Tab 2

### **Overfitting vs. Underfitting in Machine Learning**

#### **1. Overfitting**

Overfitting occurs when a machine learning model learns the training data too well, capturing noise and details that do not generalize to new data. It happens when the model is too complex relative to the dataset, leading to high accuracy on the training set but poor performance on unseen data.

* **Causes:**
  + Too many parameters (high complexity)**[learn: feature engineering (PCA)]**
  + Small training dataset
  + Too many training epochs (excessive learning)
  + Lack of regularization
* **Signs:**
  + Low training error but high test error
  + Model predicts training data perfectly but fails on new data
* **Solutions:**
  + Collect more training data

###### Use regularization techniques (L1, L2 regularization, dropout)

* + **Reduce model complexity (fewer layers or neurons)**
  + **Use cross-validation (e.g., k-fold cross-validation)**

#### **2. Underfitting**

Underfitting occurs when a model is too simple to capture the underlying pattern in the data, leading to poor performance on both training and test data. It means the model has not learned enough from the training data.

* **Causes:**
  + Too simple model (not enough parameters)
  + Insufficient training time
  + Poor feature selection or preprocessing
* **Signs:**
  + High training and test error
  + Model performs poorly even on training data
* **Solutions:**
  + Use a more complex model
  + Train for more epochs
  + Feature engineering (add relevant features)
  + Reduce regularization if applied excessively

### **Balance Between Overfitting and Underfitting**

A good model finds a balance between underfitting and overfitting. This is called **"generalization"**, where the model performs well on both training and unseen data.

Would you like an example to illustrate this better?

### **What is Generalization in Machine Learning?**

**Generalization** refers to a model's ability to perform well on **new, unseen data** after being trained on a dataset. A well-generalized model **captures the underlying patterns** in the training data without memorizing it, allowing it to make accurate predictions on test data.

### **When is a Model Considered Generalized?**

A model is **generalized** when:  
 ✅ **Low Training Error & Low Test Error** → The model performs well on both training and unseen data.  
 ✅ **Avoids Overfitting** → It does not memorize noise or outliers from training data.  
 ✅ **Avoids Underfitting** → It has learned enough complexity to understand real patterns.  
 ✅ **Performs Well on Diverse Inputs** → It can handle variations in real-world data without breaking down.

### **How to Improve Generalization?**

To make a model more generalized, you can:

1️⃣ **Use More Training Data** → Helps the model learn diverse patterns.  
 2️⃣ **Use Cross-Validation** → Ensures the model works well on different data splits.  
 3️⃣ **Apply Regularization** → Techniques like L1, L2 (Ridge, Lasso), and Dropout prevent overfitting.  
 4️⃣ **Use Early Stopping** → Stops training before the model overfits.  
 5️⃣ **Feature Engineering** → Select relevant features to avoid unnecessary complexity.  
 6️⃣ **Ensemble Learning** → Combining multiple models (e.g., bagging, boosting) improves stability.

Would you like a code example demonstrating generalization? 🚀

### **Example: Overfitting vs. Underfitting in Machine Learning**

Let’s consider a simple **house price prediction** problem. You have a dataset with features like:

* **Size of the house (sq ft)**
* **Number of bedrooms**
* **Location**
* **Age of the house**

The goal is to predict the house price based on these features.

### **1️⃣ Underfitting Example**

#### **Model: Linear Regression (Too Simple)**

You train a basic **linear regression** model that assumes house prices increase linearly with size and number of bedrooms. However, real estate prices are influenced by more complex factors (e.g., neighborhood, market trends).

**Effect:**

* The model cannot capture important patterns.
* Both training and test errors are high.
* Example: It predicts all house prices as a straight-line trend, ignoring actual fluctuations.

**Fix:**

* Use a more complex model (e.g., decision trees, neural networks).
* Add relevant features (e.g., crime rate, distance to city center).

### **2️⃣ Overfitting Example**

#### **Model: Deep Neural Network with Too Many Layers**

Now, you train a **deep neural network with 10 hidden layers** and thousands of neurons. It learns every detail of the training data—including noise and outliers.

**Effect:**

* The model memorizes training data perfectly (very low training error).
* Performs poorly on new test data because it learned unnecessary details.
* Example: It might assign too much weight to random variations in house price (e.g., a house was sold at a high price due to a rare event).

**Fix:**

* Use **regularization techniques** (L1, L2, dropout).
* Reduce model complexity (fewer layers or neurons).
* Increase training data size.

### **3️⃣ Balanced Model (Generalization)**

A well-balanced model (e.g., a decision tree with controlled depth or a neural network with dropout) would:

* Learn meaningful patterns without memorizing noise.
* Have low training **and** test error.
* Provide good predictions for both seen and unseen houses.

### **Visual Representation**

If we were to plot the data and model predictions:

* **Underfitting** → A straight line that ignores variations.
* **Overfitting** → A wiggly, complex curve that captures every noise point.
* **Good Fit** → A smooth curve that follows the general pattern without excessive fluctuations.

Would you like a code example in Python to see this in action? 🚀

The number of training parameters (i.e., model parameters) and the amount of training data required to generalize a model depend on several factors, including model complexity, task difficulty, and data quality. However, some general guidelines can be followed:

### **1. Minimum Data Requirement (Rule of Thumb)**

* **Traditional Machine Learning Models:** Typically require **10× to 30×** the number of parameters as training samples.
* **Deep Learning Models:** Often require at least **50× to 100×** the number of parameters.
* **Large Language Models (LLMs):** The ratio is much higher, often requiring billions of training examples.

### **2. Maximum Data Requirement**

* There is no strict upper limit, but practical constraints include computational power and diminishing returns. At some point, adding more data does not improve generalization significantly.
* **Scaling laws (like Chinchilla scaling laws)** suggest that for deep learning models, optimal performance is achieved by balancing the number of parameters and data points.

### **3. Example Guidelines for Different Model Types**

| **Model Type** | **Parameters (P)** | **Minimum Data Required (D)** |
| --- | --- | --- |
| Linear Regression | ~10-100 | ~1,000+ |
| Decision Trees | ~100-1,000 | ~10,000+ |
| Small CNN (Image) | ~1M | ~10M |
| Medium NLP Model | ~100M | ~1B |
| LLM (GPT-3) | ~175B | ~300B tokens |

### **4. Key Factors Affecting Generalization**

* **Data Diversity & Quality:** High-quality, diverse data is more valuable than just increasing quantity.
* **Regularization & Techniques:** Dropout, weight decay, and data augmentation help improve generalization.
* **Computational Constraints:** More data and parameters require significant computational resources.

Would you like an estimate for a specific model you're working on?

### **How Many Training Parameters and Data Points are Needed for Generalization?**

The number of **training parameters** (weights in a model) and **training data points** should be balanced to ensure good generalization. There isn't a strict **min-max limit**, but general guidelines exist.

### **1️⃣ General Rule of Thumb**

* **More parameters than data → Overfitting**
  + Example: A deep neural network with 1M parameters trained on 10K data points will memorize the data rather than generalize.
* **Fewer parameters than needed → Underfitting**
  + Example: A simple linear regression with just one feature may not capture complex relationships in a dataset with many variables.

### **2️⃣ Ideal Parameter-to-Data Ratio**

* **For traditional ML models** (Linear Regression, Decision Trees, etc.):
  + **Rule of thumb** → At least **5 to 10 times more data points** than parameters.
  + Example: If a model has 100 parameters, it should ideally have **500–1000 data points**.
* **For deep learning models** (Neural Networks, Transformers):
  + The number of parameters is usually **much higher** than data points.
  + **Rule of thumb** → A model should have at least **10x to 100x more training samples** than parameters.
  + Example: A neural network with **1M parameters** should have at least **10M data points** for good generalization.

### **3️⃣ Minimum & Maximum Limits**

* **Minimum Data Needed:**
  + Depends on model complexity.
  + Example: A simple **linear regression** can work with a few dozen examples, while a **deep learning model** may need thousands or millions.
* **Maximum Data:**
  + More data generally improves generalization **if it is diverse and clean**.
  + If data is noisy or redundant, adding more won't help.

### **4️⃣ How to Handle Limited Data?**

If you have fewer data points than needed:  
 ✔ **Data Augmentation** → Synthesize more data (for images, NLP, etc.).  
 ✔ **Transfer Learning** → Use pre-trained models.  
 ✔ **Regularization** → Apply dropout, L1/L2 penalties to prevent overfitting.  
 ✔ **Cross-Validation** → Use k-fold validation to maximize training data usage.

Would you like a dataset-specific recommendation? 🚀

### **Feature Engineering: A Step-by-Step Guide 🚀**

Feature engineering is the process of transforming raw data into meaningful inputs for a machine learning model. It involves creating, modifying, or selecting features that improve model performance.

## **🔹 Steps in Feature Engineering**

### **1️⃣ Understanding the Data**

Before engineering features, you must deeply understand the dataset.  
 ✔ **Explore the dataset** (types of features, missing values, distribution).  
 ✔ **Identify data types** (numerical, categorical, text, images, etc.).  
 ✔ **Check correlations** between features and the target variable.

🔹 **Example:** If you're working with a house price dataset, you need to know which features (e.g., square footage, location, number of bedrooms) impact the price.

### **2️⃣ Handling Missing Data**

Missing values can harm model performance. Some ways to handle them:  
 ✔ **Remove rows/columns** with too many missing values.  
 ✔ **Impute values** (mean, median, mode, KNN, deep learning methods).  
 ✔ **Use special values** (e.g., -9999 for missing numerical values).

🔹 **Example:** If a dataset has missing "Age" values, replace them with the median age of the dataset.

### **3️⃣ Feature Transformation & Scaling**

Ensuring features are in the right scale and format can improve model performance.

###### ✔ Normalization (Min-Max Scaling) → Rescales values between 0 and 1. ✔ Standardization (Z-score Scaling) → Rescales to have mean = 0 and standard deviation = 1. ✔ Log Transformation → Used for skewed data.

🔹 **Example:** House prices often follow a log-normal distribution, so applying log(price) makes the distribution more normal.

### **4️⃣ Handling Categorical Features**

Machine learning models need numerical inputs, so categorical data must be encoded.

###### ✔ One-Hot Encoding (OHE) → Creates binary columns for each category (good for small categories). ✔ Label Encoding → Assigns unique numbers to categories (used when order matters). ✔ Target Encoding → Replaces categories with the mean of the target variable. ✔ Embedding Representations → Used in deep learning for high-cardinality categorical data.

🔹 **Example:** If you have a "City" column with values ["New York", "London", "Paris"], apply **one-hot encoding** to convert it into three binary columns.

### **5️⃣ Feature Extraction (Creating New Features)**

###### ✔ Datetime Features → Extract "year," "month," "day," "hour" from timestamps. ✔ Text Features → Convert text into numerical vectors using TF-IDF, word embeddings. ✔ Image Features → Use edge detection, color histograms, or pre-trained CNN models. ✔ Interaction Features → Create new features by combining existing ones (e.g., "price per square foot" = price / area). ✔ Domain-Specific Features → Create meaningful features using expert knowledge.

🔹 **Example:** In an e-commerce dataset, create a feature like purchase\_frequency = total\_purchases / customer\_lifetime.

### **6️⃣ Feature Selection (Removing Irrelevant Features)**

Too many features can slow training and cause overfitting. Choose the most important ones.

✔ **Univariate Selection (Statistical Tests)** → Use ANOVA, chi-square, or correlation tests.  
 ✔ **Feature Importance (ML Methods)** → Use tree-based models (Random Forest, XGBoost).  
 ✔ **L1 Regularization (Lasso Regression)** → Eliminates less important features.  
 ✔ **PCA (Principal Component Analysis)** → Reduces dimensionality while retaining information.

🔹 **Example:** In a loan dataset, remove irrelevant columns like "customer ID" or "random survey comments."

### **7️⃣ Feature Engineering for Specific ML Models**

Different models require different feature engineering techniques:  
 ✔ **Linear Models** → Prefer features that follow a normal distribution.  
 ✔ **Tree-Based Models** → Can handle missing values and categorical variables well.  
 ✔ **Deep Learning** → Requires numerical encoding and scaled inputs.

### **8️⃣ Automating Feature Engineering**

✔ **Featuretools** (Python library) → Automatically generates new features.  
 ✔ **AutoML tools** (H2O.ai, DataRobot) → Automate feature selection and transformation.

### **Final Thoughts**

Feature engineering is **one of the most important skills** in machine learning. The goal is to create meaningful, predictive features while reducing noise.

Would you like a **real-world coding example** to see feature engineering in action? 🚀

### **What is an Epoch in Machine Learning?**

An **epoch** in machine learning refers to **one complete pass of the entire training dataset** through the model. During an epoch, the model updates its weights based on the error calculated using the loss function.

* If you have **10,000 samples** and a **batch size of 100**, then one epoch consists of **100 iterations** (10,000 / 100 = 100 batches).
* More epochs allow the model to learn better but may lead to overfitting if too many are used.

### **How Many Epochs Should You Run?**

The optimal number of epochs **depends on the dataset, model complexity, and the problem being solved**. Here’s a general guideline:

| **Model Type** | **Recommended Epochs** | **Why?** |
| --- | --- | --- |
| Simple models (e.g., Linear Regression, SVM) | 10-100 | Converges quickly, no deep layers. |
| Shallow Neural Networks (1-2 hidden layers) | 50-200 | Needs more training but less than deep models. |
| Deep Neural Networks (CNNs, RNNs, Transformers) | 50-1000+ | Large models need more epochs to generalize. |
| Large datasets (ImageNet, NLP tasks) | 10-50 | Even a few epochs are enough due to dataset size. |

### **How to Determine the Right Number of Epochs?**

Instead of setting a fixed number, use **early stopping** to find the optimal number dynamically.

✔ **Monitor Validation Loss** → Stop training when it starts increasing (indicating overfitting).  
 ✔ **Use Early Stopping** → Automatically stops training when improvements are minimal.  
 ✔ **Use Cross-Validation** → Ensures the model generalizes well.

🔹 **Example (Using Keras & TensorFlow):**

from tensorflow.keras.callbacks import EarlyStopping

early\_stopping = EarlyStopping(monitor='val\_loss', patience=5, restore\_best\_weights=True)

model.fit(X\_train, y\_train, epochs=100, validation\_data=(X\_val, y\_val), callbacks=[early\_stopping])

👉 This stops training if val\_loss doesn't improve for **5 consecutive epochs**.

Would you like a more detailed example with a real dataset? 🚀

### **Choosing the Right Number of Epochs & Batch Size**

The number of **epochs** and **batch size** are crucial hyperparameters that affect model performance, training time, and generalization.

## **🔹 Key Factors to Consider for Choosing Epochs & Batch Size**

### **1️⃣ Factors Affecting Number of Epochs**

🔹 **Dataset Size** → Large datasets require fewer epochs, small datasets need more.  
 🔹 **Model Complexity** → Deeper networks need more epochs to learn.  
 🔹 **Overfitting & Generalization** → Use early stopping to avoid too many epochs.  
 🔹 **Computational Power** → More epochs mean longer training; balance it based on resources.  
 🔹 **Loss Convergence** → If training/validation loss stops improving, more epochs are unnecessary.

📌 **General Rule of Thumb for Epochs:**

| **Model Type** | **Recommended Epochs** |
| --- | --- |
| Small datasets (e.g., <10,000 samples) | 100-500+ |
| Medium datasets (e.g., 10k-100k samples) | 50-200 |
| Large datasets (e.g., >1M samples) | 10-50 |
| Deep learning (CNNs, RNNs, Transformers) | 50-1000 |

✔ **Use Early Stopping** to dynamically determine the best number of epochs.

### **2️⃣ Factors Affecting Batch Size**

🔹 **Memory Limitations (GPU/TPU/CPU)** → Large batch sizes need more RAM.  
 🔹 **Gradient Noise** → Smaller batches introduce noise, which helps escape local minima.  
 🔹 **Training Speed** → Larger batch sizes run faster but may generalize poorly.  
 🔹 **Convergence Stability** → Too large a batch size might lead to poor generalization.

📌 **General Rule of Thumb for Batch Size:**

| **Batch Size** | **Use Case** |
| --- | --- |
| **Small (8-32)** | Best for small datasets, helps avoid overfitting. |
| **Medium (32-128)** | Good balance between speed and generalization. |
| **Large (128-1024+)** | Faster but may overfit, requires large memory. |

✔ **Power of 2 Rule:** Use batch sizes like **32, 64, 128, 256** for optimized GPU performance.

## **🔹 How to Choose the Right Epochs & Batch Size Together?**

1️⃣ **Start with a small batch size (32-64) and moderate epochs (50-100).** 2️⃣ **Monitor validation loss & accuracy** using early stopping.  
 3️⃣ **Increase epochs** if underfitting (training loss > validation loss).  
 4️⃣ **Increase batch size** if training is too slow but memory allows.  
 5️⃣ **Reduce epochs** if overfitting (validation loss increases while training loss decreases).

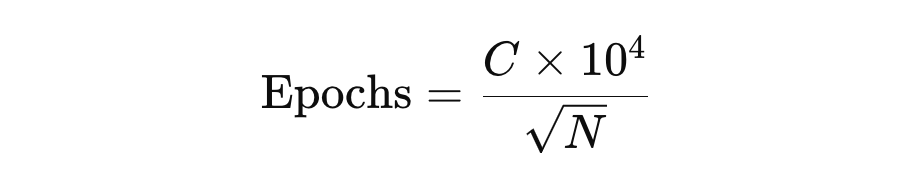
Would you like a practical example with code? 🚀

### **Formula for Approximate Epoch Calculation**

There’s no universal formula for determining the exact number of epochs, but we can create an **approximate guideline** based on dataset size, complexity, and batch size.

### **🔹 Approximate Formula for Epochs**

A good starting point for estimating the number of epochs:



where:

* **N** = Number of training samples
* **C** = Complexity factor (based on model type)
  + **C=3C = 3** for simple models (e.g., linear regression, small neural networks)
  + **C=5C = 5** for medium models (e.g., CNNs, standard RNNs)
  + **C=10C = 10** for deep models (e.g., Transformers, LSTMs, ResNets)

### **🔹 Example Calculations**

#### **Example 1: Small Dataset (10,000 samples, Medium Complexity CNN)**

Epochs=5×10410,000=50,000100=500\text{Epochs} = \frac{5 \times 10^4}{\sqrt{10,000}} = \frac{50,000}{100} = 500

🔹 **Estimated epochs: 500**

#### **Example 2: Large Dataset (1,000,000 samples, Deep Model like Transformer)**

Epochs=10×1041,000,000=100,0001000=100\text{Epochs} = \frac{10 \times 10^4}{\sqrt{1,000,000}} = \frac{100,000}{1000} = 100

🔹 **Estimated epochs: 100**

### **🔹 Adjustments Based on Training Behavior**

✅ **Use Early Stopping** → If validation loss plateaus early, stop training.  
 ✅ **Monitor Underfitting/Overfitting** → Increase epochs if underfitting, reduce if overfitting.  
 ✅ **Adjust Batch Size** → Larger batch sizes can reduce the need for high epochs.

Would you like a Python function to compute this dynamically? 🚀

### **What is Regularization?**

Regularization is a technique used in machine learning to **prevent overfitting** by adding a penalty term to the model’s loss function. It discourages overly complex models and ensures that the learned parameters remain small, leading to better generalization on unseen data.

### **Types of Regularization & How to Perform It**

There are two main types of regularization in linear models:

#### **1. L1 Regularization (Lasso Regression)**

* Adds the **absolute values** of the coefficients to the loss function.
* Can shrink some coefficients to **exactly zero**, performing **feature selection**.
* Used when you suspect some features are irrelevant.

📌 **Mathematical Formulation:**

Loss=MSE+λ∑∣wi∣Loss = MSE + \lambda \sum |w\_i|

where **λ (lambda)** is the regularization strength.

📌 **Implementation in Python (Lasso):**

from sklearn.linear\_model import Lasso

lasso = Lasso(alpha=0.1) # alpha is λ, higher means more regularization

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

#### **2. L2 Regularization (Ridge Regression)**

* Adds the **square** of the coefficients to the loss function.
* Shrinks coefficients close to zero but **never exactly zero**.
* Works well when all features contribute but need to be balanced.

📌 **Mathematical Formulation:**

Loss=MSE+λ∑wi2Loss = MSE + \lambda \sum w\_i^2

📌 **Implementation in Python (Ridge):**

from sklearn.linear\_model import Ridge

ridge = Ridge(alpha=0.1) # Regularization strength

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

#### **3. Elastic Net (Combination of L1 & L2)**

* Mixes both Lasso and Ridge regularization.
* Useful when you need **feature selection** and **some coefficient shrinkage**.

📌 **Implementation in Python (Elastic Net):**

from sklearn.linear\_model import ElasticNet

elastic = ElasticNet(alpha=0.1, l1\_ratio=0.5) # l1\_ratio balances L1 vs L2

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

### **How Does Regularization Affect Overfitting?**

1. **Reduces Variance** – Prevents the model from fitting noise in the training data.
2. **Simplifies the Model** – Keeps only the most important features.
3. **Prevents Large Coefficients** – Shrinking weights avoids overly complex decision boundaries.

🔹 **Without regularization** – The model may have very large weights, leading to high variance and overfitting.  
 🔹 **With regularization** – The weights are constrained, leading to better generalization and lower test error.

Would you like a hands-on example with real data? 🚀

### **How Regularization Works?**

Regularization works by **modifying the loss function** to include a penalty term that discourages large weights. This helps in controlling model complexity and preventing overfitting.

### **1️⃣ How Regularization is Applied in a Model?**

In a standard machine learning model like **linear regression**, the loss function is the **Mean Squared Error (MSE):**

Loss=1n∑(yi−y^i)2Loss = \frac{1}{n} \sum (y\_i - \hat{y}\_i)^2

Where:

* yiy\_i = actual values
* y^i\hat{y}\_i = predicted values
* nn = number of samples

#### **Adding Regularization:**

Regularization modifies this loss function by adding a **penalty term** that **shrinks the model coefficients (ww)**, preventing them from becoming too large.

### **2️⃣ Types of Regularization and How They Work?**

#### **📌 L1 Regularization (Lasso Regression)**

* Adds the **absolute sum** of the weights to the loss function.
* Forces some weights to **exactly zero**, performing **feature selection**.

**Formula:**

Loss=MSE+λ∑∣wi∣Loss = MSE + \lambda \sum |w\_i|

Where **λ (lambda)** is the regularization strength. A **higher λ** results in more regularization and fewer features being used.

🔹 **Effect on Overfitting:**

* **Reduces model complexity** by eliminating unimportant features.
* **Sparse models** (some coefficients become exactly zero).

📌 **Example of L1 in Python:**

from sklearn.linear\_model import Lasso

lasso = Lasso(alpha=0.1) # alpha is λ, controls regularization strength

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

#### **📌 L2 Regularization (Ridge Regression)**

* Adds the **squared sum** of the weights to the loss function.
* Shrinks coefficients **towards zero** but **never exactly zero**.

**Formula:**

Loss=MSE+λ∑wi2Loss = MSE + \lambda \sum w\_i^2

🔹 **Effect on Overfitting:**

* Prevents large coefficient values, leading to a **simpler and more stable model**.
* Retains all features but **reduces their impact**.

📌 **Example of L2 in Python:**

from sklearn.linear\_model import Ridge

ridge = Ridge(alpha=0.1)

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

#### **📌 Elastic Net (L1 + L2 Combined)**

* Mixes both **Lasso (L1)** and **Ridge (L2)**.
* Allows both **feature selection** and **weight shrinking**.

**Formula:**

Loss=MSE+λ1∑∣wi∣+λ2∑wi2Loss = MSE + \lambda\_1 \sum |w\_i| + \lambda\_2 \sum w\_i^2

📌 **Example of Elastic Net in Python:**

from sklearn.linear\_model import ElasticNet

elastic = ElasticNet(alpha=0.1, l1\_ratio=0.5) # l1\_ratio controls L1 vs L2 balance

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

### **3️⃣ Effect of Regularization on Model Performance**

| **Scenario** | **Effect on Model** |
| --- | --- |
| **No Regularization** | Model overfits (high variance, learns noise). |
| **L1 Regularization (Lasso)** | Selects important features, removes others (sparse model). |
| **L2 Regularization (Ridge)** | Shrinks weights but keeps all features. |
| **Elastic Net** | Balances feature selection and weight shrinking. |

### **4️⃣ How to Choose the Right Regularization?**

* Use **L1 (Lasso)** if you suspect many irrelevant features (feature selection).
* Use **L2 (Ridge)** if all features are useful but need to be balanced.
* Use **Elastic Net** if you need a mix of both.

Would you like a **visualization** of how regularization impacts regression coefficients? 🚀

Here’s a **visualization of how regularization affects regression coefficients** using Python and Matplotlib.

### **Concept of Visualization**

We will:

1. Generate a dataset with multiple features.
2. Fit **Linear Regression, Ridge, and Lasso** models.
3. Plot the **coefficients** to show how regularization **shrinks them**.

### **Python Code for Visualization**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.linear\_model import LinearRegression, Ridge, Lasso

from sklearn.datasets import make\_regression

from sklearn.preprocessing import StandardScaler

# Generate synthetic data

np.random.seed(42)

X, y = make\_regression(n\_samples=100, n\_features=10, noise=10, coef=True)

# Standardize features

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

# Train models

lin\_reg = LinearRegression().fit(X\_scaled, y) # No regularization

ridge\_reg = Ridge(alpha=10).fit(X\_scaled, y) # L2 regularization

lasso\_reg = Lasso(alpha=0.1).fit(X\_scaled, y) # L1 regularization

# Get coefficients

coef\_lin = lin\_reg.coef\_

coef\_ridge = ridge\_reg.coef\_

coef\_lasso = lasso\_reg.coef\_

# Plot coefficients

plt.figure(figsize=(10, 6))

plt.plot(coef\_lin, "o-", label="Linear Regression (No Regularization)", linewidth=2)

plt.plot(coef\_ridge, "s-", label="Ridge (L2 Regularization)", linewidth=2)

plt.plot(coef\_lasso, "x-", label="Lasso (L1 Regularization)", linewidth=2)

plt.axhline(0, color="black", linestyle="--", linewidth=1)

plt.xlabel("Feature Index")

plt.ylabel("Coefficient Value")

plt.title("Effect of Regularization on Regression Coefficients")

plt.legend()

plt.show()

### **Expected Results**

* **Linear Regression (No Regularization)** – Coefficients will be large and vary widely.
* **Ridge Regression (L2 Regularization)** – Coefficients will shrink but not become zero.
* **Lasso Regression (L1 Regularization)** – Some coefficients will be **exactly zero** (feature selection).

📌 **Key Takeaways from the Plot:**

* Lasso reduces some coefficients to zero → useful for feature selection.
* Ridge shrinks all coefficients but retains all features.
* Linear regression has large coefficients → prone to overfitting.

Would you like additional experiments, like adjusting **alpha values** for stronger regularization? 🚀