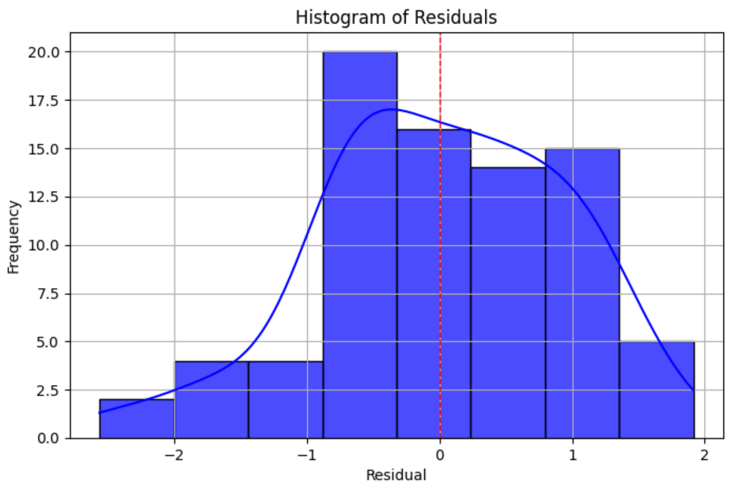

Figure 6: Residuals for the training data

Conclusion

This illustrative example showcases the complete workflow of performing simple linear regression:

1. Generating synthetic data with a known linear relationship.
2. Splitting the data into training and testing sets.
3. Implementing linear regression using gradient descent, including defining the loss function and computing gradients.
4. Training the model and evaluating its performance on unseen data.
5. Comparing the custom implementation with a standard library to validate results.

1.8 Illustrative Example 2: Nonlinear Regression - Modeling a Damped Harmonic Oscillator

In this example, we demonstrate how to model a damped harmonic oscillator (damped pendulum) using *nonlinear regression* to predict the displacement $y(t)$ over time t. Unlike linear regression, which fits a straight line to data, nonlinear regression allows us to fit more complex models that capture intricate relationships within the data. The damped harmonic oscillator is governed by the equation:

$$y(t) = Ae^{-\gamma t}\cos(\omega t),$$

where:

- A is the initial amplitude,
- γ is the damping coefficient,
- ω is the angular frequency.

It is important to note that this is not truly a time series. Instead, we are creating different pulses, and for each pulse, we measure a different data point (t, y). In this way, the data does not represent a continuous temporal sequence but rather independent observations for different pulses of the system.

We will generate synthetic data based on this known damped oscillation equation, split the data into training and testing sets, implement nonlinear regression using gradient descent to estimate the model parameters, evaluate the model's performance, and compare the results with those obtained using a Python library.

Generate Synthetic Data

We begin by generating a dataset that follows the damped harmonic oscillator equation with added noise to simulate real-world observations. This synthetic data serves as the ground truth for our regression analysis.

Python Code: Generate Synthetic Data

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

# True parameters
A_true = 5.0      # Initial amplitude
gamma_true = 0.5  # Damping coefficient
omega_true = 2.0  # Angular frequency

# Generate time values
np.random.seed(7976)  # For reproducibility
t = np.linspace(0, 10, 500)  # Time values
noise = np.random.normal(scale=0.3, size=len(t))  # Add some noise

# Generate data
y_true = A_true * np.exp(-gamma_true * t) * np.cos(omega_true * t)
y_noisy = y_true + noise

# Visualize the generated data
plt.figure(figsize=(10, 5))
plt.plot(t, y_noisy, 'o', markersize=2, label='Noisy Data')
plt.plot(t, y_true, 'r-', linewidth=2, label='True Model')
plt.xlabel('Time (t)')
plt.ylabel('Displacement (y)')
plt.legend()
plt.title('Synthetic Data from the Damped Harmonic Oscillator')
plt.savefig('images/damped_oscillator.png', dpi=300, bbox_inches='
    tight')
plt.show()
```

The plot below illustrates the generated noisy displacement data points alongside the true damped oscillation model:

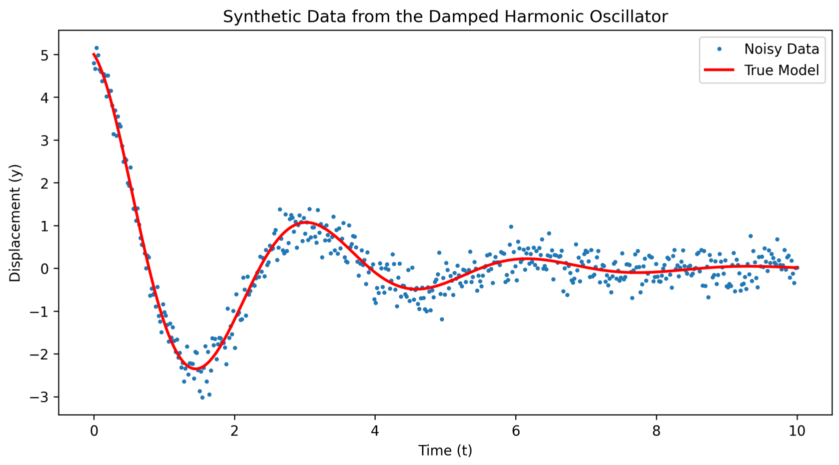

Figure 7: Synthetic Data from the Damped Harmonic Oscillator

Split the Data

To evaluate our model's performance, we split the dataset into training and testing subsets. A common split ratio is 80% for training and 20% for testing.

Python Code: Split Data

```python
from sklearn.model_selection import train_test_split

# Reshape x for compatibility
t = t.reshape(-1, 1)
y_noisy = y_noisy.reshape(-1, 1)

# Split the data
T_train, T_test, y_train, y_test = train_test_split(
    t, y_noisy, test_size=0.2, random_state=7976
)
```

Define the Model and Gradient Descent

We define the nonlinear regression model based on the damped harmonic oscillator equation and implement the gradient descent algorithm to optimize the model parameters A, γ, and ω.

Model Equation The model is defined as:

$$y(t) = Ae^{-\gamma t}\cos(\omega t),$$

where A, γ, and ω are the parameters to be estimated.

Loss Function We use the Mean Squared Error (MSE) as the loss function:

$$L(A, \gamma, \omega) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - Ae^{-\gamma t_i}\cos(\omega t_i)\right)^2,$$

where n is the number of training data points.

Gradients The gradients of the loss function with respect to each parameter are derived as follows:

$$\frac{\partial L}{\partial A} = -\frac{2}{n}\sum_{i=1}^{n}\left(y_i - Ae^{-\gamma t_i}\cos(\omega t_i)\right)e^{-\gamma t_i}\cos(\omega t_i),$$

$$\frac{\partial L}{\partial \gamma} = \frac{2}{n}\sum_{i=1}^{n}\left(y_i - Ae^{-\gamma t_i}\cos(\omega t_i)\right)At_i e^{-\gamma t_i}\cos(\omega t_i),$$

$$\frac{\partial L}{\partial \omega} = \frac{2}{n}\sum_{i=1}^{n}\left(y_i - Ae^{-\gamma t_i}\cos(\omega t_i)\right)Ae^{-\gamma t_i}\sin(\omega t_i)t_i.$$

Train the Model Using Gradient Descent

We implement the gradient descent algorithm to optimize the parameters A, γ, and ω based on the training data. In this example, we compute the performance metrics (MSE and r2) directly within the same loop. First, we initialize all our variables and create a function to update the parameters. The mathematical logic of gradient descent is fully encapsulated within this function.

Python Code: Gradient Descent Implementation. Function definition

```
from sklearn.metrics import mean_squared_error, r2_score

# Initialize Parameters
A = 1.0            # Initial guess for amplitude
gamma = 0.1        # Initial guess for damping coefficient
omega = 1.0        # Initial guess for angular frequency
learning_rate = 1e-3
iterations = 10000

n_train = len(T_train)

# Initialize Lists to Store Values
A_values = []
gamma_values = []
omega_values = []
mse_train_values = []
r2_train_values = []
mse_test_values = []
r2_test_values = []

def update_values(A,gamma, omega):
    # Compute predictions on training set
    y_train_pred = A * np.exp(-gamma * T_train.flatten()) * np.cos
        (omega * T_train.flatten())

    # Compute residuals
    error = y_train.flatten() - y_train_pred

    # Compute gradients
    grad_A = (-2 / n_train) * np.sum(error * np.exp(-gamma *
        T_train.flatten()) * np.cos(omega * T_train.flatten()))
    grad_gamma = (2 / n_train) * np.sum(error * A * T_train.
        flatten() * np.exp(-gamma * T_train.flatten()) * np.cos(
        omega * T_train.flatten()))
    grad_omega = (2 / n_train) * np.sum(error * A * T_train.
        flatten() * np.exp(-gamma * T_train.flatten()) * np.sin(
        omega * T_train.flatten()))

    # Update parameters
    new_A =A - learning_rate * grad_A
    new_gamma = gamma - learning_rate * grad_gamma
    new_omega =omega - learning_rate * grad_omega

    return new_A, new_gamma, new_omega
```

Then, we perform the loop using our custom function.

Python Code: Gradient Descent Implementation; main loop

```python
# Gradient Descent Loop
for i in range(iterations):

    # Update parameters
    A, gamma, omega =  update_values(A,gamma, omega)

    A_values.append(A)
    gamma_values.append(gamma)
    omega_values.append(omega)


    # Training set metrics
    mse_train = mean_squared_error(y_train, y_train_pred)
    r2_train = r2_score(y_train, y_train_pred)
    mse_train_values.append(mse_train)
    r2_train_values.append(r2_train)

    # Test set predictions and metrics
    y_test_pred = A * np.exp(-gamma * T_test.flatten()) * np.cos(
        omega * T_test.flatten())
    mse_test = mean_squared_error(y_test, y_test_pred)
    r2_test = r2_score(y_test, y_test_pred)
    mse_test_values.append(mse_test)
    r2_test_values.append(r2_test)


print(f"\nEstimated Parameters using Gradient Descent:\n"
      f"A = {A:.2f}, gamma = {gamma:.2f}, omega = {omega:.2f}")
```

After running the gradient descent algorithm, we obtain the estimated parameters:

$$A \approx 3.09, \quad \gamma \approx 0.33, \quad \omega \approx 1.99$$

These estimates are not that close to the true parameters ($A = 5.0$, $\gamma = 0.5$, $\omega = 2.0$), indicating that the gradient descent didnt successfully approximated the underlying model. Let's visuallizaee the performance path to see if we can understabnd why

Python Code: Model Evaluation

```python
import matplotlib.pyplot as plt

# Create a figure with three subplots
fig, axes = plt.subplots(1, 3, figsize=(15, 5))  # 3 rows, 1 column

# --- Subplot 1: Parameter Convergence ---
axes[0].plot(range(iterations), A_values, label='A', color='blue')
axes[0].plot(range(iterations), gamma_values, label='gamma', color='orange')
axes[0].plot(range(iterations), omega_values, label='omega', color='green')
axes[0].axhline(y=A_true, linestyle='--', color='blue', label='True A')
axes[0].axhline(y=gamma_true, linestyle='--', color='orange', label='True gamma')
axes[0].axhline(y=omega_true, linestyle='--', color='green', label='True omega')
axes[0].set_xlabel('Iteration')
axes[0].set_ylabel('Parameter Value')
axes[0].set_title('Convergence of Parameters')
axes[0].legend()
axes[0].grid(True)

# --- Subplot 2: Mean Squared Error (MSE) ---
axes[1].plot(range(warm_up, iterations), mse_train_values[warm_up:], label='Train MSE', color='blue')
axes[1].plot(range(warm_up, iterations), mse_test_values[warm_up:], label='Test MSE', color='orange')
axes[1].axhline(y=mse_train_true, linestyle='--', color='blue', label='True Train MSE')
axes[1].axhline(y=mse_test_true, linestyle='--', color='orange', label='True Test MSE')
axes[1].set_xlabel('Iteration')
axes[1].set_ylabel('Mean Squared Error')
axes[1].set_title('MSE Over Iterations')
axes[1].legend()
axes[1].grid(True)

# --- Subplot 3: r2 Score ---
axes[2].plot(range(warm_up, iterations), r2_train_values[warm_up:], label='Train r2', color='green')
axes[2].plot(range(warm_up, iterations), r2_test_values[warm_up:], label='Test r2, color='red')
axes[2].axhline(y=r2_train_true, linestyle='--', color='green', label='True Train r2')
axes[2].axhline(y=r2_test_true, linestyle='--', color='red', label='True Test r2')
axes[2].set_xlabel('Iteration')
axes[2].set_ylabel('r2 Score')
axes[2].set_title('r2 Score Over Iterations')
axes[2].legend()
axes[2].grid(True)

# Adjust layout and save the figure
plt.tight_layout()
plt.savefig('images/nonlin_grad_descent.png')
plt.show()
```

Giving the following plot

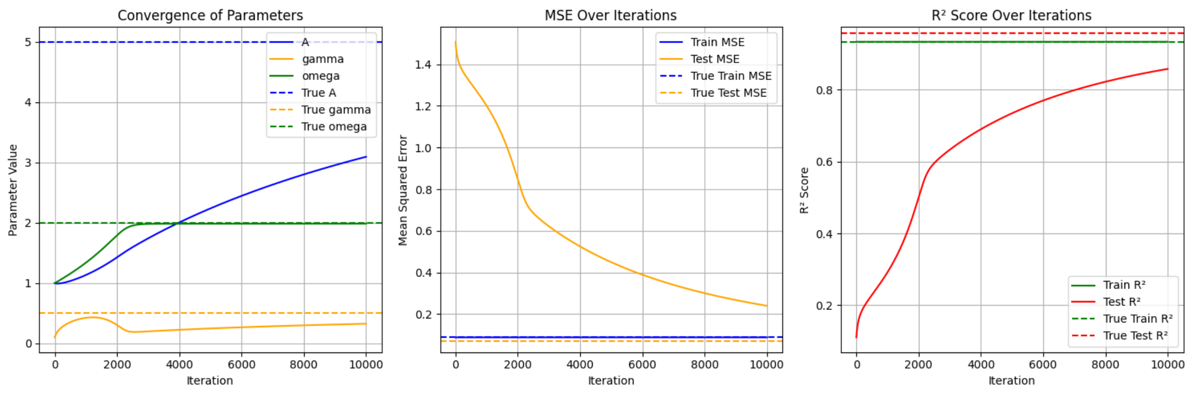

Figure 8: Gradient Descent Training Path

From these plots, we can observe that the training is not yet finished, as the MSE is still decreasing and the r2 is still increasing. Since we know the true values, we can also see that the parameters have not yet reached a plateau point, which indicates that more iterations are needed, or perhaps the learning rate was set too low. In this case, we would need to go back to our training and adjust the learning rate and the number of iterations.

It is left as an exercise to find better values and verify whether the model learns the correct parameters.

Visualize the Fitted Models

Finally, we visualize the fitted models from both implementations against the test data.

Python Code: Visualization

```python
# Optimized parameters obtained from gradient descent
A = A_values[-1]
gamma = gamma_values[-1]
omega = omega_values[-1]


y_test_pred = A * np.exp(-gamma * T_test.flatten()) * np.cos(omega
     * T_test.flatten())

mse_test = mean_squared_error(y_test, y_test_pred)
r2_test = r2_score(y_test, y_test_pred)

print(f"Test Set MSE: {mse_test:.4f}")
print(f"Test Set r2: {r2_test:.4f}")


# Visualization: True Model, Test Data, and Predicted Model

plt.figure(figsize=(12, 6))

# Plot True Model
plt.plot(t, A_true * np.exp(-gamma_true * t) * np.cos(omega_true *
     t),
         'r-', linewidth=2, label='True Model')

# Plot Test Data
plt.scatter(T_test, y_test, color='blue', alpha=0.5, label='Test
    Data')

# Sort X_test for a smooth predicted line
sorted_indices = T_test.flatten().argsort()
T_test_sorted = T_test.flatten()[sorted_indices]
y_test_pred_sorted = y_test_pred[sorted_indices]

# Plot Predicted Model
plt.plot(T_test_sorted, y_test_pred_sorted, 'g-', linewidth=2,
    label='Predicted Model')

# Labels and Title
plt.xlabel('Time (t)')
plt.ylabel('Displacement (y)')
plt.title('Damped Harmonic Oscillator: True Model vs Predicted
    Model on Test Data')

# Legend
plt.legend()

# Grid
plt.grid(True)

# Save the plot
plt.savefig('images/damped_oscillator_prediction.png', dpi=300,
    bbox_inches='tight')

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

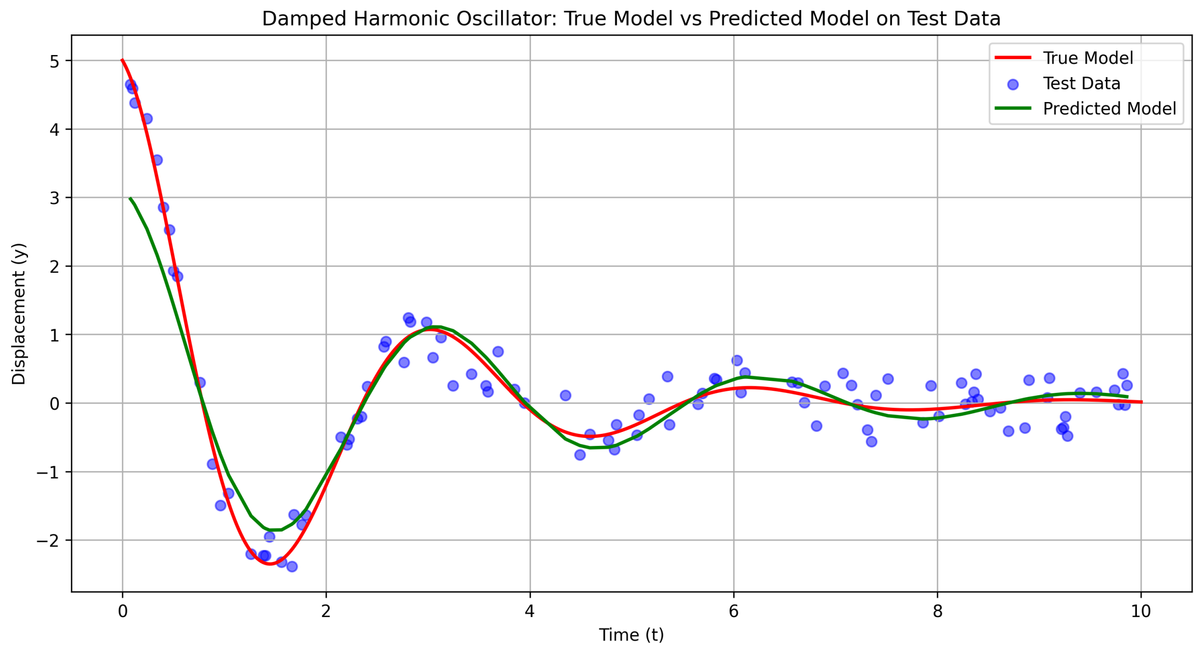

Figure 9: True vs predicted model

Note that this visualization confirms what we found earlier: the training is not finished, and the parameters of the model are not yet optimal.

It is left as an exercise to reproduce this plot once the optimal parameters of the model have been approximated more precisely.

Compare with Python Library Implementation

Since this model is non linear, we cannot use the `LinearRegression` function from the sklearn.model_selection package. However, in python, we can use the package scipy for silmilar nonlinear fitting (withouth knowing the details). Below is the code to approach the fiitting using that library.

Python Code: Visualization

```python
from scipy.optimize import curve_fit

# Define the nonlinear model (e.g., damped cosine)
def nonlinear_model(t, A, gamma, omega):
    return A * np.exp(-gamma * t) * np.cos(omega * t)

# Fit the model to training data
initial_guess = [1.0, 0.1, 1.0]  # Initial parameter guesses
optimized_params, covariance_matrix = curve_fit(
    nonlinear_model, T_train.flatten(), y_train.flatten(), p0=
        initial_guess
)

# Extract optimized parameters
A_opt, gamma_opt, omega_opt = optimized_params
print(f"Optimized Parameters:\nA = {A_opt:.4f}, gamma = {gamma_opt
    :.4f}, omega = {omega_opt:.4f}")

# Evaluate the model on the training and test sets
y_train_pred = nonlinear_model(T_train.flatten(), A_opt, gamma_opt
    , omega_opt)
y_test_pred = nonlinear_model(T_test.flatten(), A_opt, gamma_opt,
    omega_opt)

# Compute performance metrics
mse_train = mean_squared_error(y_train, y_train_pred)
r2_train = r2_score(y_train, y_train_pred)
mse_test = mean_squared_error(y_test, y_test_pred)
r2_test = r2_score(y_test, y_test_pred)

print(f"\nPerformance Metrics:")
print(f"Train Set: MSE = {mse_train:.4f}, r2 = {r2_train:.4f}")
print(f"Test Set:  MSE = {mse_test:.4f}, r2 = {r2_test:.4f}")
```

with results

$$A \approx 4.9868, \quad \gamma \approx 0.4996, \quad \omega \approx 1.9910$$

A Detailed Derivation of the Normal Equation

The Normal Equation provides a closed-form solution for finding the optimal coefficients in linear regression by minimizing the Mean Squared Error (MSE) between the predicted values and the actual values. This derivation leverages matrix algebra to efficiently compute these coefficients under the assumption of no multicollinearity among predictors.

In linear regression, the goal is to find the coefficient vector β that minimizes the MSE between the predicted values $\hat{y}$ and the actual values y. The MSE is defined as:

$$\text{MSE}(\beta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - \mathbf{x}_i^T\beta)^2$$

Where:

- n is the number of observations.
- y_i is the actual value for the i-th observation.
- $\mathbf{x}_i$ is the feature vector (including a bias term) for the i-th observation.
- β is the vector of coefficients to be estimated.
- $\hat{y}_i = \mathbf{x}_i^T\beta$ is the predicted value.

For computational efficiency, the MSE can be expressed in matrix notation:

$$\text{MSE}(\beta) = \frac{1}{n}(y - X\beta)^T(y - X\beta)$$

Where:

- y is an $n \times 1$ vector of actual values.
- X is an $n \times p$ design matrix where each row represents an observation and each column represents a feature (including the intercept).
- β is a $p \times 1$ vector of coefficients.

Expanding the quadratic form:

$$\text{MSE}(\beta) = \frac{1}{n}\left(y^Ty - y^TX\beta - \beta^TX^Ty + \beta^TX^TX\beta\right)$$

Since $y^TX\beta$ is a scalar, it is equal to its transpose:

$$y^TX\beta = \beta^TX^Ty$$

Thus, the expression simplifies to:

$$\text{MSE}(\beta) = \frac{1}{n}\left(y^Ty - 2\beta^TX^Ty + \beta^TX^TX\beta\right)$$

To find the minimum of the MSE, we take the gradient of the MSE with respect to β and set it to zero:

$$\nabla_\beta\text{MSE}(\beta) = \frac{1}{n}\left(-2X^Ty + 2X^TX\beta\right)$$

Simplifying:

$$\nabla_\beta \text{MSE}(\beta) = \frac{2}{n}\left(X^T X\beta - X^T y\right)$$

Setting the gradient equal to zero for minimization:

$$\frac{2}{n}\left(X^T X\beta - X^T y\right) = 0$$

Multiplying both sides by $\frac{n}{2}$ to eliminate the constant:

$$X^T X\beta - X^T y = 0$$

Rearranging the terms:

$$X^T X\beta = X^T y$$

Assuming that $X^T X$ is invertible (i.e., the design matrix X has full rank and there is no perfect multicollinearity), we can solve for β by multiplying both sides by $(X^T X)^{-1}$:

$$\beta = (X^T X)^{-1} X^T y$$

Thus, the Normal Equation is derived as:

$$\boxed{\hat{\beta} = (X^T X)^{-1} X^T y}$$

Interpretation of the Normal Equation

- $X^T X$: This matrix captures the covariance between the features. Its invertibility ensures that the features are not perfectly correlated.
- $X^T y$: This vector represents the covariance between the features and the target variable.
- $(X^T X)^{-1} X^T y$: This expression projects the target variable y onto the column space of X, yielding the best linear unbiased estimator (BLUE) for β under the Gaussian assumption.