Linear Regression

Welcome

Welcome to the world of Linear Regression! This book is designed to guide you through the fundamentals of this powerful statistical method using Python in Google Colab. We'll take a hands-on approach, ensuring you understand not only the "how" but also the "why" behind each concept.

1 Packing for the Trip: An Intuitive Introduction

Imagine you're planning an exciting trip! In your excitement, you start packing everything you think you might need. Soon, your bag is overflowing with clothes, gadgets, and all sorts of items.

Now, a friend walks in and sees your overloaded bag. What's their first reaction? They might say, "Wow, that's way too much!" or "Are you planning to move there?".

Another friend comes along and gives you different advice. Instead of simply telling you to remove things, they suggest replacing bulky items with lighter, more efficient alternatives. "Instead of 10 heavy shirts, take 5 lighter ones," they might suggest.

This simple scenario illustrates the core concept we'll be exploring: **finding the right "fit" for a model.** In machine learning, just like in packing, we want to include the right "features" (items in your bag) without overloading the model (your bag) with unnecessary or redundant information.

This chapter introduces you to overfitting and underfitting and teaches you how can you improve your model using the concept of regularization.

2 Regression Basics and the Problem of Overfitting

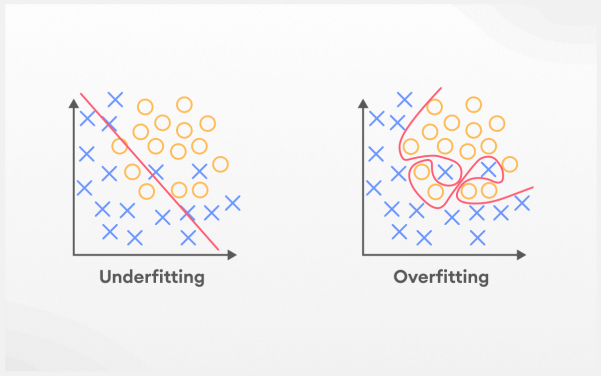

Figure 1: overfitting and underfitting

In previous modules, we learned about supervised learning, where we have labelled data and a "teacher" guiding the learning process. We explored regression for continuous data, using linear regression to find the relationship between variables.

Remember the equation of a line: $y = mx + c$. Here, x is the independent variable, y is the dependent variable, m is the slope, and c is the y-intercept. This equation describes the relationship between x and y.

We learned that linear regression doesn't have a separate code; we simply adjust the "degree" of our x variable. By increasing the degree, we can create polynomial regression, allowing us to fit more complex relationships.

However, a question arises: how far should we increase the degree? Is it necessary to achieve a perfect fit with an R^2 value of 1? This brings us to the crucial concepts of **overfitting** and **underfitting.**

Overfitting occurs when our model fits the training data *too* well. It learns the noise and specific details of the training data, making it perform poorly on new, unseen data. Think of it like memorizing the answers to a specific test instead of understanding the underlying concepts.

Underfitting happens when our model is *too* simple and fails to capture the underlying patterns in the data. It performs poorly on both the training and new data. Think of it like trying to fit a straight line to data that clearly follows a curve.

3 Introducing Multiple Variables

Up until now, we've focused on relationships between a single x and a single y. What if y depends on multiple factors?

Let's say we want to predict a student's marks (y) based on both the hours they studied (x_1) and the hours they slept (x_2). Our equation now becomes:

$$y = w_1x_1 + w_2x_2 + b$$

Where:

- w_1 is the weight or coefficient for x_1 (study hours).
- w_2 is the weight or coefficient for x_2 (sleep hours).
- b is the y-intercept.

This equation expresses how both study hours and sleep hours contribute to the final marks. Our goal is to find the best values for w_1, w_2, and b that accurately predict the marks.

4 Evaluating Model Performance: Mean Squared Error

To evaluate how well our model is performing, we need a way to measure the difference between the predicted values and the actual values. One common metric is the **Mean Squared Error (MSE)**.

Here's how it works:

1. **Calculate the difference (error)** between each actual y value and its corresponding predicted y value.
2. **Square each of these errors.** Squaring ensures that all errors are positive and emphasizes larger errors.
3. **Sum up all the squared errors.**
4. **Divide the sum by the number of data points** to get the average squared error.

The lower the MSE, the better our model is performing. It indicates that our predictions are closer to the actual values.

5 Regularization: Lajjo and Rachel to the Rescue!

Now, how do we deal with the issue of overfitting? That's where regularization comes in. Remember our friends from the packing trip, Lajjo and Rachel? They represent two different regularization techniques.

- **Lajjo (Lasso Regression):** Lajjo is all about simplicity. She throws away unnecessary items from your bag. In machine learning terms, Lasso Regression aims to set the weights of less important features to zero, effectively removing them from the model.
- **Rachel (Ridge Regression):** Rachel is more about optimization. Instead of removing items, she finds lighter, more efficient alternatives. Ridge Regression reduces the magnitude of the weights, making the model less sensitive to individual features and preventing overfitting.

Both Lasso and Ridge Regression add a penalty term to our MSE cost function. This penalty discourages the model from assigning large weights to features, preventing overfitting.

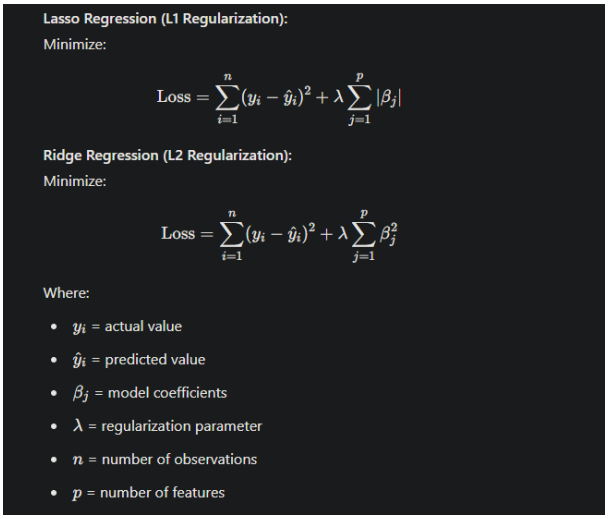

Figure 2: Lajjo and Rache

6 Lasso and Ridge Regression in Action

Let's dive into the equations that define Lasso and Ridge Regression:

Lasso Loss Function:

$$\text{Loss} = \sum(\text{Squared Errors}) + \alpha\left(|w_1| + |w_2| + \cdots\right)$$

Ridge Loss Function:

$$\text{Loss} = \sum(\text{Squared Errors}) + \alpha\left(w_1^2 + w_2^2 + \cdots\right)$$

Where:

- α is a hyperparameter that controls the strength of the penalty. A higher α value increases the penalty, leading to more regularization.
- w_1, w_2, ... are the weights assigned to the features of our data.

The value α determines by how much we want to penalize our weights.

How does it work in practice? The machine learning algorithm will test with a combinatorial explosion of the different values. It will take, for example, $w_1 = w_2 = 0.5$, and compute the Loss with that. If the loss turns out to be great, the weights will be accepted. This continues until the machine learning algorithm finds weights such that the Loss is minimized.

- Lasso Regression adds a penalty proportional to the *absolute value* of the weights. This encourages the model to set some weights to exactly zero, effectively performing feature selection.
- Ridge Regression adds a penalty proportional to the *square* of the weights. This shrinks the weights towards zero, but it rarely sets them exactly to zero.

These penalty terms prevent the model from becoming overly complex and reduce its sensitivity to noise in the training data.

7 Implementing Lasso and Ridge Regression in Python

Now, let's see how to implement Lasso and Ridge Regression using Python and scikit-learn. We will be working on the same Colab notebook that was used previously for this course.

```
from sklearn.linear_model import LinearRegression, Lasso, Ridge
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import r2_score
```

```
# Our Data (Study Hours, Sleep Hours, Marks)
X = [[1, 5], [2, 5], [3, 5], [4, 6]]
y = [2, 4, 7, 8]

# 1. Linear Regression
lr = LinearRegression()
lr.fit(X, y)
y_predicted_linear = lr.predict(X)

# 2. Ridge Regression
ridge = Ridge(alpha=1.0)  # alpha determines the strength of regularization
ridge.fit(X, y)

# 3. Lasso Regression
lasso = Lasso(alpha=1.0)
lasso.fit(X, y)
```

In this code:

- We import the necessary libraries from scikit-learn.
- We define our data X and y.
- We create instances of `LinearRegression`, `Ridge`, and `Lasso` models.
- We fit each model to our data using the `fit()` method.

Now, you can access the coefficients (weights) learned by each model using the `coef_` attribute:

```
print("Linear Regression Coefficients:", lr.coef_)
print("Ridge Regression Coefficients:", ridge.coef_)
print("Lasso Regression Coefficients:", lasso.coef_)
```

You'll notice that the Lasso Regression might set one of the coefficients to zero, effectively removing that feature from the model. The Ridge Regression will shrink the coefficients, making them smaller than those of the Linear Regression.

```
# Example: Adding Polynomial Features (Degree=2)
poly = PolynomialFeatures(degree=2)
X_poly = poly.fit_transform(X)
poly_lr = LinearRegression()
poly_lr.fit(X_poly, y)
y_predicted_poly = poly_lr.predict(X_poly)

# Evaluate Model with R-squared values
r2_linear = r2_score(y, y_predicted_linear)
r2_poly = r2_score(y, y_predicted_poly)
```

When fitting the polynomial regression with `degree = 2`, you will see that the R^2 score turns out to be 1. This might indicate that our model is overfitting, and therefore we might want to use regularization methods like Ridge Regression and Lasso Regression.

8 Lasso vs Ridge: Making the Right Choice

- **Feature Selection:** If you suspect that some features are irrelevant, Lasso Regression is a good choice because it can perform feature selection by setting their weights to zero.
- **Preventing Overfitting:** If all features are potentially useful, Ridge Regression can prevent overfitting by shrinking the weights and making the model less sensitive to individual features.