# Machine Learning Self-Study #1

Price Prediction with Linear Regression

# 1. Motivation And Purpose

I am starting this ‘*Machine Learning Self-Study*’ series because I want to understand how today’s Artificial Intelligence systems work, not just at the surface level, but at their core. Rather than simply using AI tools, I want to explore **the underlying mechanisms and algorithms** that power them. Although my ultimate goal is to pursue a more bio-inspired approach to AI, I understand the importance of first mastering the common practices and foundational concepts in AI and ML.

I’m still new to advanced math, and diving into it has been challenging. However, I’m not one to shy away from difficult concepts. This document is my way of organizing my thoughts and building a clear, structured understanding of these foundational ideas, both in theory and in practice. My goal is to demonstrate my commitment to mastering complex concepts and to show my willingness to confront challenges head-on.

My first attempt was to create a neural network capable of recognizing numbers from 0 to 9 using the MNIST training set, all from scratch without relying on extensive frameworks. However, I quickly realized that I needed a stronger foundation in certain underlying concepts. Which then led me to shift my focus towards more fundamental projects like this one... to build a solid understanding before tackling more complex topics.

Everything here is written based on my own research, I did not use any pre-written material or tutorials for this.

# 2. Concepts

Linear Regression is a fundamental tool in **predictive modeling**. But what exactly is predictive modeling? It’s the process of making informed decisions based on past data by finding patterns and using them to anticipate future outcomes. Interestingly, strong evidence suggests that the human brain works in a similar way, by using past experiences to predict the future and comparing those predictions to reality. When the brain’s predictions are wrong, it adjusts its internal model to minimize future errors. This is closely related to how predictive modeling works in machine learning. By constantly making predictions, checking for errors, and updating the model, machines can learn from data in a way that mirrors how the human brain learns from experience.

Linear Regression is one of the simplest and most commonly used tools for predictive modeling. It identifies **linear patterns in data by drawing the best-fitting line through data points**, allowing it to make predictions or estimations about future outcomes. Because Linear Regression makes strictly linear predictions, it works best when there is a clear linear relationship between input and output variables. For more complex patterns or non-linear relationships, other techniques can be used, although Linear Regression remains a powerful tool for understanding simple connections in data.

This is why I have decided to build a **price estimator for pre-built PCs** as my project. By analyzing the relationship between hardware specifications and market prices, the model can provide reasonable price estimates for different configurations, making it a practical application of Linear Regression.

I won’t be tackling **Gradient Descent** just yet, as I want to keep this project fairly simple. Once I understand the basics, I’ll move on to more advanced concepts.

# 3. Objective

This project serves as an introduction to Linear Regression, showing how it can be used to find patterns in data and make numerical estimations. We will do this by building a model that can **estimate the price of pre-built PCs based on their hardware specifications**. By analyzing the relationship between key features such as *CPU model*, *GPU model*, and *RAM size*, the model aims to provide reasonable price estimates for different configurations. This can help consumers make informed purchasing decisions and give sellers a benchmark for competitive pricing.

## Input Features

This model could include many input features, such as CPU type, GPU type, RAM, VRAM, or even factors like market fluctuations and brand reputation. However, to keep the project simple and focused, I have chosen to use only three key features: **CPU model**, **GPU model**, and **RAM size**. These features provide the most significant influence on the price of a pre-built PC while keeping the model manageable and easy to understand.

## Output

The model will produce a single output: **the estimated price of the pre-built PC**. This estimated price reflects the relationship between the chosen input features and the market value of different PC configurations.

# 4. Data Collection

Data is essential for powering AI systems. The more data an AI can learn from, the better its predictions will be. Data comes in various forms and types. For this project, we'll use the **'Laptop Price Dataset'** from **Kaggle**, a platform that offers many freely available datasets for machine learning. While it's possible to use custom data or scrape data from websites, it would be too time-consuming and isn't feasible within the scope of this project.

| | A | B | C | D | E | F | G | H | I | J | K |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 127 | Razer | AMD Ryzen 7 | 64 | 1TB HDD | AMD Radeon RX 6800 | 14 | 3840x2160 | 10.1 | 2.8 | Linux | 5766.77 |
| 128 | Asus | AMD Ryzen 3 | 32 | 1TB SSD | Nvidia RTX 3080 | 16 | 3840x2160 | 4.4 | 1.75 | Windows | 2490.41 |
| 129 | MSI | Intel i9 | 64 | 512GB SSD | Integrated | 17.3 | 2560x1440 | 4.7 | 1.38 | FreeDOS | 4705.97 |
| 130 | Lenovo | Intel i7 | 64 | 512GB SSD | AMD Radeon RX 6800 | 16 | 2560x1440 | 8 | 2.14 | macOS | 4036.38 |
| 131 | MSI | AMD Ryzen 3 | 4 | 1TB HDD | Nvidia GTX 1650 | 16 | 2560x1440 | 6.5 | 1.89 | Windows | 1261.77 |
| 132 | Samsung | AMD Ryzen 5 | 8 | 1TB HDD | Integrated | 17.3 | 1366x768 | 11 | 2.66 | Linux | 620.44 |
| 133 | HP | Intel i5 | 32 | 2TB SSD | Nvidia GTX 1650 | 13.3 | 3840x2160 | 7.4 | 2.7 | Linux | 2500.94 |
| 134 | Lenovo | Intel i9 | 32 | 2TB SSD | Nvidia RTX 2060 | 15.6 | 1366x768 | 10.9 | 3.17 | Windows | 2431.14 |
| 135 | Asus | Intel i5 | 4 | 1TB SSD | Nvidia RTX 3080 | 14 | 2560x1440 | 11.5 | 3.43 | macOS | 1217.24 |
| 136 | MSI | Intel i5 | 4 | 512GB SSD | AMD Radeon RX 6800 | 16 | 1366x768 | 5.8 | 2.4 | Linux | 954.65 |
| 137 | HP | AMD Ryzen 5 | 8 | 1TB SSD | Nvidia GTX 1650 | 17.3 | 1920x1080 | 5.7 | 3.07 | FreeDOS | 1250.43 |
| 138 | Acer | AMD Ryzen 5 | 64 | 512GB SSD | Nvidia RTX 3060 | 17.3 | 2560x1440 | 6.3 | 1.37 | Windows | 3401.59 |
| 139 | HP | AMD Ryzen 7 | 64 | 1TB SSD | Integrated | 13.3 | 1920x1080 | 4.3 | 1.31 | FreeDOS | 2041.37 |
| 140 | Lenovo | Intel i9 | 64 | 1TB SSD | Nvidia RTX 3080 | 13.3 | 2560x1440 | 10.3 | 2.99 | macOS | 4995.89 |
| 141 | HP | AMD Ryzen 7 | 4 | 1TB HDD | Nvidia GTX 1650 | 15.6 | 1366x768 | 7.1 | 2.63 | macOS | 777.38 |
| 142 | Acer | Intel i5 | 8 | 2TB SSD | AMD Radeon RX 6800 | 16 | 3840x2160 | 11.7 | 3.27 | Windows | 1969.19 |
| 143 | Apple | Intel i5 | 8 | 256GB SSD | Nvidia RTX 2060 | 14 | 3840x2160 | 5.2 | 2.1 | Linux | 2607.44 |
| 144 | Microsoft | Intel i9 | 16 | 1TB HDD | Nvidia RTX 3060 | 15.6 | 1920x1080 | 11.3 | 1.81 | macOS | 1776.13 |
| 145 | Microsoft | AMD Ryzen 9 | 32 | 1TB SSD | Nvidia RTX 3080 | 13.3 | 1920x1080 | 11.8 | 1.28 | Linux | 2721.4 |
| 146 | HP | AMD Ryzen 5 | 16 | 512GB SSD | AMD Radeon RX 6600 | 16 | 1366x768 | 8.9 | 1.75 | Windows | 1016.16 |
| 147 | Acer | AMD Ryzen 3 | 4 | 1TB SSD | Integrated | 15.6 | 1920x1080 | 7.8 | 1.66 | FreeDOS | 590.45 |
| 148 | Razer | Intel i9 | 8 | 1TB SSD | Nvidia RTX 2060 | 15.6 | 1920x1080 | 8.7 | 2.78 | Linux | 2549.6 |
| 149 | Asus | AMD Ryzen 3 | 32 | 512GB SSD | Nvidia RTX 2060 | 17.3 | 1920x1080 | 11.9 | 1.63 | FreeDOS | 1609.14 |
| 150 | Acer | AMD Ryzen 7 | 64 | 2TB SSD | AMD Radeon RX 6800 | 16 | 3840x2160 | 6.5 | 2.09 | Linux | 5483.75 |
| 151 | Dell | AMD Ryzen 3 | 8 | 1TB SSD | Nvidia RTX 3060 | 15.6 | 1366x768 | 5.5 | 1.28 | Linux | 728.2 |

*Figure 1: Our dataset containing the columns: Brand, Processor, RAM (GB), Storage, GPU, Screen Size (inch), Resolution, Battery Life (hours), Weight (kg), Operating System, Price ($)*

The Laptop Price Dataset comes in the Comma-Separated Values (csv) format, a simple, structured and easy to use fileformat to use in Python for smaller projects.

# 5. Data Preparation

In order for our model to effectively work with the data, we need to format it in a way that it can process. The first step in this process is to **clean the data.** While our dataset doesn't contain any obvious mistakes, like missing values or duplicate entries, there are still some things we have to change in order for our data to be ready to be read by the model.

First, we'll select the relevant columns: **Processor**, **GPU**, **RAM (GB)**, and **Price ($)**. The **Processor** and **GPU** models are categorical, meaning they are **strings** (e.g., "Intel i7", "RTX 3060"). Since Linear Regression only works with numerical data, we need to **encode these categorical variables** into numbers (label encoding: each unique category gets a corresponding numerical value).

| | A | B | C | D |
|---|---|---|---|---|
| 1 | Processor | RAM (GB) | GPU | Price ($) |
| 2 | AMD Ryzen 3 | 64 | Nvidia GTX 1650 | 3997.07 |
| 3 | AMD Ryzen 7 | 4 | Nvidia RTX 3080 | 1355.78 |
| 4 | Intel i5 | 32 | Nvidia RTX 3060 | 2673.07 |
| 5 | Intel i5 | 4 | Nvidia RTX 3080 | 751.17 |
| 6 | Intel i3 | 4 | AMD Radeon RX 6600 | 2059.83 |
| 7 | AMD Ryzen 3 | 64 | Nvidia RTX 3060 | 1676.64 |
| 8 | Intel i5 | 8 | Nvidia RTX 2060 | 1449.05 |
| 9 | AMD Ryzen 9 | 8 | Nvidia RTX 3080 | 2193.55 |

*Figure 2: Our simplified dataset*

For the **RAM** and **Price**, we will **standardize/normalize** the values. This means scaling them to a range between **0 and 1**, so the model treats each feature equally regardless of scale. For example, 128 GB of RAM is a large number compared to the price in dollars ($128 for a PC would be a very good deal), which could lead the model to prioritize one feature over the other. Normalizing the values helps prevent this issue by putting them on the same scale and ensure the weighting is fair for all inputs.

Manually handling these tasks can be time-consuming, but luckily there are libraries that automate the process. In Python, we can use the **pandas** library to read the CSV file and the **sklearn** library to prepare the data. To make it easier for others to set up the project,

I've included a **requirements.txt** file, which can be installed with the following command:

*pip install -r requirements.txt*

I have created a **DataPreparation** class to keep the main file tidy. Here you can see how the **pandas** and **sklearn** libraries can be used:

- The **Processor** and **GPU** categories will be **encoded numerically** to make them usable in the model.
- **RAM** and **Price** will be **normalized** to a scale between 0 and 1. For **RAM**, 0 will correspond to 4GB and 1 will correspond to 64GB. For **Price**, 0 will represent the lowest price in the dataset, and 1 will represent the highest price.
- These libraries also allow for **SQL-like querying**, which makes it easy to filter, group, and manipulate data
- I use **DataClasses** to keep the code clean (no boilerplate methods such as __init__ and __repr__)

*Note: I have removed some exception handling to keep the example tidy.*

```
from dataclasses import dataclass, field
from sklearn.preprocessing import LabelEncoder, MinMaxScaler
import pandas as pd

@dataclass
class DataPreparation:
    csv_file: str  # Path to the CSV file
    df: pd.DataFrame = field(init=False)  # DataFrame to store the CSV data
    label_encoder: LabelEncoder = field(init=False, default_factory=LabelEncoder)
    minmax_scaler: MinMaxScaler = field(init=False, default_factory=MinMaxScaler)

    def __post_init__(self):
        """ Load the CSV file into a DataFrame and initialize encoders/scalers """
        csv = pd.read_csv(self.csv_file)
        self.df = csv[['Processor', 'GPU', 'RAM (GB)', 'Price ($)']]

    def prepare_data(self):
        """ Prepare the data by encoding categories and normalizing numerical values """
        self.df['Processor'] = self.label_encoder.fit_transform(self.df['Processor'])
        self.df['GPU'] = self.label_encoder.fit_transform(self.df['GPU'])
        self.df['RAM (GB)'] = self.minmax_scaler.fit_transform(self.df[['RAM (GB)']])
        self.df['Price ($)'] = self.minmax_scaler.fit_transform(self.df[['Price ($)']])

    def get_data(self):
        """ Return the prepared DataFrame """
        return self.df
```

## Splitting The Training Data

We will split the training data into two parts. 80% of the samples will be used for training the model. The remaining 20% will be used to evaluate the model's performance. This helps us test how well the model performs on new, unseen data. It also helps prevent overfitting and helps with generalization. We can also use sklearn for this: **sklearn.model_selection – train_test_split**

This is the **main** python file so far:

```
import os
from data_preparation import DataPreparation
from sklearn.model_selection import train_test_split

if __name__ == '__main__':
    #append the directory of the current file to the csv file
    csv_path = os.path.join(os.path.dirname(__file__), 'laptop_prices.csv')

    #Use the DataPreparation class to prepare the data
    dp = DataPreparation(csv_file = csv_path)
    dp.prepare_data()
    priceDataFull = dp.get_data()

    #Split the inputs and output data into two arrays
    X = priceDataFull[['Processor', 'GPU', 'RAM (GB)']] #Capitalized because it's a matrix
    y = priceDataFull['Price ($)']                      #Not capitalized because it's a vector

    #Use sklearn train_test_split to split the data into training and testing sets
    #test_size is the proportion of the dataset to include in the test split (train_size is inferred)
    #random_state is the seed used by the random number generator
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 1)

    print("\n\nTraining Input Features Head:\n\n",X_train.head())
    print("\n\nTraining Output Head:\n\n",y_train.head())
```

# 6. Model Design

As mentioned earlier, our model will be based on **Linear Regression**, a type of **supervised learning** algorithm. In supervised learning, the model is trained on **labeled data**, meaning each input in the training set is paired with the correct output. The model learns from these examples and adjusts itself to predict the correct outcome for new, unseen data.

For this project, the **input features** we will use are the **processor**, **GPU**, and **RAM**, while the **output** is the predicted **price** of the laptop.

In computer applications, **business logic** defines the behavior of the program, acting as the "meat" of the application. Similarly, in machine learning, the driving force behind the model is the **model equation**. This equation captures the relationship between the input features and the output, and in the case of **Linear Regression**, it takes the form of a simple linear equation.

The linear equation can be written as:

$$y = w_0 + w_1 \cdot x_1 + w_2 \cdot x_2 + w_3 \cdot x_3$$

- **y** = The output (**predicted price**)
- **$w_0$** = The **intercept** (the point at which the line intersects the y-axis). This acts as a default value when all input features are zero, which might not represent a "natural" zero in real-world terms. In neural networks, this is referred to as the **bias**.
- **$w_1$, $w_2$, $w_3$, ...** = **Weights**, which determine how much each feature influences the predicted price. The main goal is to learn these values from the **training data**.
- **$x_1$, $x_2$, $x_3$, ...** = The **input features** (e.g., CPU, GPU, RAM).

In essence, this equation represents a simple linear relationship between the input features and the output. This is similar to a **single neuron** in a **neural network**, where inputs are weighted and combined to produce an output, but without the activation function that introduces non-linearity.

So overall, our model will have:

- 3 inputs
- 3 weights (1 weight for each input)
- a bias (the y-intercept)

# 7. Training Process

The training process for **Linear Regression** is about finding the best-fitting line that can predict our output. At a high level, this process is about tweaking the model's **weights** based on how **"wrong"** the model's predictions are. We will be doing this for every individual sample (we'll call this the **error**), as well as for each epoch (we'll call this **cost**).

The goal is to keep the cost as low as possible by adjusting the weights so that the model's predictions better align with the actual outcomes.

## Weight Initialization

Initially, the weights can be set to initial values, zeroes, or random values. From what I have read, setting all weights to zero can cause symmetry problems, while random values help break this symmetry. To err on the safe side, I'll be picking small random values instead of starting from zero.

## For Each Sample, We …

### i. Make A Prediction

The **prediction**, often depicted as the letter y with a hat symbol on top, can be calculated using the following formula:

$$\hat{y} = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3$$

Where:

- $w_0$ is the **y-intercept** (or **bias**), the point where the line intercepts the y-axis.
- $w_i$ represents the **$i^{th}$ weight**.
- $x_i$ represents the **$i^{th}$ input feature**.

We **multiply each input** by its corresponding weight and **add the bias** to get the prediction.

### ii. Calculate The Error

The **error** is difference between the predicted value and actual value of the current sample. This error variable should not be confused with the **cost** (I frequently confused the two during my research).

Error says something about **the current sample**, cost says something about how **the whole model overall** is doing.

$$error = \hat{y} - y$$

## iii. Compute The Gradient

The gradient can be seen as the **slope** for each weight, showing how much the output (y) changes for each unit of the corresponding input (x).

In order for the model to learn, it needs to figure out **how much each weight contributed to the error** of the current sample. This can be calculated by **multiplying the error by the input value** for that feature.

**Gradient formula:**

$$\frac{\partial}{\partial w_i} = error \times x_i$$

The symbol on the left, the **partial derivative**, means that we're looking at **how much the error changes when we tweak one weight**, while keeping the other weights the same.

What this formula is saying is: for each weight of a feature, **multiply the error by the input** for that feature. Doing so gives us the **slope** for each weight, which tells us:

- **How much** that weight contributed to the overall error.
- **In which direction** we need to adjust the weight to reduce the error.

## iv. Update The Weights

Updating the weights is done by **multiplying** the learning rate α by the **gradient** and **subtracting** the result from the old weight.

**Weight update formula:**

$$w_i = w_i - \alpha \times \frac{\partial}{\partial w_i}$$

We **repeat this for every weight** in the model, including the bias. This is what allows the model to learn and **gradually reduce the error**.

**Note:** The bias (w0) is simply the first weight and follows the same update rule. Its corresponding input is always **1**, so it's just like the other weights, but with a fixed input. This means we don't need

any special rules for the bias, we just simply include it in the weight update loop.

## Evaluating The Model (Per Epoch)

After we've looped through all the samples, we have completed an **epoch**. This is the moment where we evaluate how well the model has predicted the output of the samples. For each sample, we have measured the error and tweaked the weights. But how can we measure how well we've done and 'steer' the training in the right direction? That's where the **cost** comes in. We've handled the error, which was on a per-sample basis, but the cost parameter will be used to influence the **learning rate**. The learning rate is one of the **hyperparameters** that control how the model learns.

### v. Calculating The Cost

How can we figure out what the **cost** is? To determine the cost of our model's predictions, we will use the **Mean Squared Error (MSE)**.

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

- **y-hat$_i$** = the predicted value at index i
- **y$_i$** = the actual value at index I
- **m** = the number of data-samples

This function will loop through all of the samples again, take the error (y-hat – y), but this time square it. Why squaring? Because it will turn negative numbers into positive ones and it will scale larger numbers up while scaling smaller ones down. When you square a number between 0 and 1, it will shrink. When you square a number above 1, it will grow. This will make it so that bigger errors will have a bigger impact on the cost. After squaring each error, we sum them together and divide them by the number of samples to get the average cost for the epoch.

### vi. Setting The Learning Rate

The cost can then be used to set the learning rate, which is used during the updating process of the weights. So indirectly, the cost influences the weights in a way like guiding the learning process. There are a couple of ways to adjust the learning rate, but I will be

using a fairly simple formula that divides the learning rate each time the cost plateaus.

For example, if the cost hasn’t decreased for an x amount of epochs, we **reduce the learning rate**:

$$\alpha = \alpha / 2$$

## vii. Sample Shuffling

In Stochastic Gradient Descent, it’s important to shuffle the training samples at the end of each epoch. Why? This prevents the model from learning the order of the data, which can lead to **overfitting**. Which is when the model learns the answers rather than the underlying concepts and patterns in the data (like how a student who memorizes the answers won’t really learn anything).

In our case, if the cheap PCs are listed first and the expensive ones last, the model can learn the order instead of the actual relationship between the hardware and the price.

## viii. Convergence And Overfitting

When the cost isn’t decreasing by much anymore for several epochs, we can stop training. The model has **‘converged’**. Convergence means the model has trained all it can from the data and is no longer improving.

Why for several epochs and not immediately after 2 epochs? There is a chance that the model mistakes a temporary plateau for convergence. We want to avoid that by checking the cost for multiple epochs.

But how do we decide when to stop? Here are the stopping criteria we will use:

- If the cost stops decreasing by a small amount for several consecutive epochs, we’ll stop training. This means the model has **converged** and is no longer learning anything new.
- If the cost increases for multiple consecutive epochs, we’ll stop training to prevent **overfitting**. This means the model is starting to memorize noise rather than learning the pattern.

This makes sure the model learns as much as possible without overfitting or wasting time.

## Number of Epochs

An epoch is one full pass over all the samples in the training set. We will repeat this process for multiple epochs to allow the model to learn from the data.

But how many epochs should we use? There isn’t a fixed rule for this, as it depends on the data and the learning rate. We will continue training until the model converges or starts overfitting, based on the stopping criteria above.

To avoid infinite loops or over-training, we will also set a **maximum number of epochs**, even if the model hasn’t converged. This prevents the model from overfitting or wasting time on unnecessary training.

# 8. Reflection

(will be filled in)