50.039 Theory and Practice of Deep Learning

W1-S1 Introduction and Machine Learning Reminders

Matthieu De Mari

A quick word about instructors

- Matthieu (**Matt**) De Mari
- Lecturer at SUTD (Python, Deep Learning, AI, and more)
- Information Systems Technology and Design (ISTD) pillar/faculty
- PhD from CentraleSupelec (France)

- Email: matthieu_demari@sutd.edu.sg
- Office @ SUTD: 2.401.07

Objectives of 50.039 DL

- Introduce the technical aspects related to the **implementation of Deep Neural Networks from scratch**.
- Teach the students about the **most popular framework** for implementing Deep Neural Networks (as of Jan 2023): **PyTorch.**
- Discuss the **mathematical foundations and intuitions** behind the Deep Learning framework, layers and some techniques.
- Give the students a **global overview of the advanced techniques** related to Deep Learning and Deep Neural Networks.
- Describe **examples of practical applications** of Deep Learning, showing how some key AIs were implemented using Deep Learning.

Objectives of 50.039 DL

- Eventually, **assemble all concepts and tricks of the trade** discussed in this course to produce big architecture models that are close to the current state-of-the-art ones (e.g. ChatGPT, Dall-E, etc.).
- Discuss **ethical aspects** related to some of the concepts discussed.
- Give students **an understanding of the current state of research in the Deep Learning community**, and more specifically **pointers** as to where the most up-to-date information can be found.
- This includes **tracking big names** in the field, identifying **key scientific papers** that have changed the field, but also **newsletters**, **websites** or **magazines** that disclose information in layman terms.

Skills needed for this course

- **Must-have CS:** Python, Numpy, Matplotlib.
- **Must-have Math:** Linear Algebra, multiple variables calculus, derivatives, optimization probability and statistics.

- **Good-to-have CS:** Machine Learning, Data Processing, Scipy, Sklearn.
- **Good-to-have Math:** Graph Theory, Game theory.

- **Need to revise?** Time to check your 10.014 CTD, 10.013 M&A, 10.018 MS&S, 10.022 MU, 10.020 DDW, and 50.007 ML!

The way I teach things

An expert is not someone who

- Undertook a 6h-long online course,
- Implements Neural Networks using basic layers (Linear, Conv, and that is it),
- Using a random framework he does not understand (or autoML!),
- and is happy to get an 80% accuracy on a simple dataset like MNIST.

An expert is someone who

- Understands how Neural Network operate and the mathematical intuition behind typical (advanced) operations,
- Understands how the frameworks have been built and what they do behind the scene.
- Someone who knows how to stay up to date when it comes to AI.

The way I teach things

- **One topic per week** (or so) and hopefully, we will cover all the important concepts and important directions of DL these days.
- **Lectures slides** uploaded the day before, and **supporting notebooks**.
- **Mathematical aspects** discussed in class (behind the scene of Neural Networks/Deep Learning).
- Made a choice: providing lots of code to demonstrate concepts, but **need you to play with them autonomously** to explore concepts!
- **Extra reading**, for curiosity (supporting papers, articles, etc.) and suggestions of pointers to follow to stay up to date.
- Suggestions for **continuing your learning**, after this course.

Syllabus

- **Week 1:** Introduction to course, some ML jargon reminders, linear and polynomial regression, generalization, ridge regression and regularization, overfitting/underfitting, logistic regression, neurons objects and how they relate to biology, our first shallow neural network, gradient descent reminders, training and testing procedure.

- **Week 2:** Introduction to PyTorch framework, tensors and dataloaders, implementing a shallow neural network in PyTorch, backpropagation in PyTorch with AutoGrad, advanced optimizers, multi-label classification with shallow neural networks, moving from shallow to deep neural networks.

Syllabus

- **Week 3:** Guided project and good practices for Deep Learning projects (train/test/dev, bias/variance, advanced regularization, dropout, normalizing inputs/outputs/layers, trainer functions, savers/loader functions for reproducibility and transfer learning).

- **Week 4:** The image data type, image processing techniques and typical computer vision operations, the convolution operation and layers, Convolutional Neural Networks, advanced CNNs and SotA. Preparing transition to the 50.035 Computer Vision course.

Syllabus

- **Week 5:** Sequential data (times series, text, etc.), vanilla Recurrent Neural Networks, Gated Recurrent Units, Long-Short Term Memory cells, advanced RNN networks, mixing models for advanced architectures and project announcement.

- **Week 6:** Adversarial machine learning, attacking a Neural Network with basic gradient-based attacks, fundamental limits of Neural Networks, defense mechanisms and state-of-the-art of some advanced attacks techniques. MidTerm exam (based on W1-5).

Syllabus

- **Week 8:** The embedding problem, more advanced concepts on RNNs, introduction to Natural Language Processing (NLP) and Word Embeddings for NLP, brief state-of-the-art on NLP, attention and transformers architectures.
  Preparing transition to the 50.040 Natural Language Processing course.

- **Week 9:** Quick introduction to Graph Theory and typical graph datasets and problems, basics of Graph Convolutional Networks, brief state-of-the-art of advanced Graph Convolutional Networks.

Syllabus

- **Week 10:** Generative Models, Autoencoders and Variational Autoencoders, Generative Adversarial Networks (GANs), Advanced concepts on Generative Adversarial Networks, Practice on GANs.

- **Week 11:** Brief introduction to reinforcement learning, and state-action-rewards systems, multi-armed bandit problem and the exploration/exploitation trade-off, Q-learning and Deep Q-Learning. Brief state-of-the-art discussion about further works in Reinforcement Learning.

Syllabus

- **Week 12:** Explainability/Interpretability and open questions in research about Neural Networks. Deep belief models and diffusion models. What will be the next revolution in AI? (a word on ChatGPT, Dall-E, etc.). Closing and future directions for studying Deep Learning.

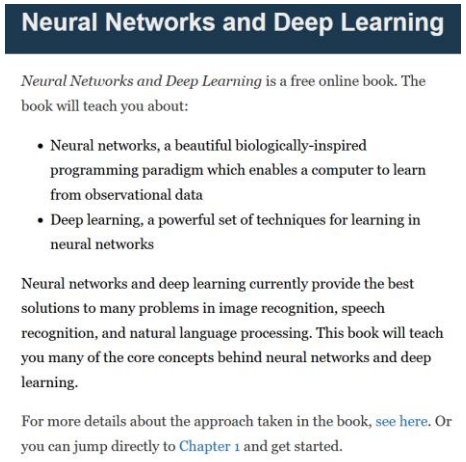
- **Week 13:** Recap. Project presentations and guest conferences (TBA).

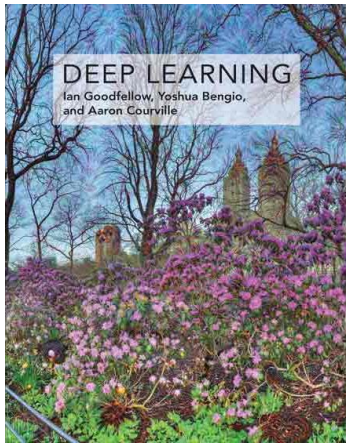
- **Week 14:** Final exam (probably W1-13).

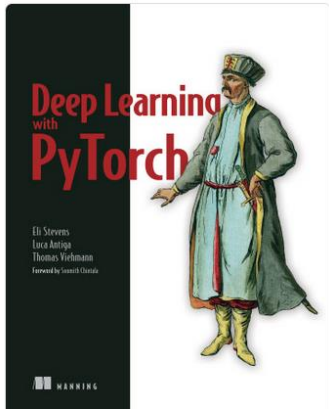
Supporting Textbooks

Supporting textbooks, for your curiosity (not needed to understand this course).

- Michael A. Nielsen, “Neural networks and deep learning”, 2015. (http://neuralnetworksanddeeplearning.com/)
- Ian Goodfellow, Yoshua Bengio and Aaron Courville, “Deep learning”, 2016. (https://www.deeplearningbook.org/)
- Stevens et al., “Deep Learning with PyTorch”, 2020. (https://www.manning.com/books/deep-learning-with-pytorch)







Evaluation and grading

- **Homeworks (24%):** Given on Weeks 2, 5, 8, 10.
  Usually come in the form of a Jupyter Notebook, containing explanations, code snippets and questions.
  Submissions on eDimension, two weeks later or so, as a small PDF report containing code, figures and answers to questions.
  When time allows, debrief of homeworks in class.

- **MidTerm Exam (20%):** Given on Week 6, March 1st 2023.
  Theoretical, paper exam, notions of Week 1-5 to be tested.
  More details about venue and exam details to be announced closer to the exam date.

Evaluation and grading

- **Final Exam (20%):** Given on Week 14, April 26th 2023. Theoretical, paper exam, notions of Week 1-13 to be tested. More details about venue and exam details to be announced.

- **Project (29%):** Groups of 2-3 students. Submission for project (code, report, presentation) expected on Week 13. Problem statement to be freely decided by students, as long as it matches a list of given requirements. We will have a guided demo project on Week 3, more details about the project will be given on Week 5.

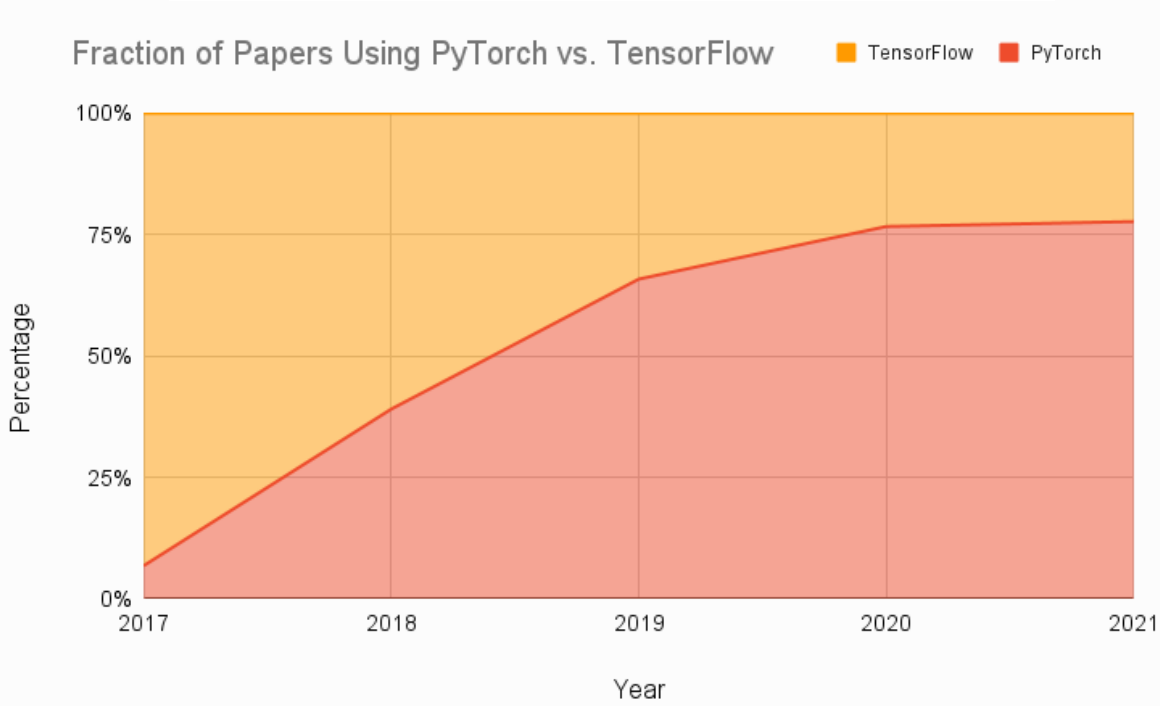
- **Participation (5%):** to my discretion.
- **Student Feedback Survey (2%):** the usual.

Technical pre-requisites

- Install **Python 3**, if you have not done so already (this course does not cover C++/Java).
- **Libraries needed (maybe more based on projects/homeworks):** numpy, matplotlib, scipy, sklearn, networkx, pillow, gym.
- **Jupyter notebooks** for demos of code, along with the slides.
- In doubt, you can always use **Google Colab** to run the codes.

Technical pre-requisites

- Framework of choice will be **PyTorch!** (not Tensorflow, not Keras, not MXNet, etc.)
- Increasing popularity and preferred to Google's Tensorflow these days for many reasons.
- Learn more, if curious: https://www.assemblyai.com/blog/pytorch-vs-tensorflow-in-2022/



Installing PyTorch and CUDA

- Install PyTorch, by getting the right version from https://pytorch.org/get-started/locally/

PyTorch Build	**Stable (1.13.0)**		Preview (Nightly)	
Your OS	Linux	Mac	**Windows**	
Package	Conda	**Pip**	LibTorch	Source
Language	**Python**		C++ / Java	
Compute Platform	CUDA 11.6	**CUDA 11.7**	~~ROCm 5.2~~	CPU
Run this Command:	`pip3 install torch torchvision torchaudio --extra-index-url https://download.pytorch.org/whl/cu117`			

Installing PyTorch and CUDA

Compromised PyTorch-nightly dependency chain between December 25th and December 30th, 2022.

If you installed PyTorch-nightly on Linux via pip between December 25, 2022 and December 30, 2022, please uninstall it and torchtriton immediately, and use the latest nightly binaries (newer than Dec 30th 2022).

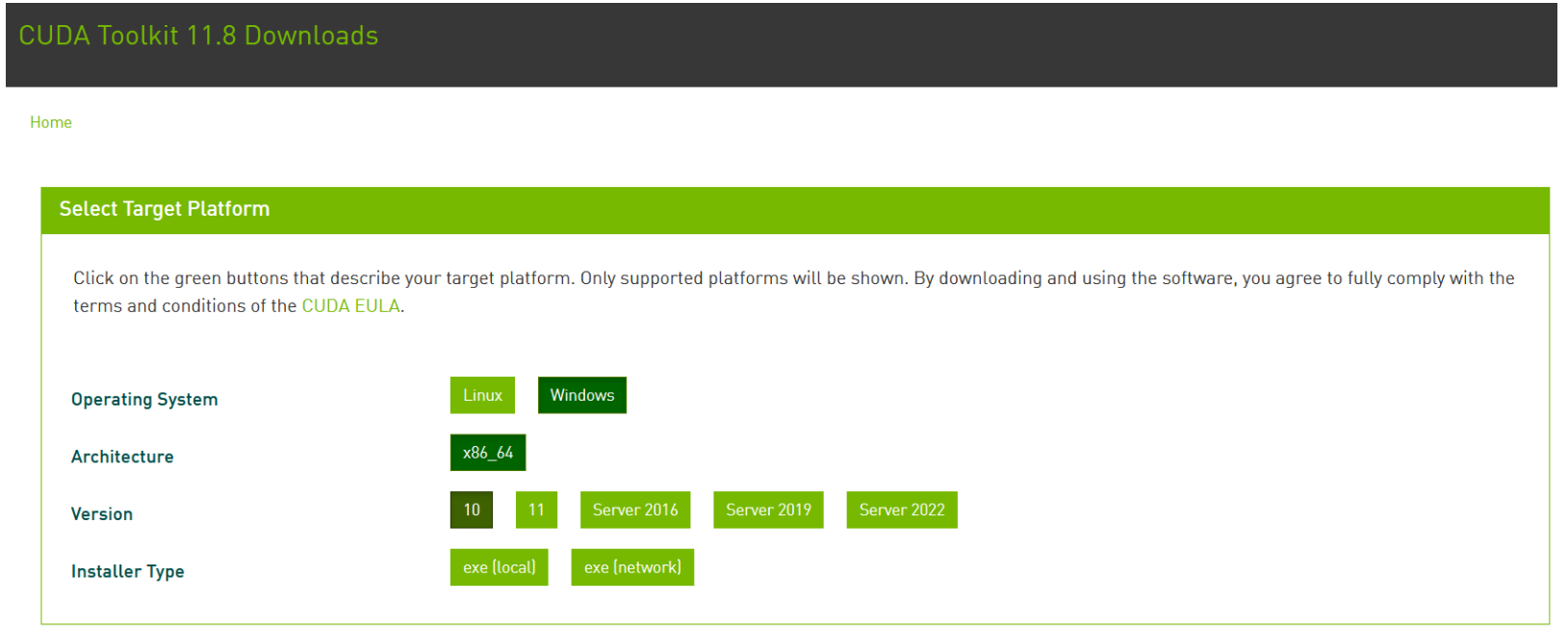
```
$ pip3 uninstall -y torch torchvision torchaudio torchtriton
$ pip3 cache purge
```

PyTorch-nightly Linux packages installed via pip during that time installed a dependency, torchtriton, which was compromised on the Python Package Index (PyPI) code repository and ran a malicious binary. This is what is known as a supply chain attack and directly affects dependencies for packages that are hosted on public package indices.

NOTE: Users of the PyTorch **stable** packages **are not** affected by this issue.**

Installing PyTorch and CUDA

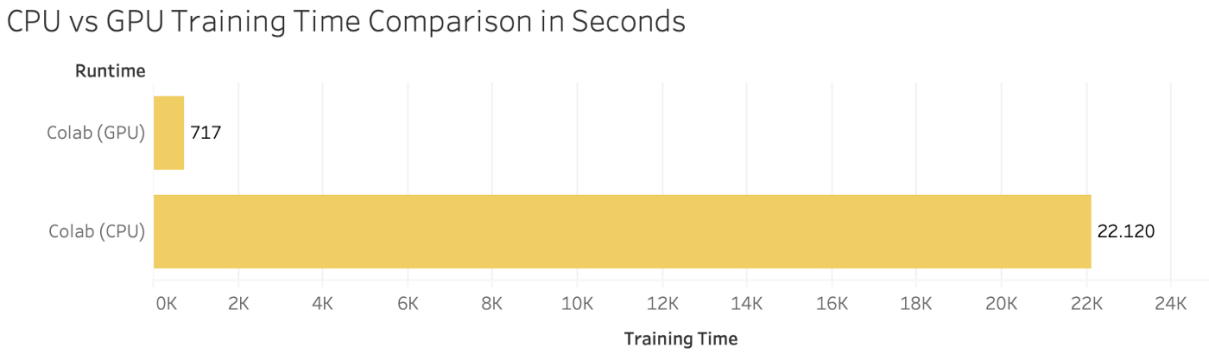
- Check if your GPU is in the list of acceptable GPUs. https://developer.nvidia.com/cuda-gpus
- If so, install CUDA (check version number matches PyTorch install!) https://developer.nvidia.com/cuda-downloads



Installing PyTorch and CUDA

- GPU not in the list of CUDA-enabled GPUs?
  Most notebooks can still run on CPU (but they might take significantly longer).

- Always the option of using Google Colab, or create an education account on AWS.



Google Colaboratory

aws educate

Checking your PyTorch and CUDA install

- **Hello World for PyTorch:** to check you have PyTorch installed correctly. The code below should run and display a tensor.

```
import torch
x = torch.rand(5, 3)
print(x)
```

```
tensor([[0.3380, 0.3845, 0.3217],
        [0.8337, 0.9050, 0.2650],
        [0.2979, 0.7141, 0.9069],
        [0.1449, 0.1132, 0.1375],
        [0.4675, 0.3947, 0.1426]])
```

- **Hello World for CUDA:** to check you have correctly installed CUDA on top of PyTorch. The code below should print *True*, as the output of *torch.cuda.is_available()*

```
import torch
torch.cuda.is_available()
```

A quick word about PyTorch 2.0

PyTorch 2.0 has been announced end of 2022 and will have several new features, including

- **torch.compile(),** which allows for faster training,
- and the ability to move parts of the PyTorch framework from C++ back into Python.

It will also be cloud-agnostic, open-source.

A quick word about PyTorch 2.0

PyTorch 2.0 is currently still in its experimental phase. **Stable PyTorch 2.0 release date is slotted for March 2023.**

(Some nightlies releases are available, but I would advise against using them for now!

Allow me to explore it a bit more first and let us discuss it again in Week 13!)

About this week (Week 1)

1. What are the **typical concepts of Machine Learning** to be used as a starting point for this course?
2. What are the **different families of problems** in Deep Learning?
3. What is the **typical structure of a Deep Learning problem**?
4. What is **linear regression** and how to implement it?
5. What is the **gradient descent algorithm** and how is it used to **train Machine Learning models**?
6. What is **polynomial regression** and how to implement it?
7. What is **regularization** and how to implement it in **Ridge regression**?

About this week (Week 1)

8. What is **overfitting** and why is it bad?
9. What is **underfitting** and why is it bad?
10. What is **generalization** and how to evaluate it?
11. What is a **train-test split** and why is it related to **generalization**?
12. What is a **sigmoid** function? What is a **logistic** function?
13. How to perform **binary classification** using a **logistic regressor** and how is it related to linear regression?

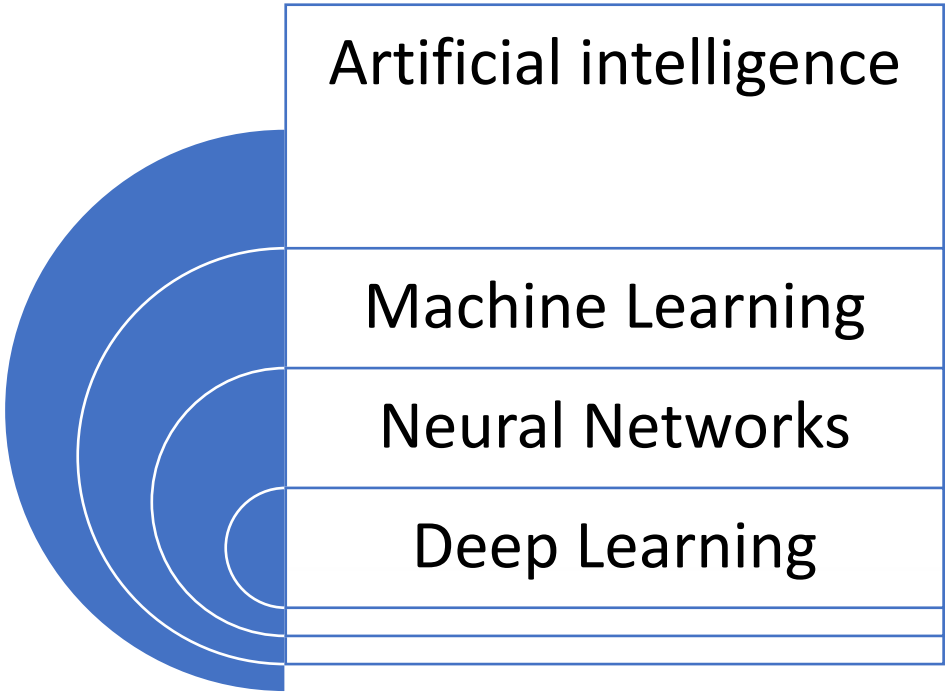
About this week (Week 1)

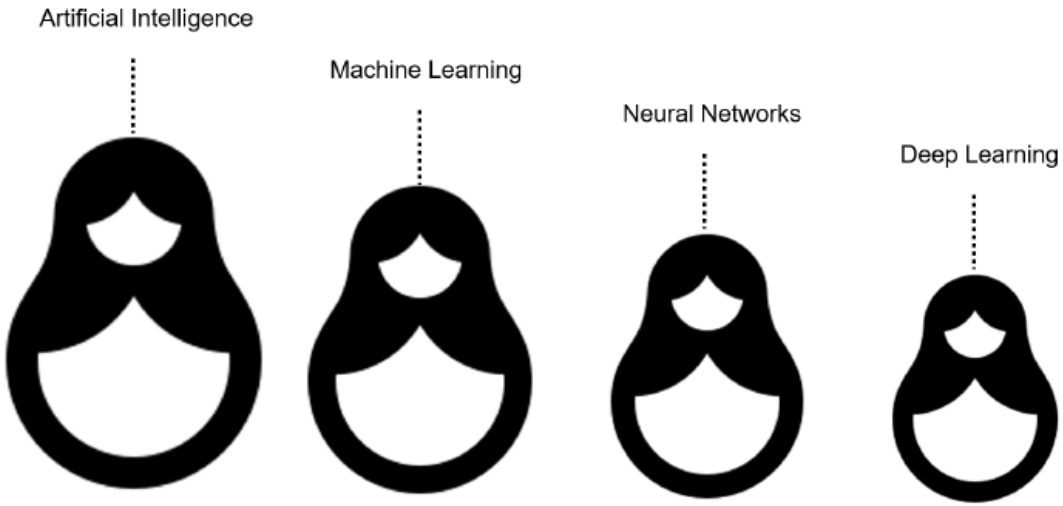
14. What are **Neural Networks** and how do they relate to the **biology of a human brain**?
15. What is a **Neuron** in a Neural Network and how does it relate to linear/logistic regression?
16. What is the **difference** between a **shallow** and a **deep neural network**?
17. How to **implement a shallow Neural Network** manually and define a **forward propagation** method for it?
18. How to **train a shallow Neural Network** using **backpropagation**? How to define **backward propagation** and **trainer** functions?

What is AI/ML/NN/DL?

Definition (Artificial Intelligence):

"In Computer Science, **Artificial Intelligence (AI)** refers to the theory and development of computer systems capable to **perform cognitive tasks** that normally require human intelligence, such as visual perception, speech recognition, decision-making, and translation between languages."





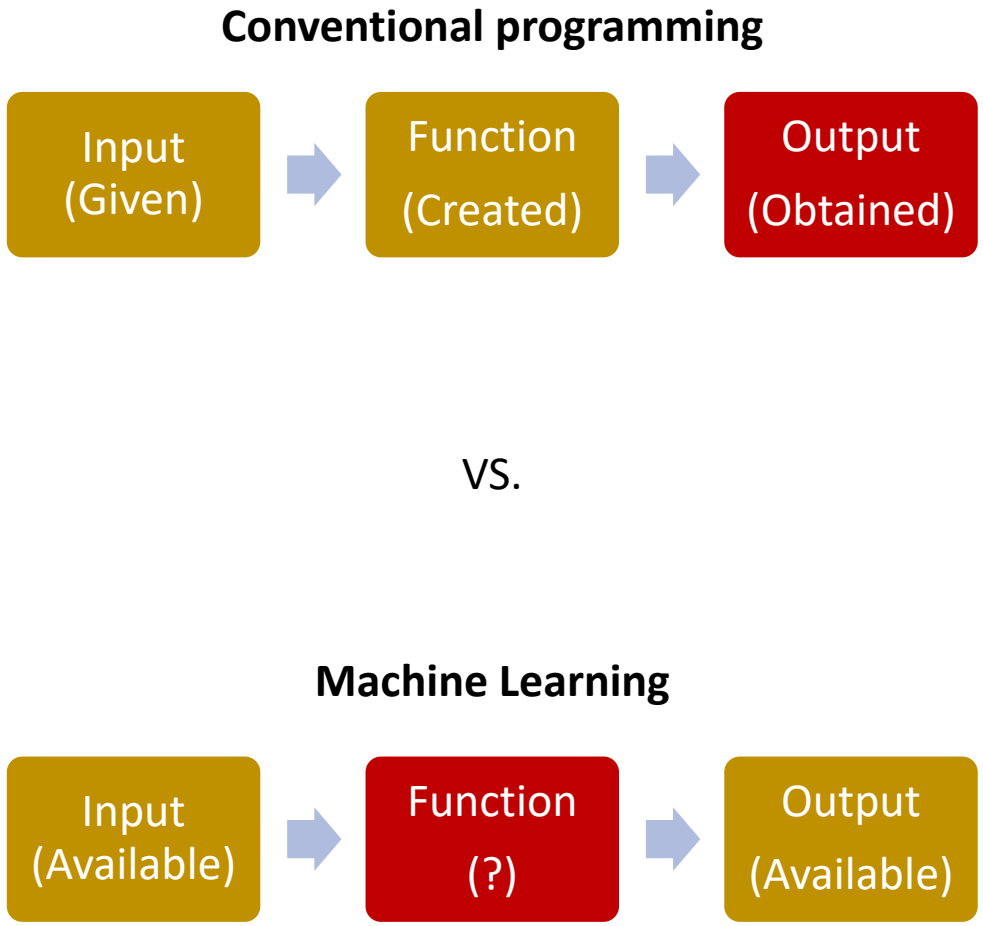
What is AI/ML/NN/DL?

Definition (Machine Learning):

In Computer Science, **Machine Learning (ML)** refers to the field of study that describes **techniques** and **algorithms** that give computers the **ability to learn without being explicitly programmed**.

Some implementations of machine learning may rely on data and neural networks.

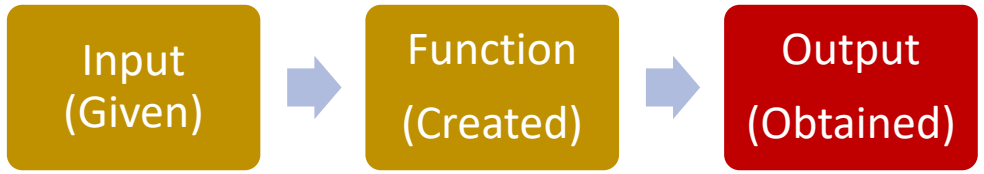
- Arthur Samuel (1959).

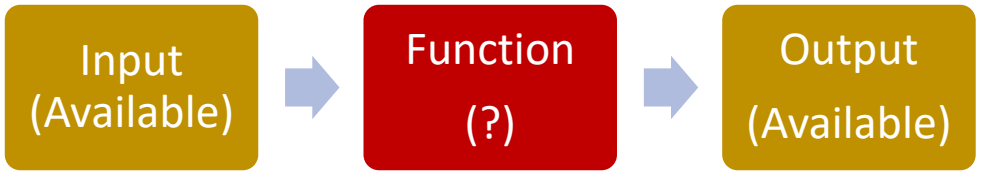


What is AI/ML/NN/DL?

- What we have done in programming so far was to design functions,
  - which would do **specific operations,**
  - and return **outputs,**
  - for any **input** we could give it.

- But sometimes, we can encounter problems where
  - we can easily find **inputs** and **expected** outputs,
  - but the **function** to be coded is **not simple** to figure out.
  - **E.g., what animal is in the picture?**

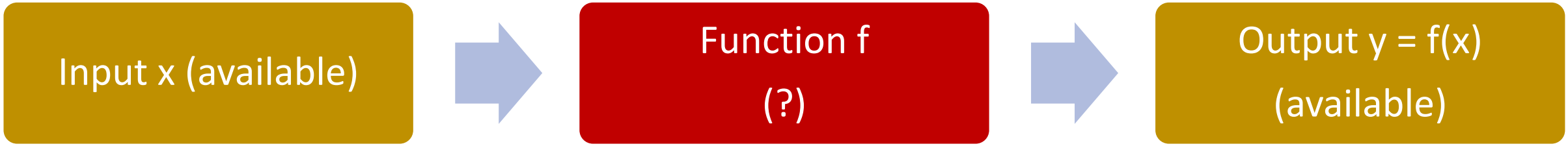




What is AI/ML/NN/DL?

E.g., what animal is in the picture?

Typical problem in **Computer Vision**, called **Image Recognition**.



It’s a cat!

What is AI/ML/NN/DL?

E.g., what animal is in the picture?

Typical problem in **Computer Vision**, called **Image Recognition**.

Input x (available) → Function f (?) → Output y = f(x) (available)

Very easy for a human…
But, how would we do it with a computer?

It’s a cat!

What is AI/ML/NN/DL?

Other scenarios of difficult functions have to do with tasks where there is **no easy closed-form expression connecting inputs to outputs.**

- **E.g., what is a good selling price for my appartment?**
- Guessing the selling price of an appartment based on its parameters (size, location, etc.) and previous sales.

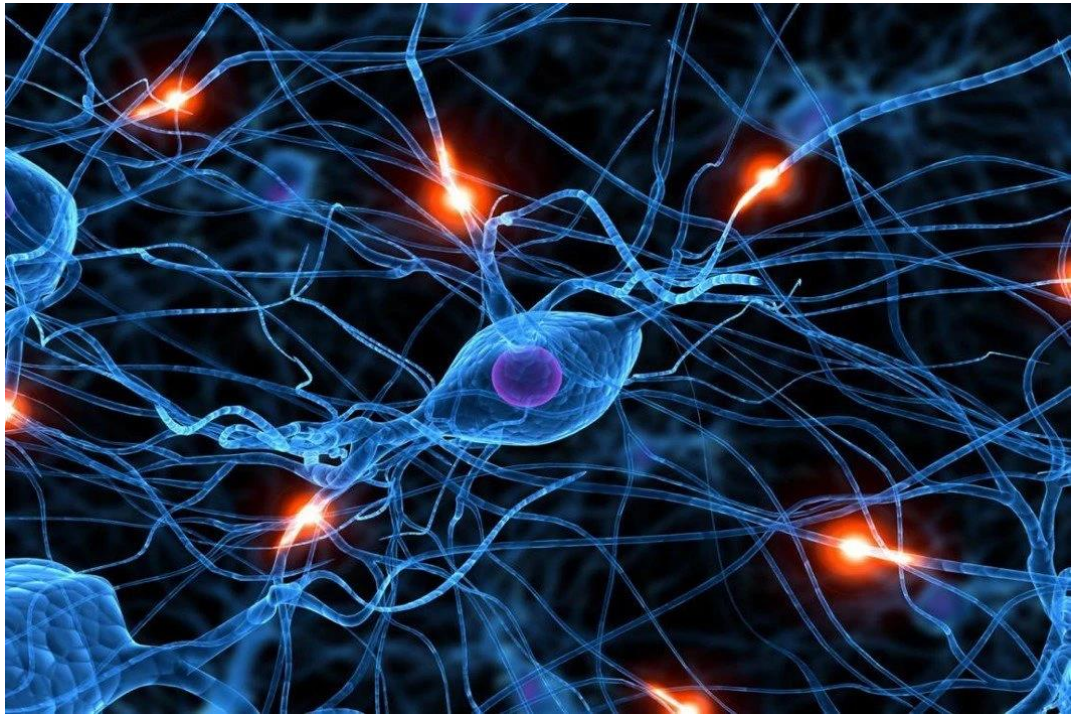
S$ 1,680,000 Negotiable

3 3 1184 sqft S$ 1,418.92 psf

Details

Property Type Executive Condominium For Sale	Floor Size 1184 sqft
Developer Tampines EC Pte Ltd	PSF S$ 1,418.92 psf
Furnishing Partially Furnished	Floor Level High Floor
Tenure 99-year Leasehold	TOP January, 2016

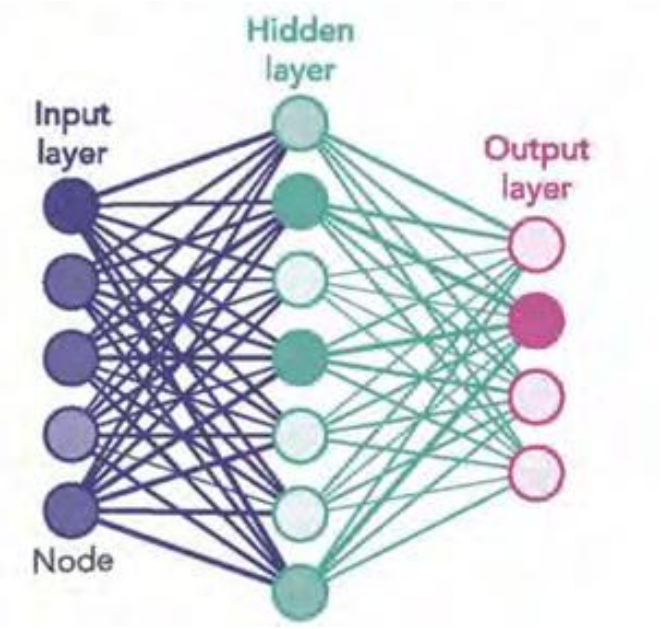
What is AI/ML/NN/DL?



Definition (Neural Networks):

Neural Networks (**NNs**) are computing systems inspired by the biological neural networks that constitute animal brains.

NNs are based on a **collection of connected units or nodes called artificial neurons**, which loosely model the neurons in a biological brain.



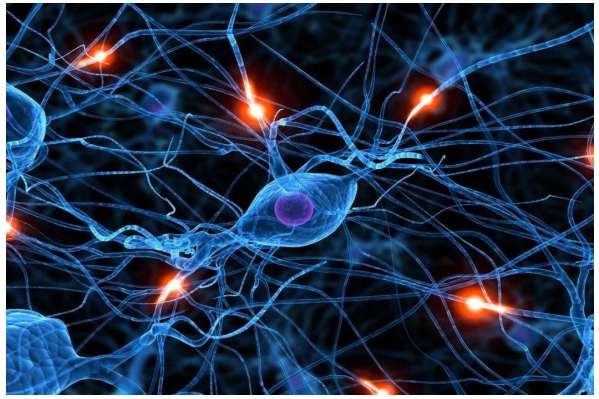
What is AI/ML/NN/DL?

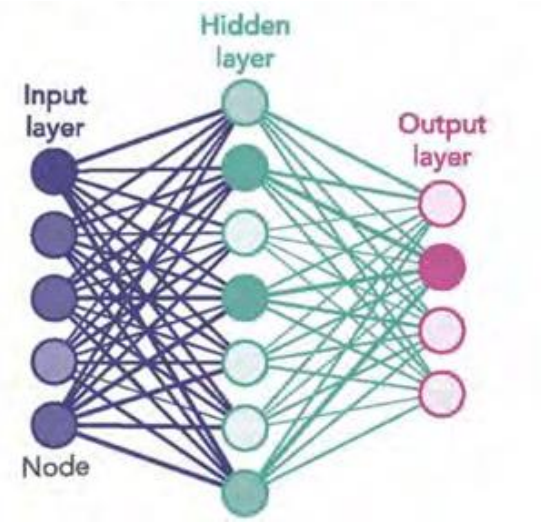
Definition (Neural Networks):

Neural Networks (**NNs**) are computing systems inspired by the biological neural networks that constitute animal brains.

NNs are based on a **collection of connected units or nodes called artificial neurons**, which loosely model the neurons in a biological brain.

Each connection, like the synapses in a biological brain, can transmit a signal to other neurons, therefore **processing any information** given as inputs to produce a final signal as output.



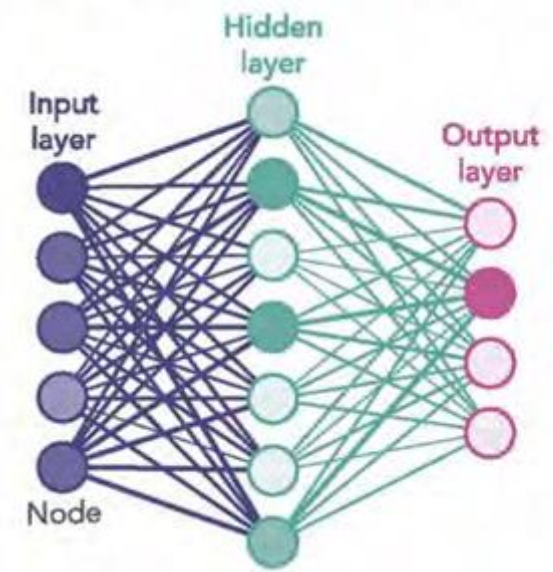


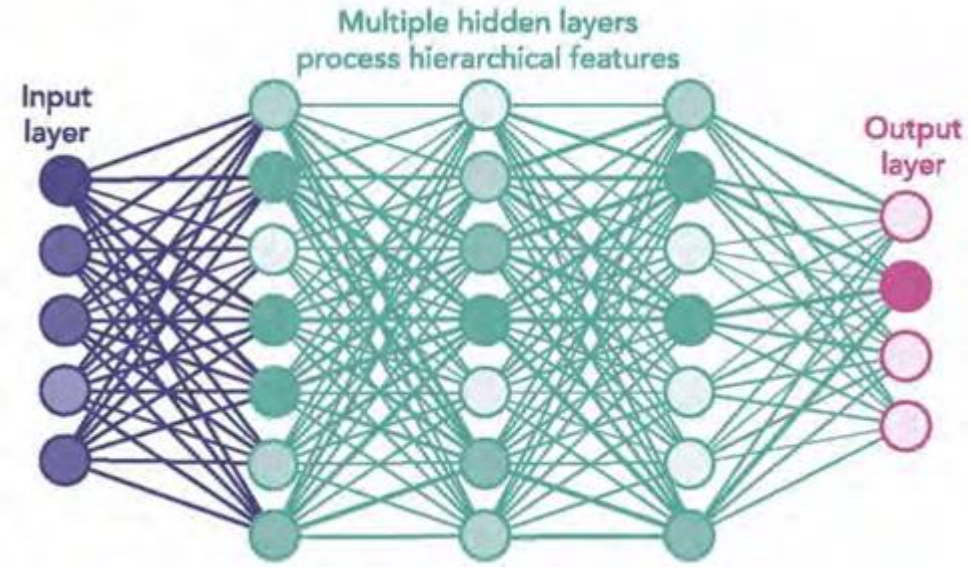
What is AI/ML/NN/DL?

Definition (Deep Learning):

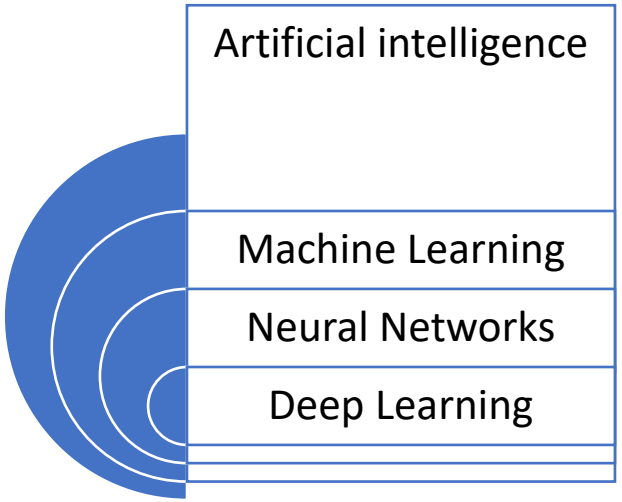
Deep Learning (DL) is a subfield of machine learning, and deep **neural networks** make up the backbone of deep learning algorithms.

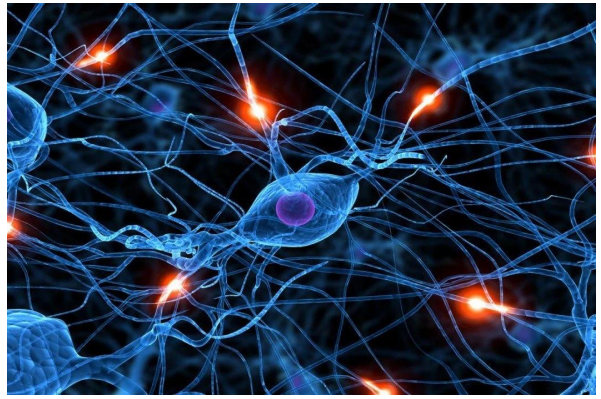
The **number of node layers,** or **depth**, of neural networks is what distinguishes a **shallow neural network** from a **deep neural network**, which must have more than three.

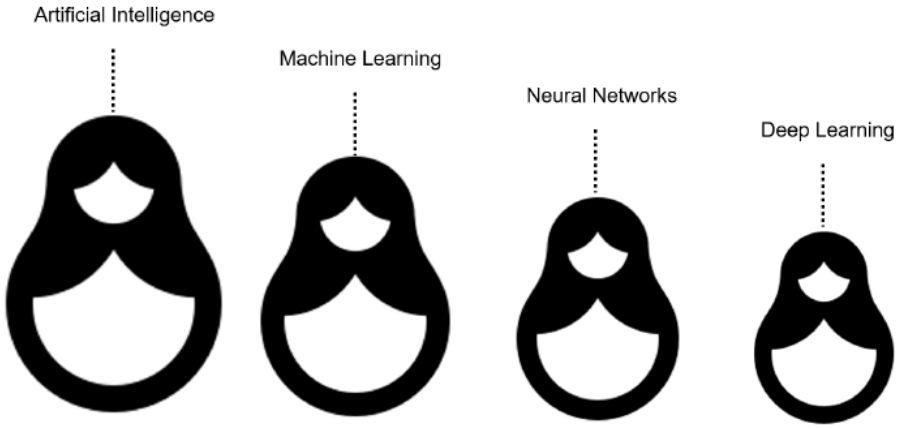


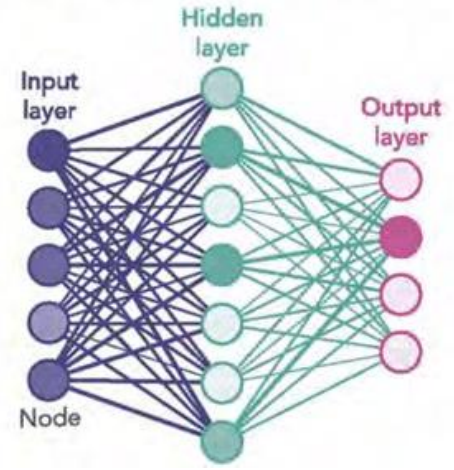


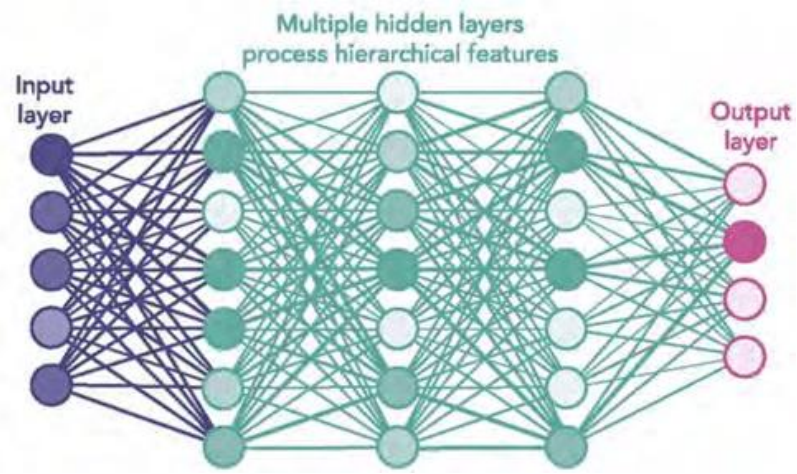
What is AI/ML/NN/DL?











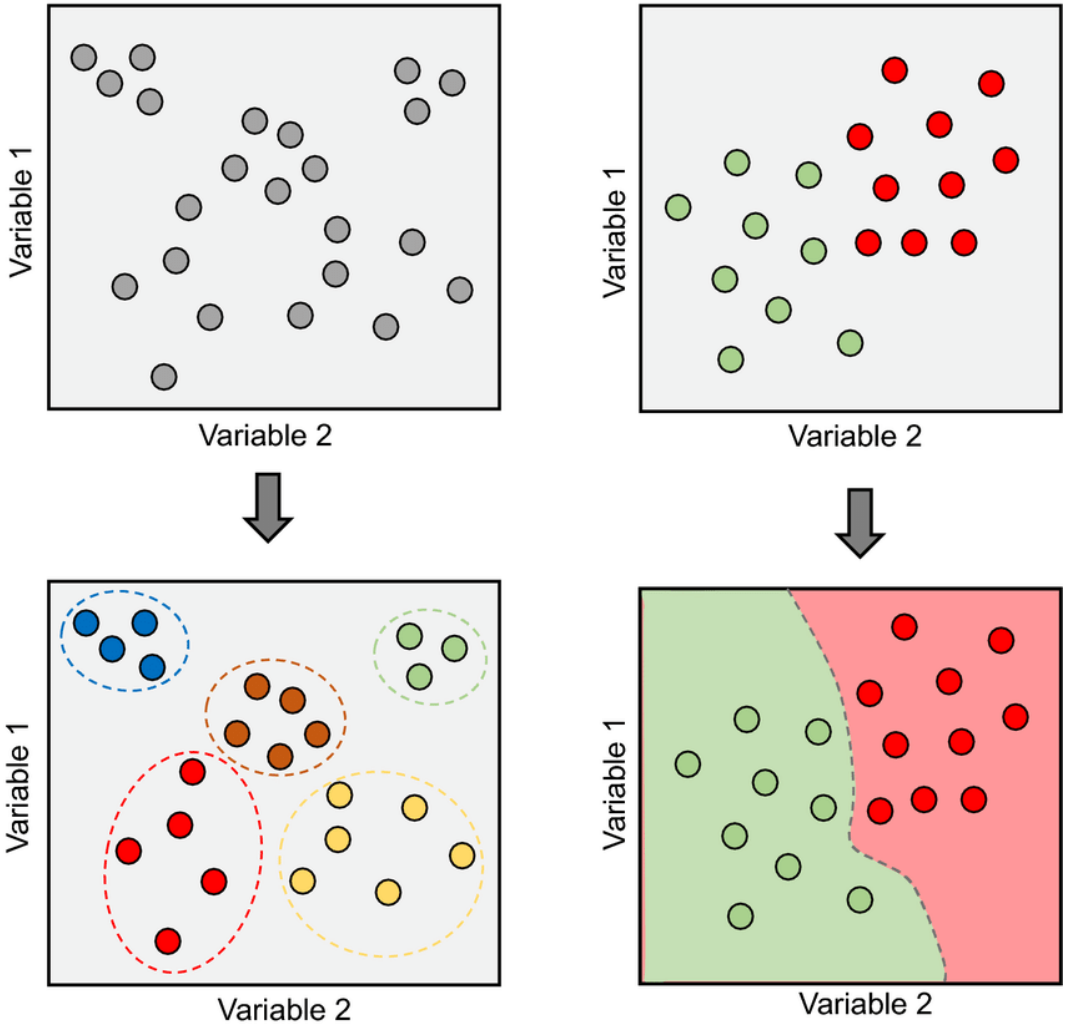
Supervised vs Unsupervised Learning

Definition (Supervised vs. Unsupervised Learning):

Supervised and **Unsupervised Learning** are the two techniques of machine learning.

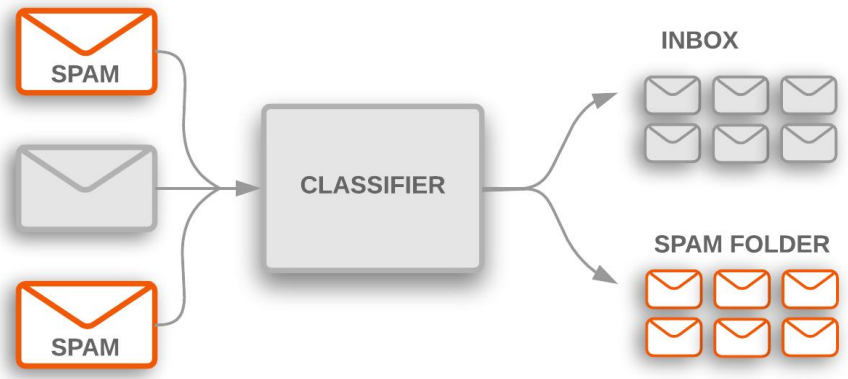
The main difference is **the need for labelled training data**:

- Supervised machine learning relies on labelled input and output data to learn and make predictions,
- while **unsupervised learning does not require labelled data.**



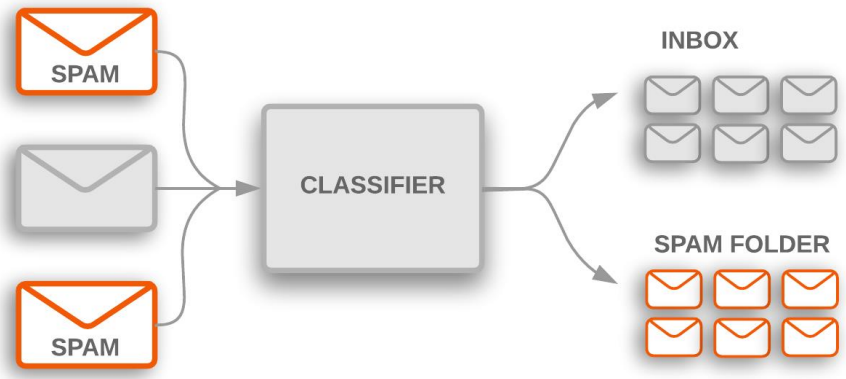
Supervised Learning Examples

- Examples of **supervised learning:** spam detection, text classification, predicting the stock market, etc.



Supervised Learning Examples

- Examples of **supervised learning:** spam detection, text classification, predicting the stock market, etc.



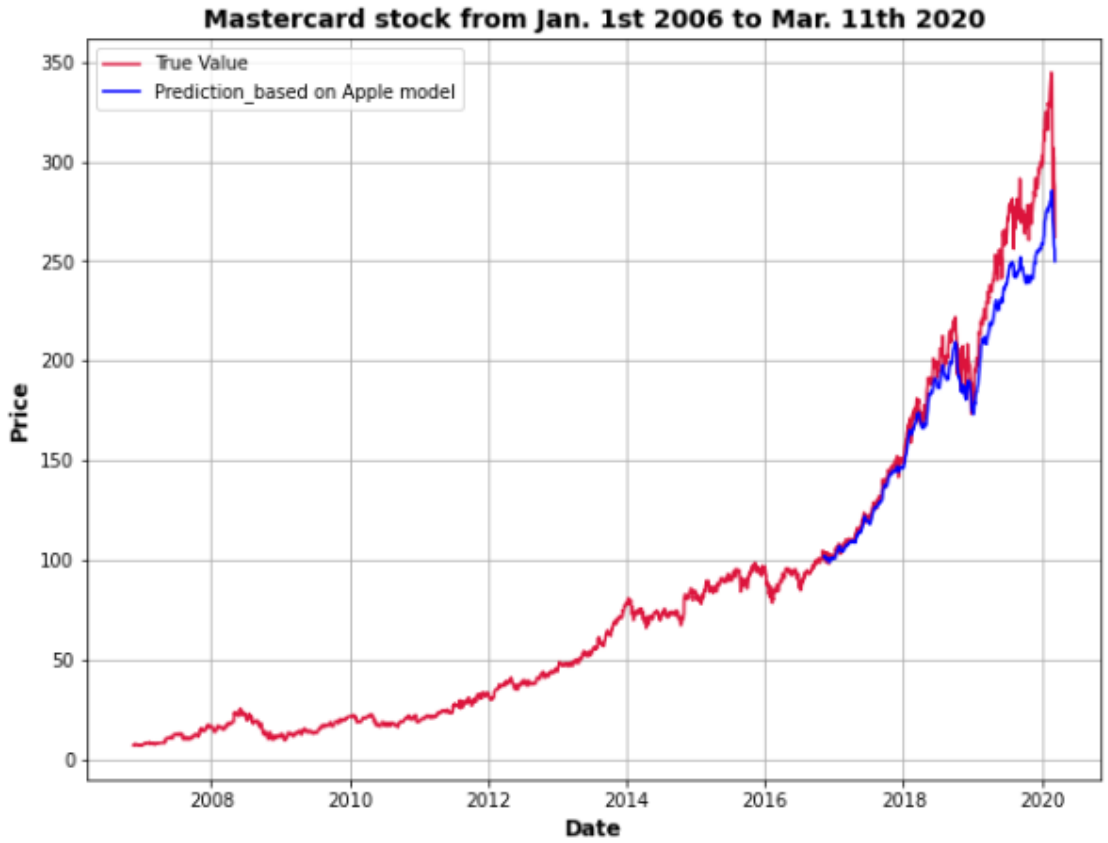


Figure 7. MAST stock price predictions using LSTM trained on AAPL

Regression vs. Classification

Definition (Regression):

Regression models are used to identify the relationships between the input and output variables.

Regression algorithms are used to **predict continuous values** such as price, salary, age, etc.

In regression, the outputs are often **continuous numerical values**.

S$ 1,680,000 Negotiable

3 3 1184 sqft S$ 1,418.92 psf

Details

Property Type Executive Condominium For Sale	Floor Size 1184 sqft
Developer Tampines EC Pte Ltd	PSF S$ 1,418.92 psf
Furnishing Partially Furnished	Floor Level High Floor
Tenure 99-year Leasehold	TOP January, 2016

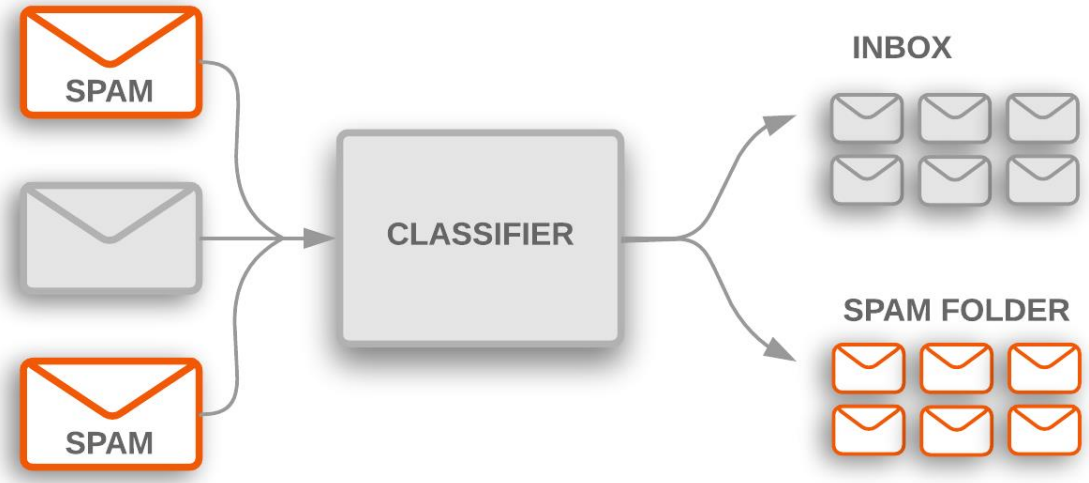
Regression vs. Classification (Supervised)

Definition (Classification):

Classification models are used to divide the samples in the dataset into different **classes**.

Classification algorithms are then used to predict/classify **discrete values**, such as Male or Female, True or False, Spam or Not Spam, etc.

In Classification, the outputs are **discrete** or **categorical values**.



Supervised Learning Examples

Example: predicting the market.

Question: is it a **regression** or a **classification** task?

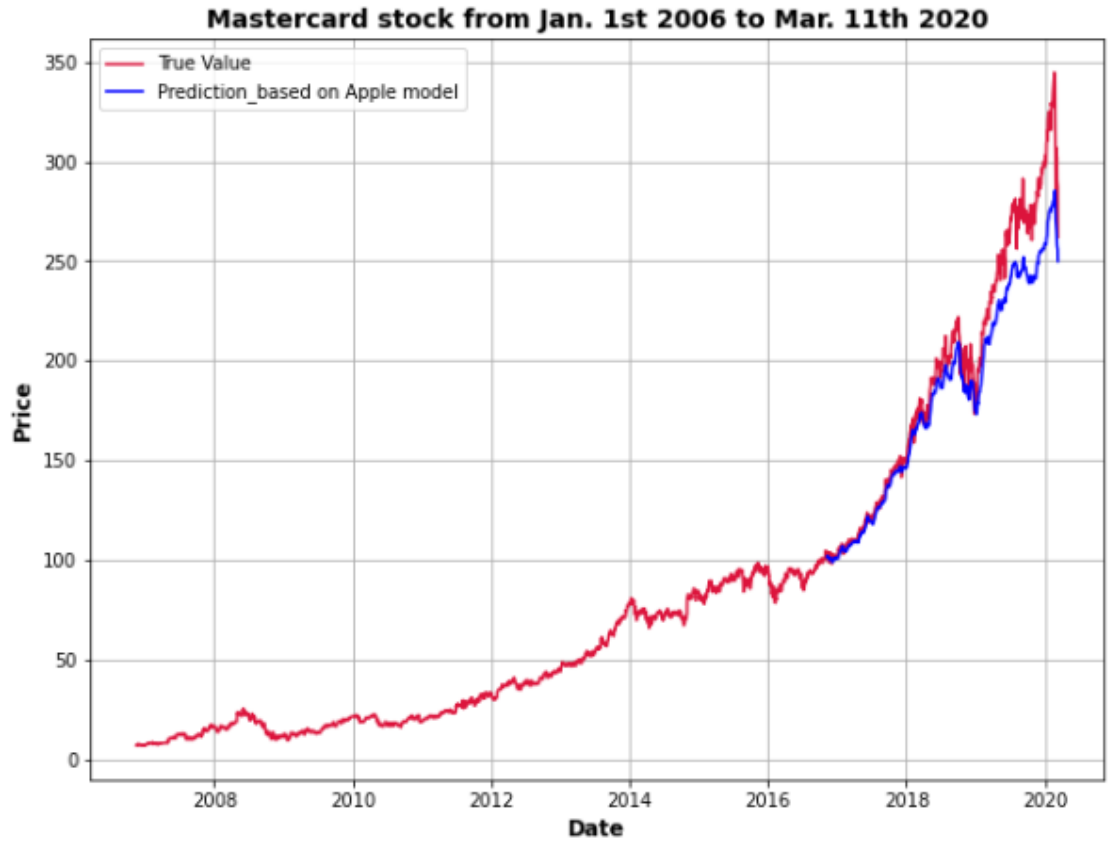


Figure 7. MAST stock price predictions using LSTM trained on AAPL

Supervised Learning Examples

Example: predicting the market.

Question: is it a **regression** or a **classification** task?

Depends.

If the output we are **predicting is the value of the stock in the future**, then probably **regression**.

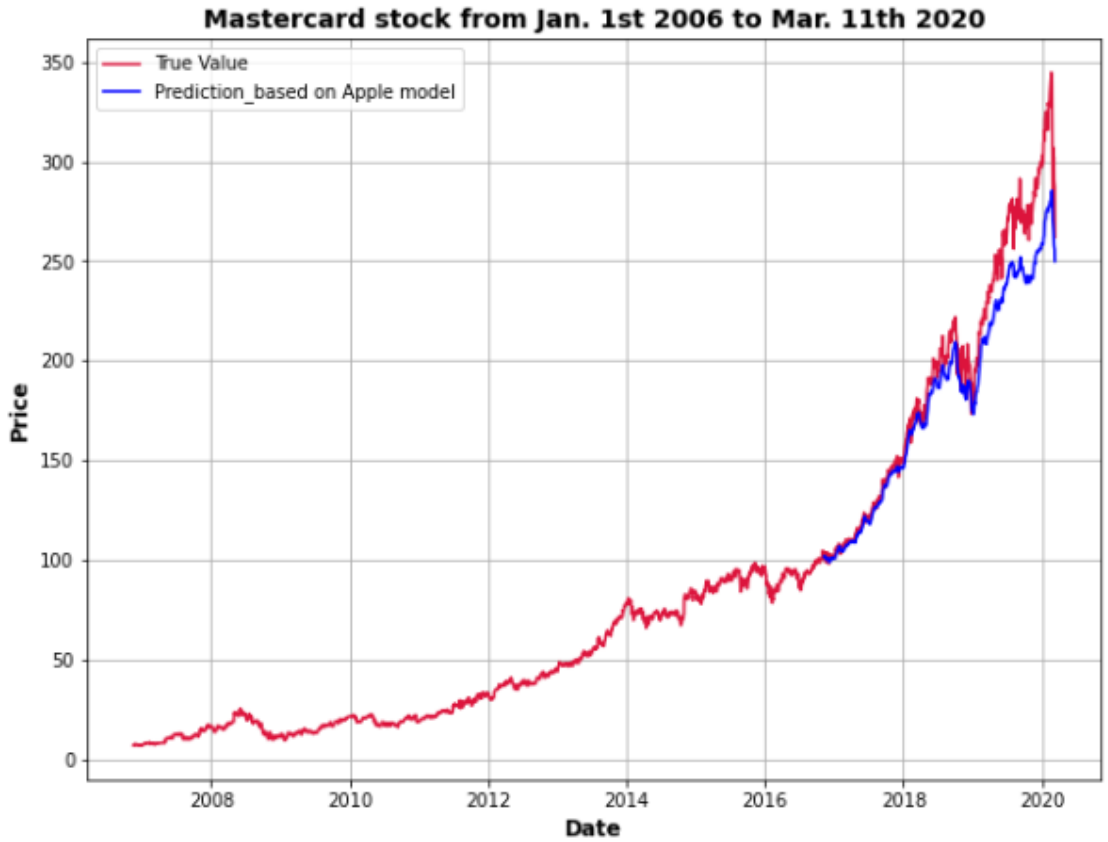


Figure 7. MAST stock price predictions using LSTM trained on AAPL

Supervised Learning Examples

Example: predicting the market.

Question: is it a **regression** or a **classification** task?

Depends.

If the output we are **predicting is the value of the stock in the future**, then probably **regression**.

If the plan is to **predict whether we should buy, sell or wait**, then probably **classification**.

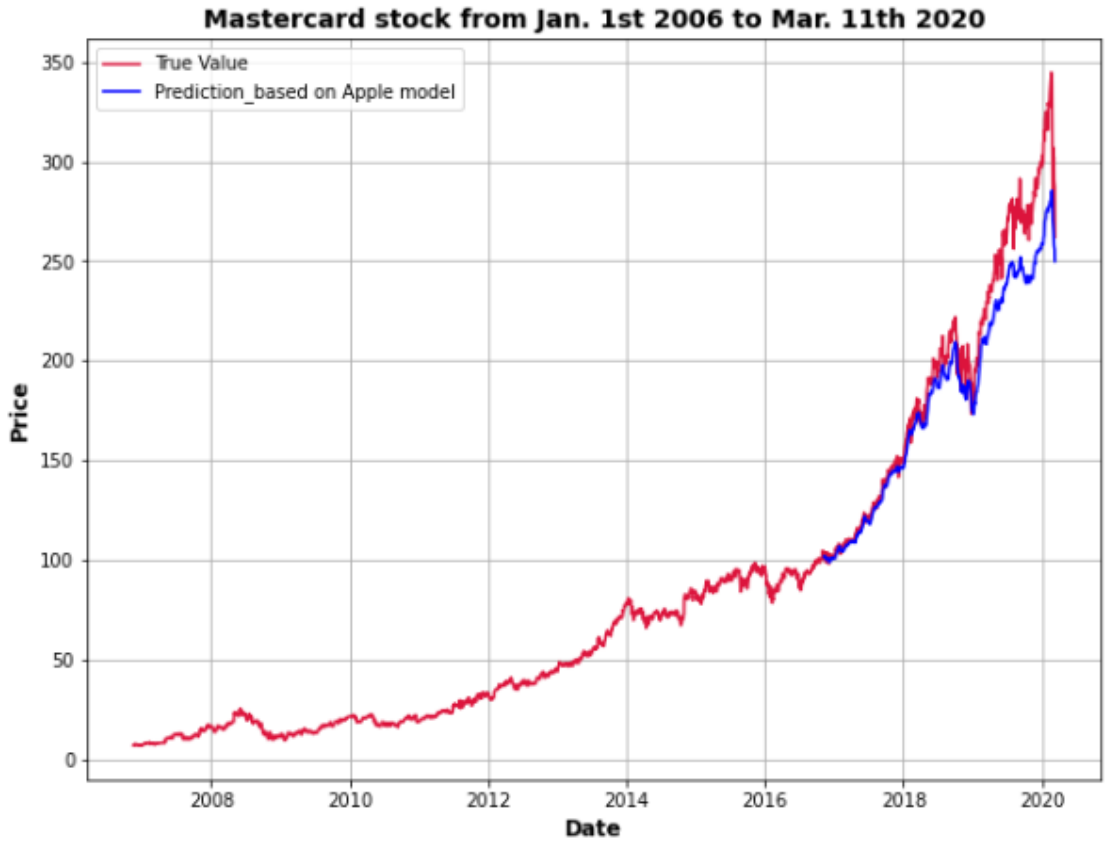
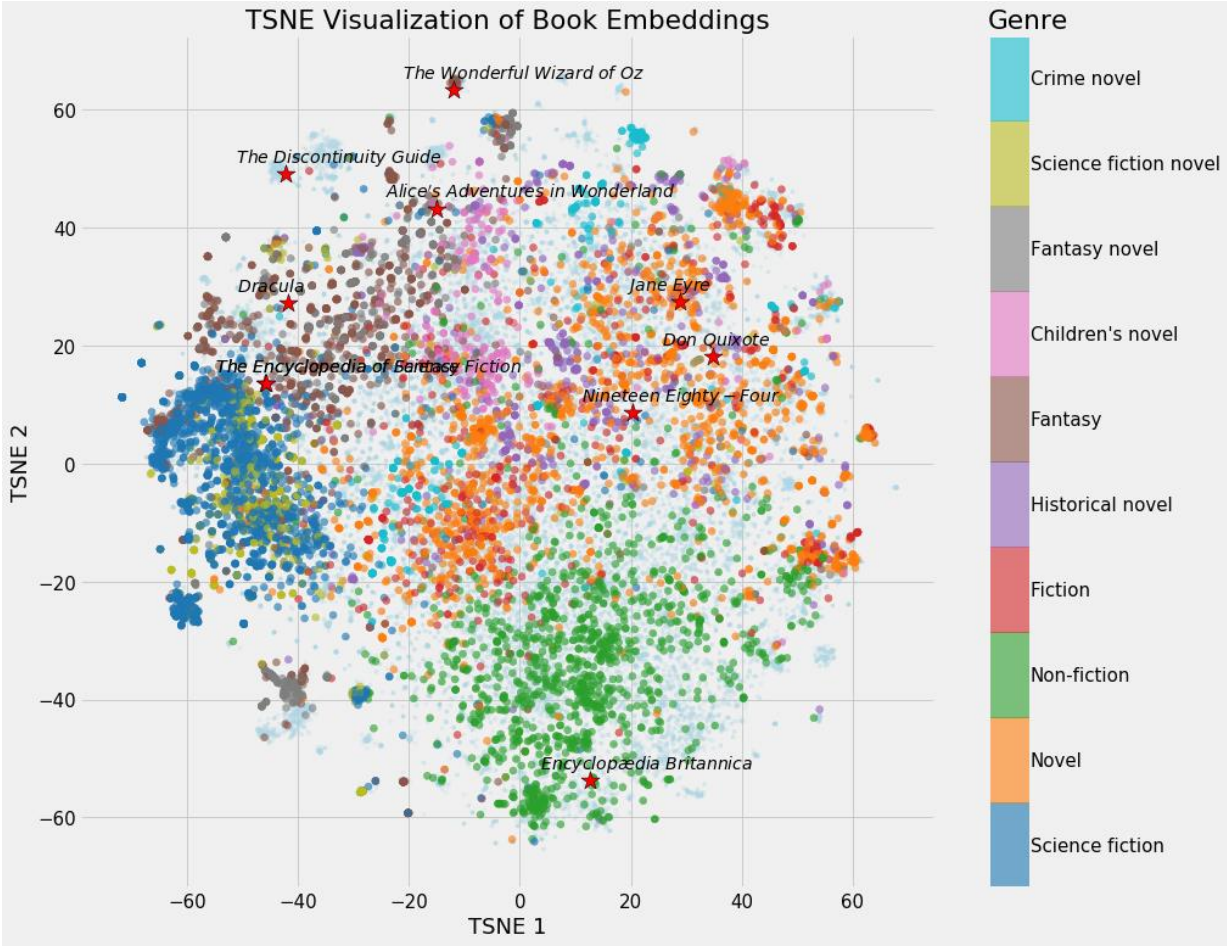


Figure 7. MAST stock price predictions using LSTM trained on AAPL

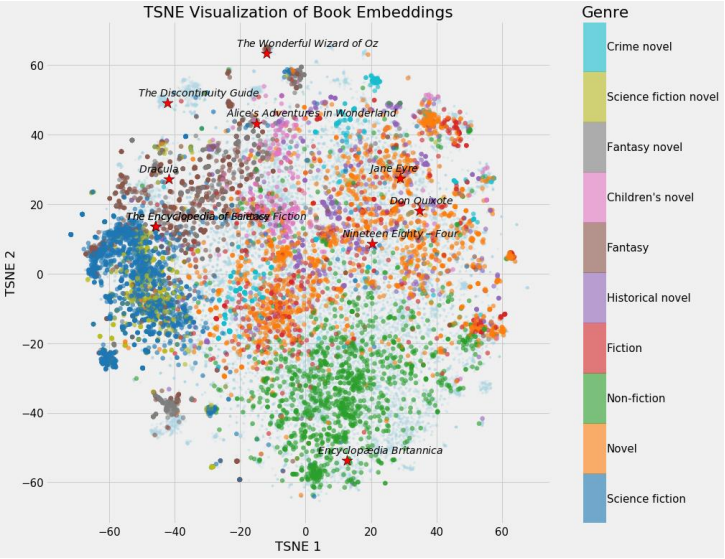
Unsupervised Learning Examples

- Examples of **unsupervised learning:** clustering customers based on their behaviors, segmenting images, and identifying topics in a document.

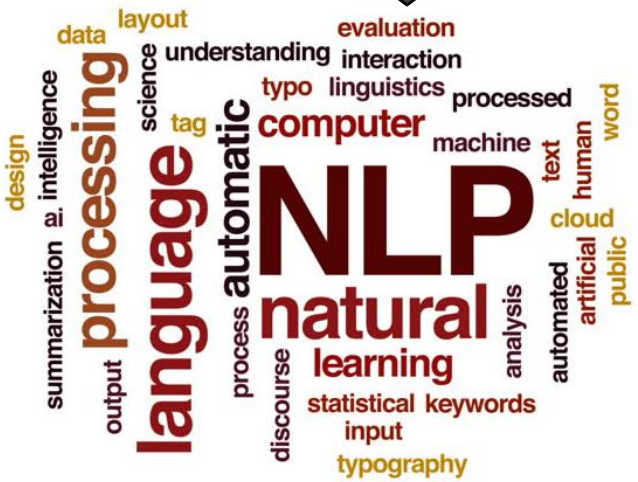


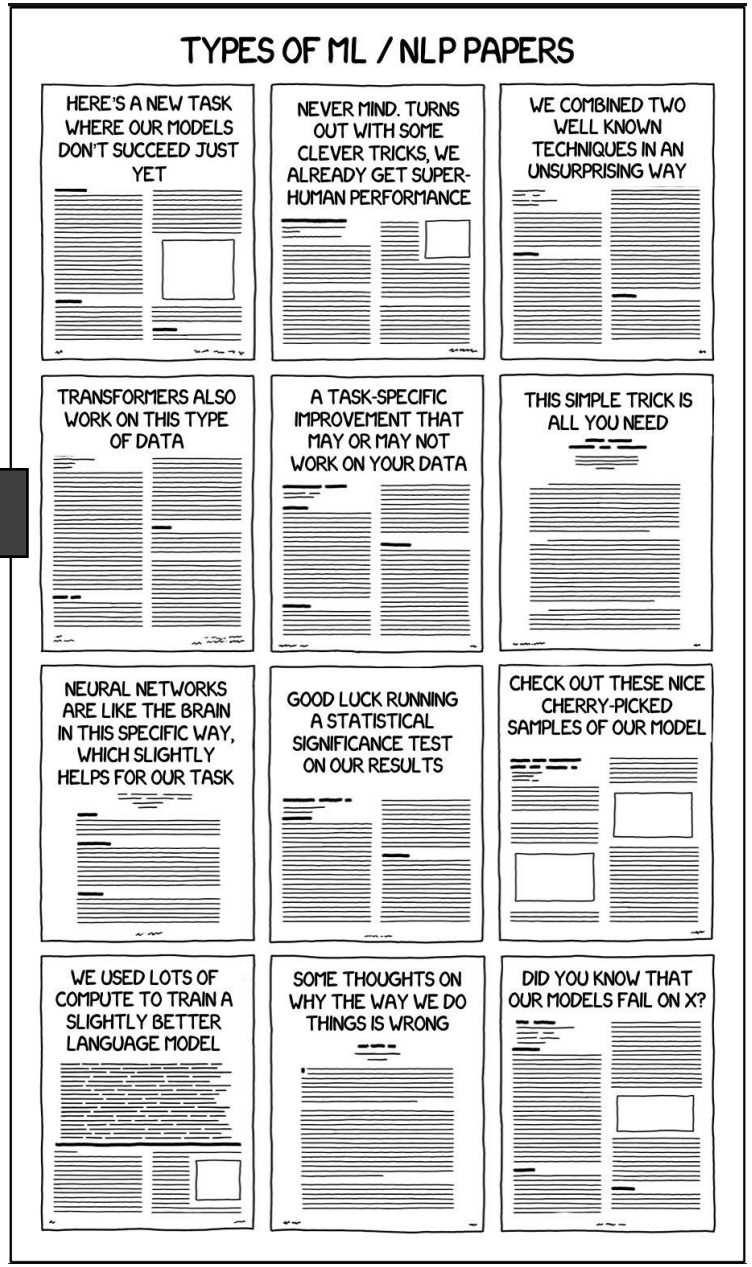
Unsupervised Learning Examples

- Examples of **unsupervised learning:** clustering customers based on their behaviors, segmenting images, and identifying topics in a document.



Identify topics that seem to appear in those papers and their prominence





Clustering vs. Association (Unsupervised)

Definition (Clustering vs. Association):

Unsupervised learning can be used for two types of problems: **Clustering** and **Association**.

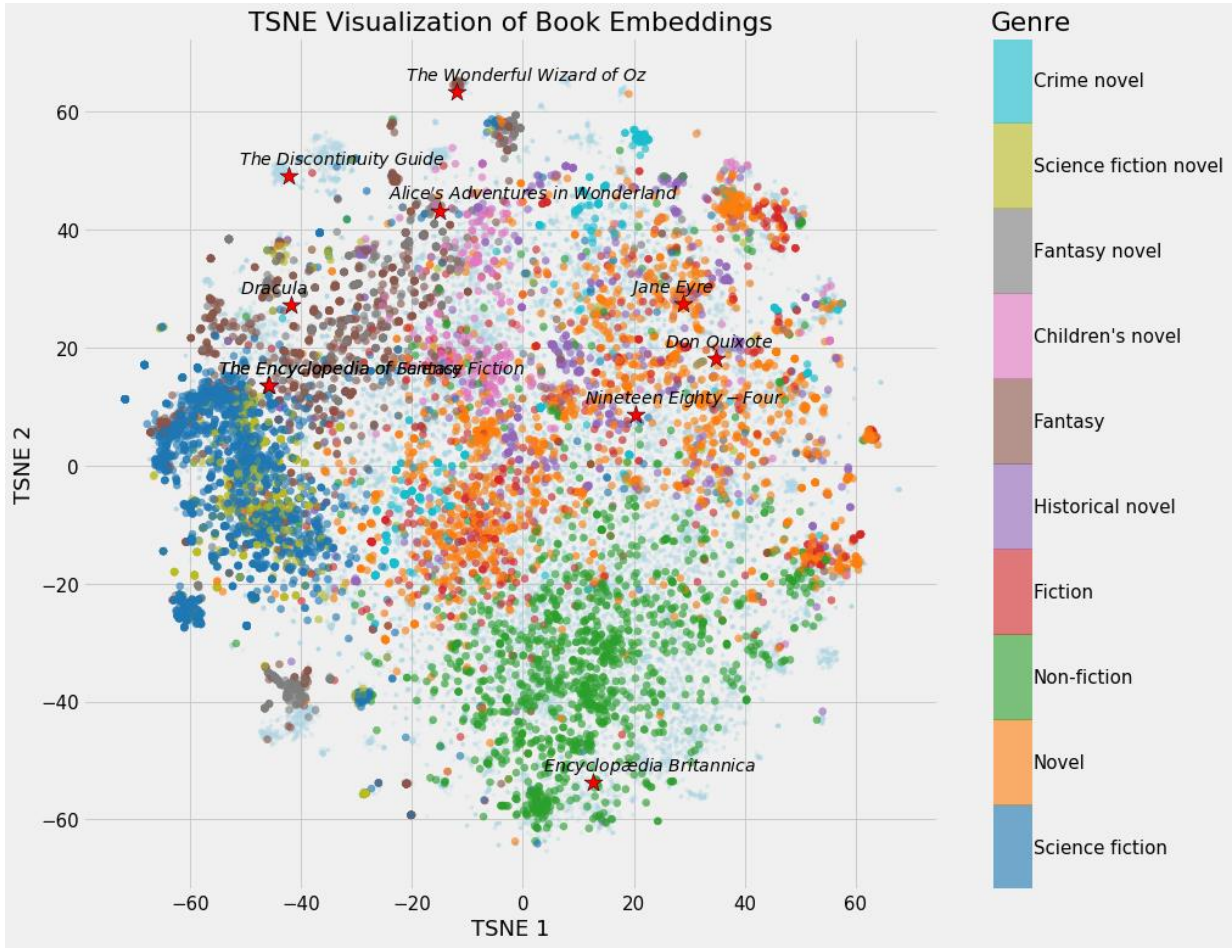
The main difference is that

- **clustering groups data based on similarity**,
- while **association looks for relationships between variables**.

Clustering vs. Association (Unsupervised)

Examples of clustering:

- Group customers into clusters based on the similarities in their purchasing behaviours, such as their spending habits or types of products they like.
- Group the catalogue of movies available on Netflix and identify what movies have in common, so you can later recommend a movie to a user based on what he/she has seen before.



Clustering vs. Association (Unsupervised)

Example of association:

- Finding the relationship between customer buying behaviors and customer satisfaction.
- Association can be used to understand how changes in customer behavior impacts customer satisfaction, such as if buying a certain product increases customer satisfaction (or the opposite).

Deep Learning and Supervised/Unsupervised

While it is possible to use Deep Learning models and Neural Networks to perform both Supervised and Unsupervised tasks…

The vast majority of DL/NN tasks fall in the category of supervised learning.

(And we will therefore mostly focus on that)

Elements of a ML problem

Definition (Elements of a Machine Learning problem):

Machine learning problems should be well-defined, i.e. the following elements must be clearly identified:

1. **Task (T)** at hand,
2. **Inputs (I)** and **Ouputs (O),**
3. **Model (M)** definition and its **trainable parameters (P),**
4. **Loss (L)** function to measure the performance on said task and dataset.

S$ 1,680,000 Negotiable

3 3 1184 sqft S$ 1,418.92 psf

Details

Property Type Executive Condominium For Sale	Floor Size 1184 sqft
Developer Tampines EC Pte Ltd	PSF S$ 1,418.92 psf
Furnishing Partially Furnished	Floor Level High Floor
Tenure 99-year Leasehold	TOP January, 2016

Elements of a ML problem

In the example on the right…

1. **Task (T)** at hand:

- Predict price of an apartment based on the apartment features.
- Using a dataset consisting of previously seen appartments to learn the selling prices of apartments.
- It is therefore a supervised regression task.

S$ 1,680,000 Negotiable

3 3 1184 sqft S$ 1,418.92 psf

Details

Property Type Executive Condominium For Sale	Floor Size 1184 sqft
Developer Tampines EC Pte Ltd	PSF S$ 1,418.92 psf
Furnishing Partially Furnished	Floor Level High Floor
Tenure 99-year Leasehold	TOP January, 2016

Elements of a ML problem

In the example on the right…

2. **Inputs (I)** and **Ouputs (O):**

- Inputs will be a list of apartment features (e.g. number of bathrooms, number of bedrooms, surface size, etc.).
- Output will be the selling price in SGD or surface price in SGD per sqft.

S$ 1,680,000 Negotiable

3 3 1184 sqft S$ 1,418.92 psf

Details

Property Type Executive Condominium For Sale	Floor Size 1184 sqft
Developer Tampines EC Pte Ltd	PSF S$ 1,418.92 psf
Furnishing Partially Furnished	Floor Level High Floor
Tenure 99-year Leasehold	TOP January, 2016

Elements of a ML problem

In the example on the right…

3. **Model (M)** definition and its **trainable parameters (P):**

- Could be a linear or polynomial regression model (to be discussed this week). Or a neural network of some sort. Or even something else (check your ML course!)

4. **Loss (L)** function to measure the performance on said task and dataset:

- To be discussed later, as this might depend on the model used.

S$ 1,680,000 Negotiable

3 3 1184 sqft S$ 1,418.92 psf

Details

Property Type Executive Condominium For Sale	Floor Size 1184 sqft
Developer Tampines EC Pte Ltd	PSF S$ 1,418.92 psf
Furnishing Partially Furnished	Floor Level High Floor
Tenure 99-year Leasehold	TOP January, 2016

Our first toy example

Our first toy example: simplified version of the apartment price prediction.

1. **Task (T)** at hand:

- Predict price of an apartment based on the apartment features.
- We will generate a “fake” dataset consisting of appartments with features and selling prices.
- Supervised regression task.

S$ 1,680,000 Negotiable

3 3 1184 sqft S$ 1,418.92 psf

Details

Property Type Executive Condominium For Sale	Floor Size 1184 sqft
Developer Tampines EC Pte Ltd	PSF S$ 1,418.92 psf
Furnishing Partially Furnished	Floor Level High Floor
Tenure 99-year Leasehold	TOP January, 2016

Our first toy example

Our first toy example: simplified version of the apartment price prediction.

2. **Inputs (I)** and **Ouputs (O):**

- Inputs will be a list of apartment features, which here will only consist of the surface of the apartment, in square meters (sqm).
- Output will be the selling price in millions of SGD (mSGD).

S$ 1,680,000 Negotiable

3 3 1184 sqft S$ 1,418.92 psf

Details

Property Type Executive Condominium For Sale	Floor Size 1184 sqft
Developer Tampines EC Pte Ltd	PSF S$ 1,418.92 psf
Furnishing Partially Furnished	Floor Level High Floor
Tenure 99-year Leasehold	TOP January, 2016

Our first toy example (mock dataset)

Have a look at Week 1, Notebook 1.

We will create a dataset, which uses surfaces as inputs, with values drawn randomly between 40 and 150 sqm.

We will assume that the price is randomly generated, using a uniform distribution, with

- average 100000 + 14373 times the surface in sqm,
- and a "variance" of +/- 10% around the average value.

Our first toy example (mock dataset)

```python
# Two random generator functions to generate a mock dataset.
# 1. Surfaces randomly generated as uniform between min_surf and max_surf
def surface(min_surf, max_surf):
    return round(np.random.uniform(min_surf, max_surf), 2)
```

```python
# 2. Price is 100000 + 14373 times the surface in sqm, +/- 10%
# (randomly shifted to give the dataset some diversity).
def price(surface):
    # Note: this will return the price in millions of SGD.
    return round((100000 + 14373*surface)*(1 + np.random.uniform(-0.1, 0.1)))/1000000
```

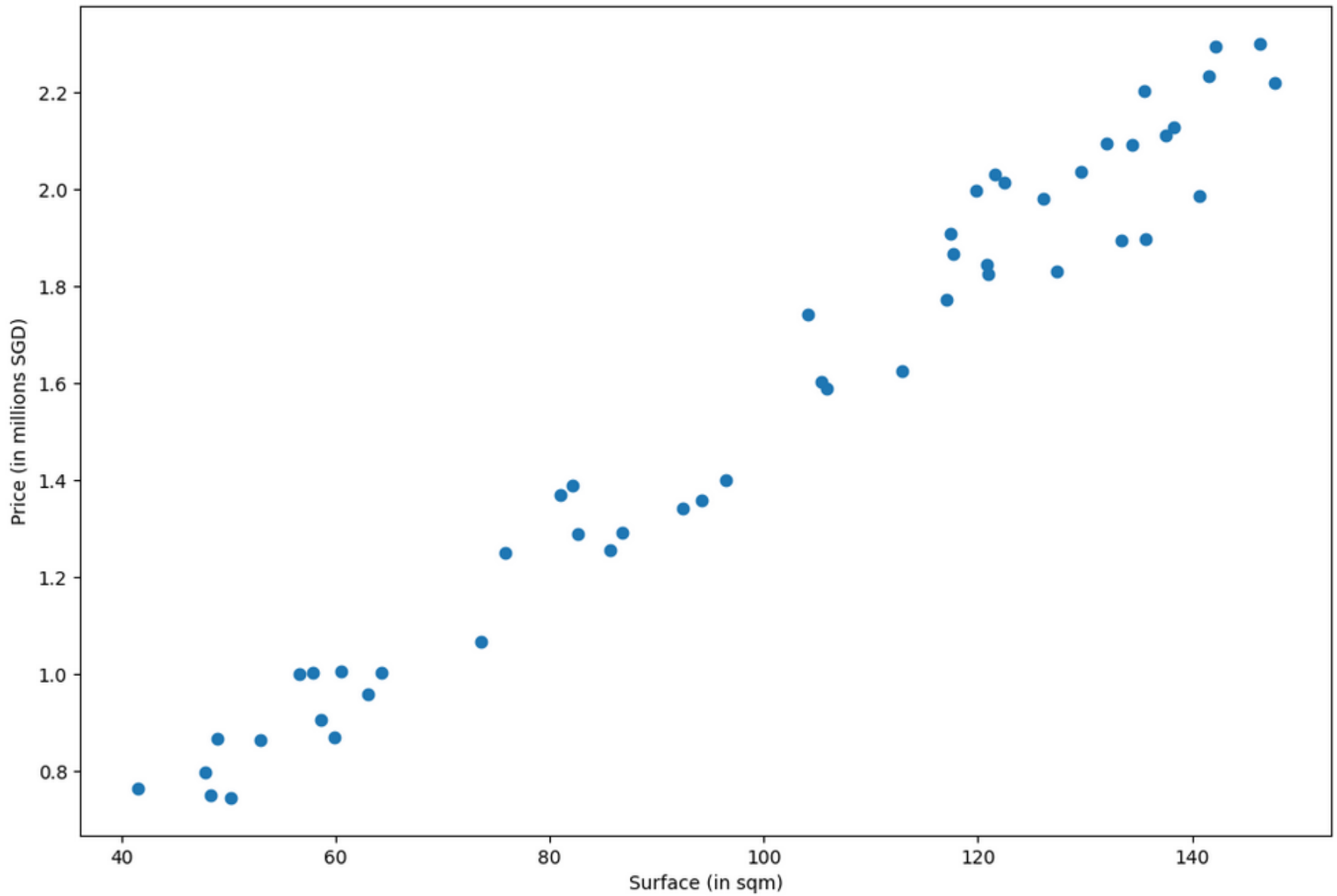
```python
# 3. Generate dataset function
def generate_datasets(n_points, min_surf, max_surf):
    x = np.array([surface(min_surf, max_surf) for _ in range(n_points)])
    y = np.array([price(i) for i in x])
    return x, y
```

Our first toy example (mock dataset)

```python
# 4. Dataset generation (n_points points will be generated).
# Surfaces randomly generated as uniform between 40sqm and 150sqm.
# We will use a seed for reproducibility.
min_surf = 40
max_surf = 150
np.random.seed(27)
n_points = 50
inputs, outputs = generate_datasets(n_points, min_surf, max_surf)
print(inputs)
print(outputs)
```

```
[ 86.83 129.6  120.89 135.48  82.17 147.74 138.25  63.07 121.6  112.95
 137.55 134.38 122.42 135.72  60.54  75.81  81.02 127.31  56.62  58.69
  48.93  73.57 126.16  57.92  47.77 117.12  59.91 105.88  85.68  96.49
  64.27 119.81 133.44 142.18 120.95  92.42  94.22 105.4   48.36  52.92
 146.31 104.17  50.17  41.5  132.06 140.63 117.5   82.57 117.63 141.56]
[1.290893 2.034977 1.84501  2.201767 1.389632 2.218678 2.127228 0.959054
 2.029469 1.623609 2.111638 2.09194  2.012386 1.89553  1.004256 1.250228
 1.368325 1.830127 1.000719 0.906513 0.867629 1.065907 1.979544 1.001403
 0.796199 1.771816 0.867878 1.587176 1.25434  1.40047  1.002361 1.9972
 1.894479 2.293443 1.823577 1.340533 1.358613 1.602167 0.750759 0.863093
 2.30035  1.741468 0.7448   0.763732 2.093772 1.986868 1.90702  1.289541
 1.86578  2.231851]
```

```python
# Display dataset and see that there is a rather clear linear trend.
plt.figure(figsize = (12, 8))
plt.scatter(inputs, outputs)
plt.xlabel("Surface (in sqm)")
plt.ylabel("Price (in millions SGD)")
plt.show()
```



Our first toy example

Our first toy example: simplified version of the apartment price prediction.

3. **Model (M)** definition and its **trainable parameters (P):**

- Here, our model will consist of a **linear regression** model.

S$ 1,680,000 Negotiable

3 3 1184 sqft S$ 1,418.92 psf

Details

Property Type Executive Condominium For Sale	Floor Size 1184 sqft
Developer Tampines EC Pte Ltd	PSF S$ 1,418.92 psf
Furnishing Partially Furnished	Floor Level High Floor
Tenure 99-year Leasehold	TOP January, 2016

Our first model, the linear regression

Definition (Linear Regression):

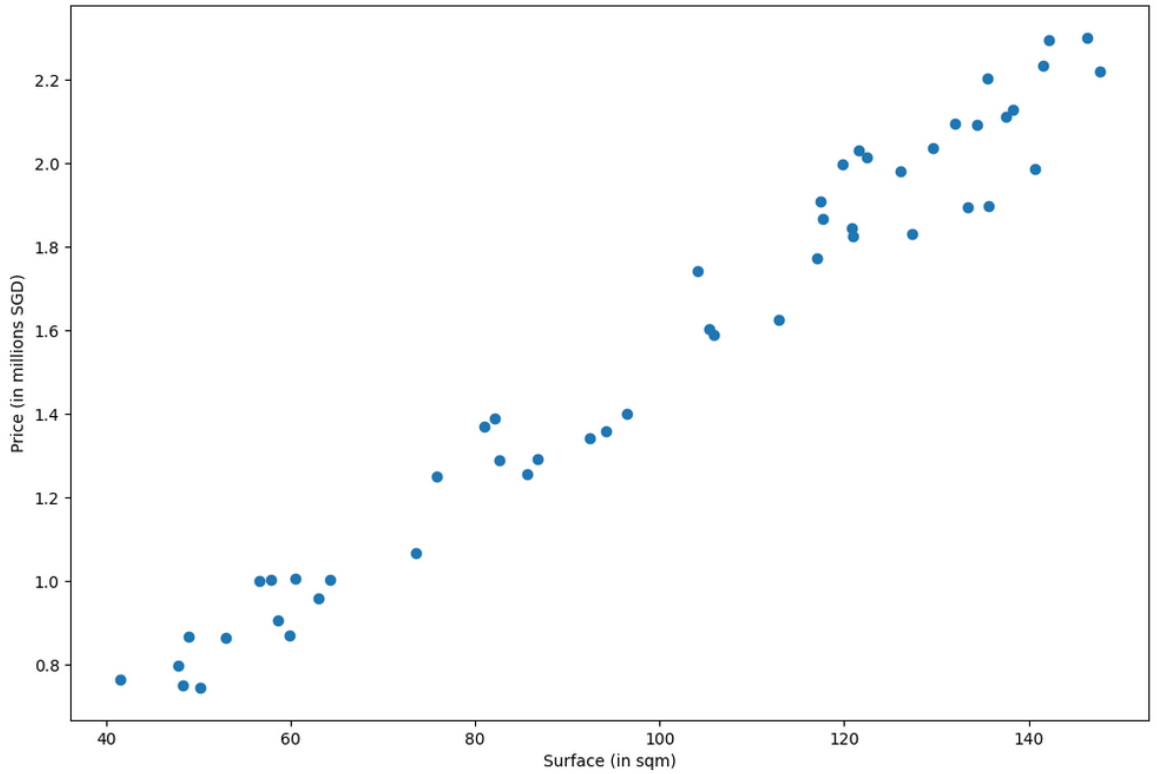
Linear Regression is a model, which **assumes that there is a linear relationship between inputs and outputs.**

It therefore consists of two trainable parameters (a, b), to be freely chosen.

These will connect any input x_i to its respective output y_i, with the following equation:

$$y_i \approx a\, x_i + b$$

```
# Display dataset and see that there is a rather clear linear trend.
plt.figure(figsize = (12, 8))
plt.scatter(inputs, outputs)
plt.xlabel("Surface (in sqm)")
plt.ylabel("Price (in millions SGD)")
plt.show()
```



Our first model, the linear regression

Definition (Linear Regression):

Linear Regression is a model, which **assumes that there is a linear relationship between inputs and outputs.**

It therefore consists of two trainable parameters (a, b), to be freely chosen.

These will connect any input x_i to its respective output y_i, with the following equation:

$$y_i \approx a\, x_i + b$$

```python
# Linear regression has two trainable parameters (a and b).
# Other parameters, like min_surf, max_surf, n_points will
# help get points for the upcoming matplotlib displays.
def linreg_matplotlib(a, b, min_surf, max_surf, n_points = 50):
    x = np.linspace(min_surf, max_surf, n_points)
    y = a*x + b
    return x, y
```

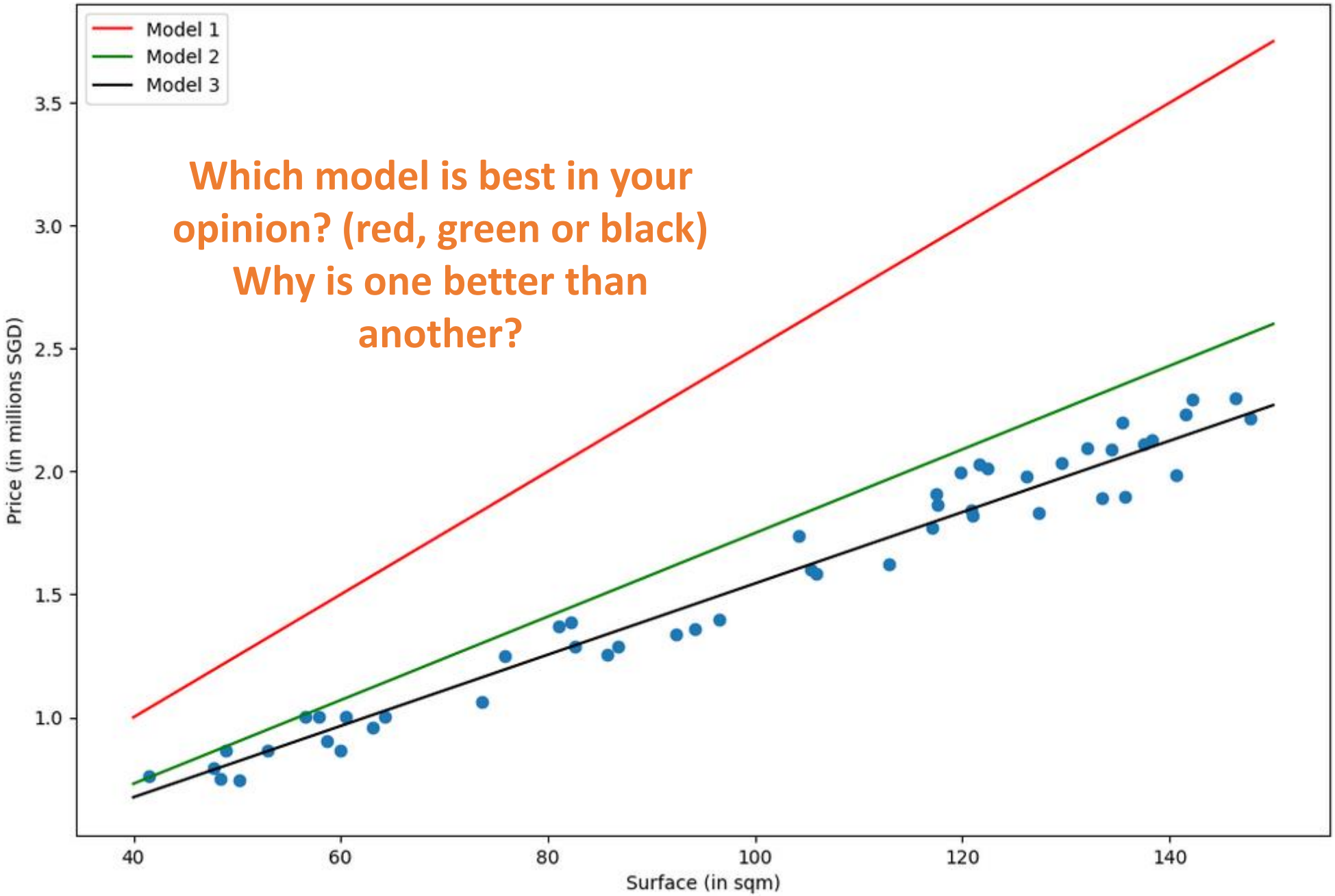
Let us try different values of (a, b)

Model 1

Model 2

Model 3

```
# Display dataset
plt.figure(figsize = (12, 8))
plt.scatter(inputs, outputs)
plt.xlabel("Surface (in sqm)")
plt.ylabel("Price (in millions SGD)")

# Add some linreg (model 1)
a1 = 25000/1000000
b1 = 0
linreg_dataset1_inputs, linreg_dataset1_outputs = linreg_matplotlib(a1, b1, min_surf, max_surf, n_points)
plt.plot(linreg_dataset1_inputs, linreg_dataset1_outputs, 'r', label = "Model 1")

# Another linreg (model 2)
a2 = 17000/1000000
b2 = 50000/1000000
linreg_dataset2_inputs, linreg_dataset2_outputs = linreg_matplotlib(a2, b2, min_surf, max_surf, n_points)
plt.plot(linreg_dataset2_inputs, linreg_dataset2_outputs, 'g', label = "Model 2")

# A final linreg (model 3)
a3 = 14500/1000000
b3 = 95000/1000000
linreg_dataset3_inputs, linreg_dataset3_outputs = linreg_matplotlib(a3, b3, min_surf, max_surf, n_points)
plt.plot(linreg_dataset3_inputs, linreg_dataset3_outputs, 'k', label = "Model 3")

# Display
plt.legend(loc = 'best')
plt.show()
```



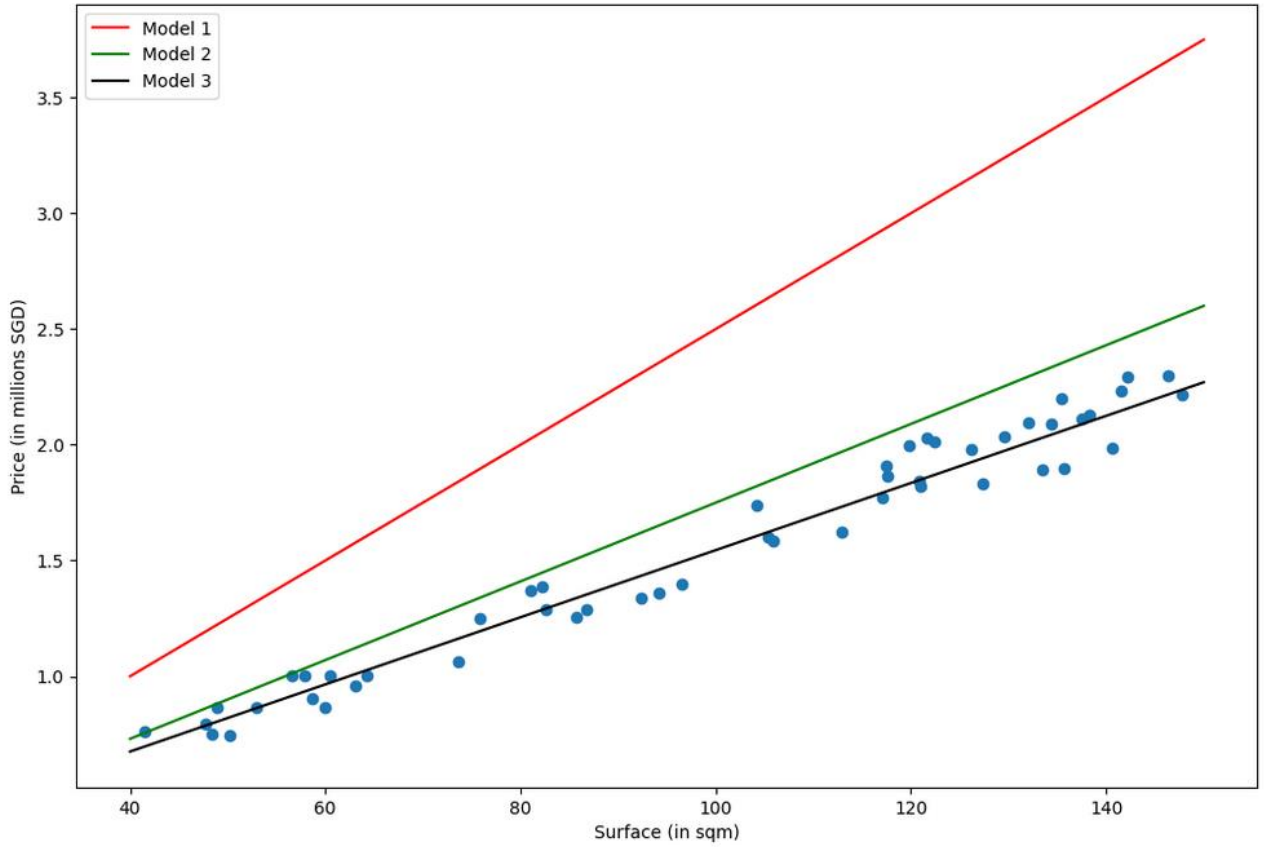
The need for a loss function

Which model is best in your opinion? (red, green or black) Why is one better than another?

Telling which model is best can prove difficult, especially for close values of (a, b).

Good news, there is one last element we have not used.

4. **Loss (L)** function to measure the performance of the model on said task and dataset.



The need for a loss function

Definition (Loss function):

A **loss function** (also known as a **cost function**) is a mathematical function that measures the difference between the predicted output of a model and the true output we should be matching.

It is used to

- **evaluate the performance of the model,**
- And more specifically **evaluate how good our choice of trainable parameters** (here, the parameters are a and b) are at fitting the data.

It is used to guide the optimization process during **training**.

The need for a loss function

Definition (The Mean Square Error loss function):

In our linear model, a good loss function we could use consists of the **Mean Square Error (MSE) loss function**.

The MSE is calculated by calculating the square difference between:

- The **prediction** $\boldsymbol{p_i = ax_i + b}$ formed by the model for some given inputs x_i,
- The **true value** $\boldsymbol{y_i}$ that we should be matching.

We then repeat this operation for every sample i, and average those errors together, i.e.

$$L(a, b, x, y) = \frac{1}{N}\sum_{i=1}^{N}(ax_i + b - y_i)^2$$

The need for a loss function

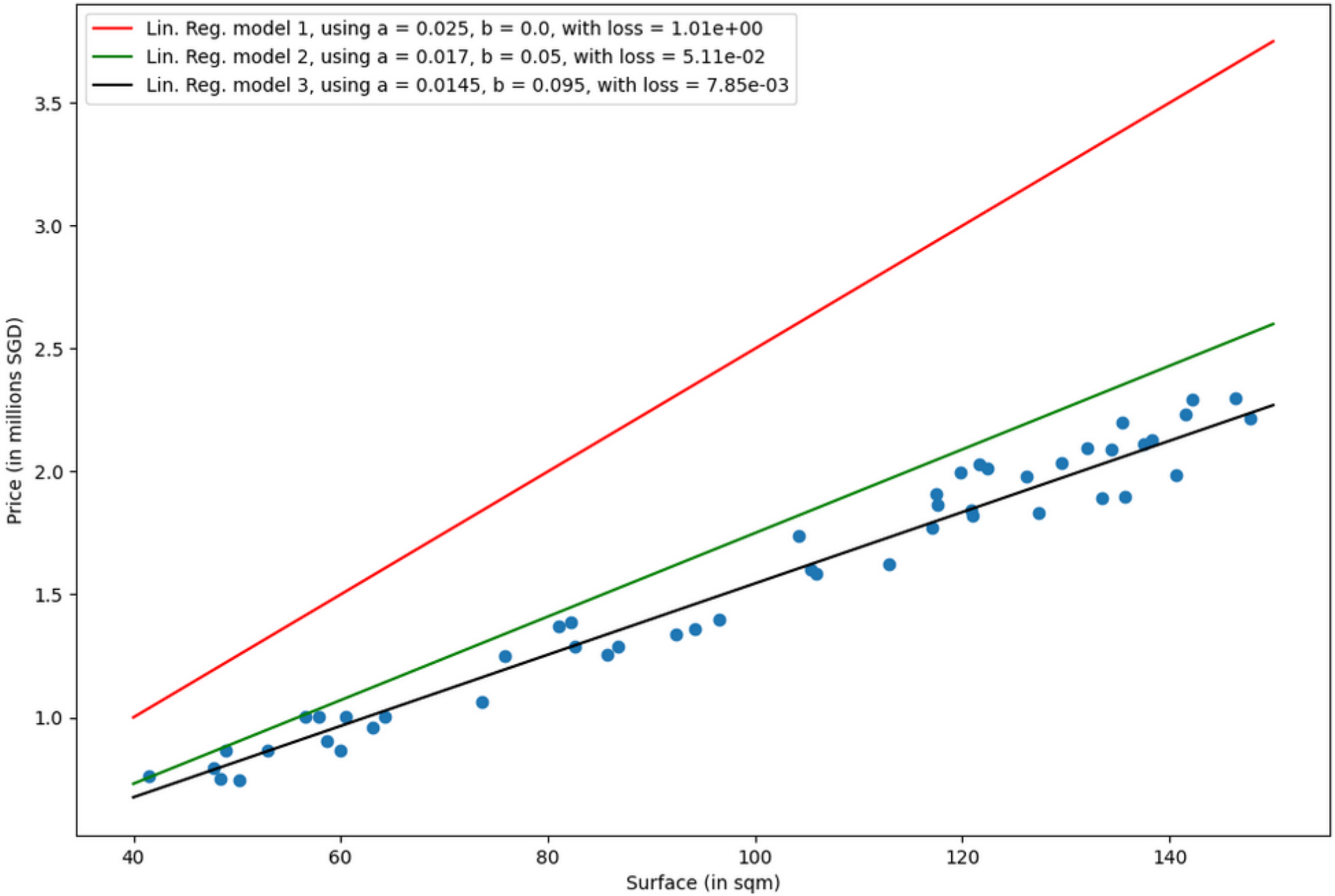
Recall the definition of the MSE loss function

$$L(a, b, x, y) = \frac{1}{N}\sum_{i=1}^{N}(ax_i + b - y_i)^2$$

In [9]:
```
# Mean square error as a loss function
# Displaying loss using exponential notation (XXXe-YYY)
def loss_mse(a, b, x, y):
    val =  np.sum((y - (a*x + b))**2)/x.shape[0]
    return '{:.2e}'.format(val)
```

In [10]:
```
# The lower the loss function, the better the linear regression values (a, b) fit the dataset.
loss1 = loss_mse(a1, b1, inputs, outputs)
loss2 = loss_mse(a2, b2, inputs, outputs)
loss3 = loss_mse(a3, b3, inputs, outputs)
print(loss1, loss2, loss3)
```

```
1.01e+00 5.11e-02 7.85e-03
```



Training a model

Definition (Training a model):

Training a model consists of finding the best values for the trainable parameters of the chosen model, for the given task and dataset.

The goal of training is to find the set of model parameters that minimize the loss function. Once a minimal value is obtained on the loss function, then we can claim that the “optimal” trainable parameters have been found (or in other word, the model has been trained).

Training a linear regression model

In the case of linear regression, it consists of solving the following optimization problem.

$$(a^*, b^*) = \underset{a,b}{\operatorname{argmin}} \left[L(a, b, x, y) \right]$$

That is

$$(a^*, b^*) = \underset{a,b}{\operatorname{argmin}} \left[L(a, b, x, y) = \frac{1}{N} \sum_{i=1}^{N} (a x_i + b - y_i)^2 \right]$$

Training a linear regression model

Definition (The normal equation for linear regression):

Solving this optimization problem can be done analytically, as it can be proven that the optimal choice of parameters $W^* = \begin{pmatrix} b^* \\ a^* \end{pmatrix}$ follows the **normal equation**, below.

$$W^* = (X^T X)^{-1} X^T Y$$

With

$$X = \begin{pmatrix} 1 & x_1 \\ \dots & \dots \\ 1 & x_N \end{pmatrix}$$

And

$$Y = \begin{pmatrix} y_1 \\ \dots \\ y_N \end{pmatrix}$$

Finally, X^T denotes the transposed matrix of X and $(X)^{-1}$ its inverse.

Proof of the Normal Equation

Using the notations

$$X = \begin{pmatrix} 1 & x_1 \\ \dots & \dots \\ 1 & x_N \end{pmatrix},$$

$$Y = \begin{pmatrix} y_1 \\ \dots \\ y_N \end{pmatrix},$$

$$W^* = \begin{pmatrix} b^* \\ a^* \end{pmatrix}$$

We have, in matrix form

$$XW^* = \begin{pmatrix} a^*x_1 + b^* \\ \dots \\ a^*x_N + b^* \end{pmatrix}$$

And, also

$$XW^* - Y = \begin{pmatrix} a^*x_1 + b^* - y_1 \\ \dots \\ a^*x_N + b^* - y_N \end{pmatrix}$$

Proof of the Normal Equation

Our loss function, the MSE L, is therefore defined, in matrix format as:

$$L = \frac{1}{N}(XW^* - Y)^T(XW^* - Y)$$

This is equivalent to

$$L = \frac{1}{N}\left((XW^*)^T - Y^T\right)(XW^* - Y)$$

We then get

$$L = \frac{1}{N}\left(W^{*T}X^T - Y^T\right)(XW^* - Y)$$

Expanding

$$L = \frac{1}{N}(W^{*T}X^TXW^* - Y^TXW^* - W^{*T}X^TY + Y^TY)$$

Proof of the Normal Equation

The expression

$$L = \frac{1}{N}(W^{*T}X^TXW^* - Y^TXW^* - W^{*T}X^TY + Y^TY)$$

Recall that we are trying to find the solution to optimization problem

$$(a^*, b^*) = \underset{a,b}{\operatorname{argmin}} [L(a, b, x, y)]$$

Recognizing that $Y^TXW^* = W^{*T}X^TY$, differentiating our function L with respect to W and equating it to zero gives:

$$\frac{\partial L}{\partial W} = \frac{1}{N}(2X^TXW - 2X^TY) = 0$$

Proof of the Normal Equation

This is equivalent to

$$(X^T X W^* - X^T Y) = 0$$

Or,

$$X^T X W^* = X^T Y$$

Assuming $X^T X$ is invertible (and that might not always be the case based on your dataset!), this finally gives:

$$\boldsymbol{W^* = \left(X^T X\right)^{-1} X^T Y}$$

Training a linear regression model

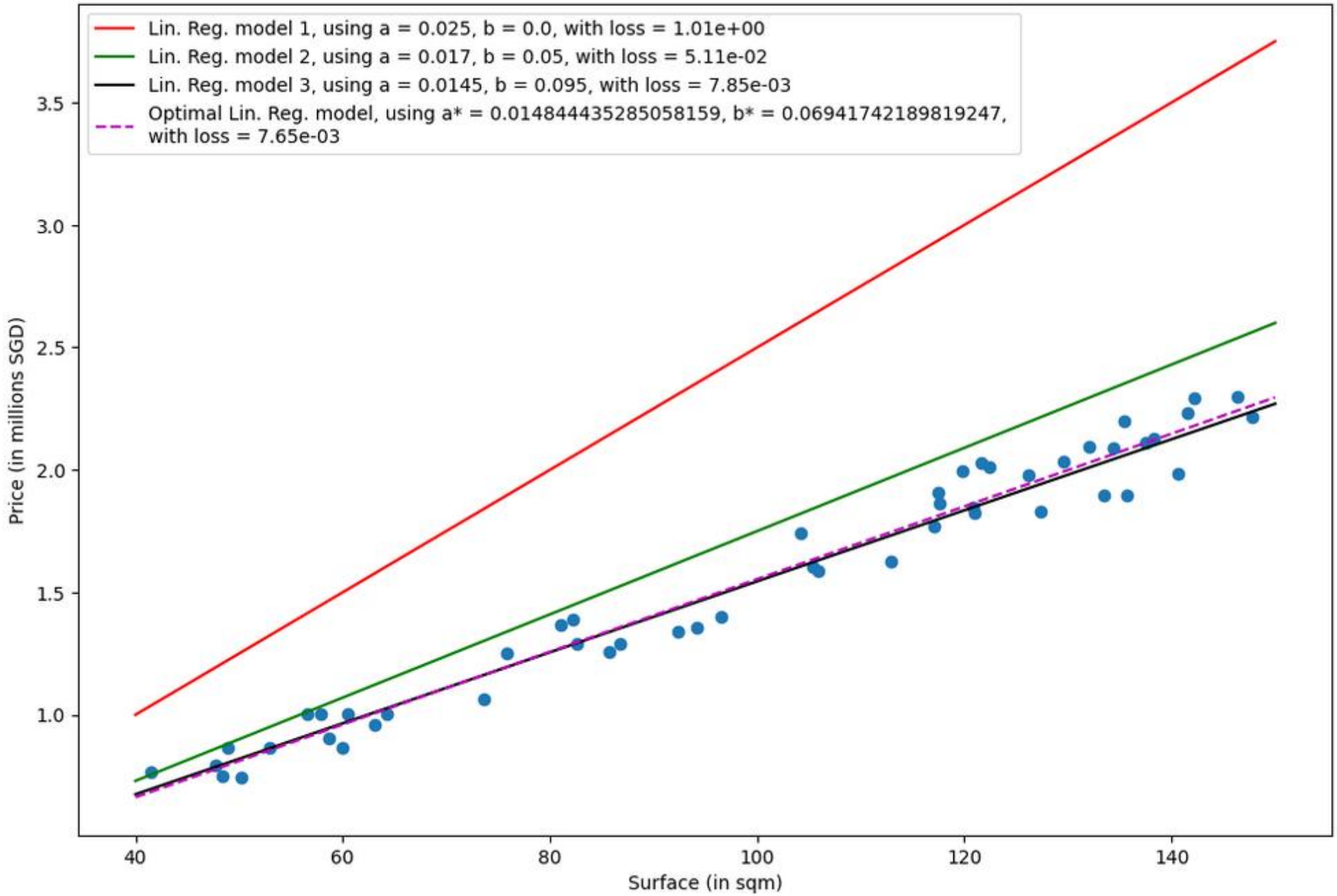
- Normal Equation formula: $W^* = (X^T X)^{-1} X^T Y$

```
# Defining the X matrix, following the notation above, as a numpy array.
X = np.array([[1, x_i] for x_i in inputs])
print(X)
```

```
# While we are at it, let us define the transposed version of X.
XT = np.transpose(X)
print(XT)
```

```
# Defining the Y matrix, following the notation above, as a numpy array.
Y = np.array([[y_i] for y_i in outputs])
print(Y)
```

```
# Defining W_star according to our formula.
W_star = np.matmul(np.linalg.inv(np.matmul(XT,X)), np.matmul(XT,Y))
print(W_star)
# Extracting a_star and b_star values from W_star.
b_star, a_star = W_star[0, 0], W_star[1, 0]
print("Optimal a_star value: ", a_star)
print("The value we used for a in the mock dataset generation: ", 14373/1000000)
print("Optimal b_star value: ", b_star)
print("The value we used for b in the mock dataset generation: ", 100000/1000000)
```



A problem with the normal equation

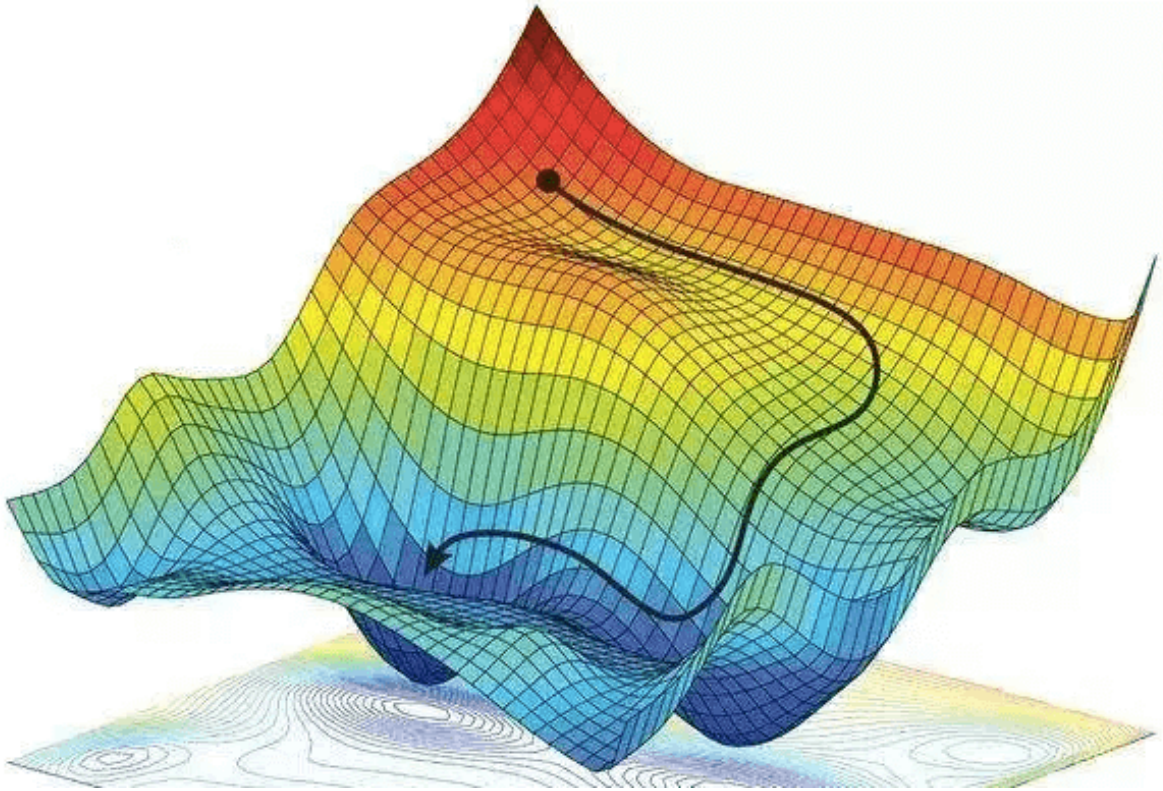
- The normal equation $W^* = (X^T X)^{-1} X^T Y$ immediately gives the optimal set of parameters (a^*, b^*) to use for linear regression.
- However, it can become **computationally expensive when the number of features is very large**.
- It might even be **impossible to use when the matrix of feature variables $(X^T X)^{-1}$ is not invertible**.
- **Problem: In these cases, the normal equation can be slow to compute, or it may not even have a solution.**
- **Another problem: More sophisticated models than linear regression will not have a normal equation, anyway.**

Gradient Descent, to the rescue!

Definition (Gradient Descent):

Gradient Descent (GD) is an iterative algorithm used to solve optimization problems.

- It starts with an initial set of parameters.
- It then repeatedly updates the parameters in the direction of the negative gradient of the given cost function, until it converges to a local minimum.



Forgot about GD?
Have a look at your notes from 10.013 M&A!

Gradient Descent, to the rescue!

Definition (Gradient Descent):

Gradient Descent (GD) is an iterative algorithm used to solve optimization problems.

- It starts with an initial set of parameters.
- It then repeatedly updates the parameters in the direction of the negative gradient of the given cost function, until it converges to a local minimum.

- The normal equation has many problems…
- But GD, on the other hand can be used to find the optimal solution even when the normal equation is not applicable.
- The main advantage of gradient descent is that it can handle very large datasets and it can also be used for non-linear models.

GD in Linear Regression

- In the case of Linear Regression, with the MSE loss defined earlier, we have

$$(a^*, b^*) = \underset{a,b}{\operatorname{argmin}} \left[L(a, b, x, y) = \frac{1}{N} \sum_{i=1}^{N} (ax_i + b - y_i)^2 \right]$$

GD in Linear Regression

We have the following derivatives with respect to a and b:

$$D_a = \frac{\partial L}{\partial a} = \frac{-2}{N} \sum_{i=1}^{N} x_i \left(y_i - (a\, x_i + b) \right)$$

$$D_b = \frac{\partial L}{\partial b} = \frac{-2}{N} \sum_{i}^{N} y_i - (a\, x_i + b)$$

GD in Linear Regression

The gradient descent **update rules** are therefore defined as

$$a \leftarrow a - \alpha D_a$$

$$a \leftarrow a + \frac{2\alpha}{N} \sum_{i=1}^{N} x_i \left(y_i - (a\, x_i + b) \right)$$

And

$$b \leftarrow b - \alpha D_b$$

$$b \leftarrow b + \frac{2\alpha}{N} \sum_{i}^{N} y_i - (a\, x_i + b)$$

With parameter α being the **learning rate** for the gradient descent, a parameter to be decided manually, later.

GD in Linear Regression

Gradient Descent Linear Regression procedure:

- We will then initialize a and b with some values (could be zero, random, or something else).
- For a given number of iterations, we will apply the two update rules defined earlier.
- (Optionally, we might decide to stop iterating, if we realize that the values of a and b are no longer changing. This is called **early stopping**, and is typically implemented by tracking the changes on each iteration and breaking if the changes are less than a threshold δ.)

```python
def gradient_descent_linreg(a_0, b_0, x, y, alpha = 1e-5, delta = 1e-5, max_count = 1000):
    # Define the initial values of a and b as a_0 and b_0
    a, b = a_0, b_0
    # Define N as the number of elements in the dataset
    N = len(x)
    # Keep track of how much a and b changed on each iteration
    change = float("Inf")
    # Counter as safety to prevent infinite looping
    counter = 0
    # List of losses, to be used for display later
    losses = []
    while change > delta:
        # Helper to visualize iterations of while loop
        print("-----")
        # Use gradident descent update rules for a and b
        D_a = -2/N*(sum([x_i*(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        D_b = 2/N*(sum([(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        a = a - alpha*D_a
        b = b - alpha*D_b
        print("Gradients: ", D_a, D_b)
        print("New values for (a, b): ", a, b)
        # Compute change
        change = max(abs(alpha*D_a), abs(alpha*D_b))
        print("Change: ", change)
        # COmpute and display current loss
        loss = loss_mse(a, b, x, y)
        losses.append(float(loss))
        print("Loss: ", loss)
        # Counter update, will break if iterations number exceeds max_count,
        # to prevent gradient descent from going indefinitely.
        # (Just a safety measure, for good practice, we would definitely prefer to see
        # the while loop break "naturally", because change eventually fell under the threshold stop.)
        counter += 1
        if(counter > max_count):
            print("Maximal number of iterations reached.")
            break
    return a, b, losses
```

Initialize a and b as you please.

```python
def gradient_descent_linreg(a_0, b_0, x, y, alpha = 1e-5, delta = 1e-5, max_count = 1000):
    # Define the initial values of a and b as a_0 and b_0
    a, b = a_0, b_0
    # Define N as the number of elements in the dataset
    N = len(x)
    # Keep track of how much a and b changed on each iteration
    change = float("Inf")
    # Counter as safety to prevent infinite looping
    counter = 0
    # List of losses, to be used for display later
    losses = []
    while change > delta:
        # Helper to visualize iterations of while loop
        print("-----")
        # Use gradident descent update rules for a and b
        D_a = -2/N*(sum([x_i*(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        D_b = 2/N*(sum([(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        a = a - alpha*D_a
        b = b - alpha*D_b
        print("Gradients: ", D_a, D_b)
        print("New values for (a, b): ", a, b)
        # Compute change
        change = max(abs(alpha*D_a), abs(alpha*D_b))
        print("Change: ", change)
        # COmpute and display current loss
        loss = loss_mse(a, b, x, y)
        losses.append(float(loss))
        print("Loss: ", loss)
        # Counter update, will break if iterations number exceeds max_count,
        # to prevent gradient descent from going indefinitely.
        # (Just a safety measure, for good practice, we would definitely prefer to see
        # the while loop break "naturally", because change eventually fell under the threshold stop.)
        counter += 1
        if(counter > max_count):
            print("Maximal number of iterations reached.")
            break
    return a, b, losses
```

Number of samples in dataset.

```python
def gradient_descent_linreg(a_0, b_0, x, y, alpha = 1e-5, delta = 1e-5, max_count = 1000):
    # Define the initial values of a and b as a_0 and b_0
    a, b = a_0, b_0
    # Define N as the number of elements in the dataset
    N = len(x)
    # Keep track of how much a and b changed on each iteration
    change = float("Inf")
    # Counter as safety to prevent infinite looping
    counter = 0
    # List of losses, to be used for display later
    losses = []
    while change > delta:
        # Helper to visualize iterations of while loop
        print("-----")
        # Use gradident descent update rules for a and b
        D_a = -2/N*(sum([x_i*(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        D_b = 2/N*(sum([(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        a = a - alpha*D_a
        b = b - alpha*D_b
        print("Gradients: ", D_a, D_b)
        print("New values for (a, b): ", a, b)
        # Compute change
        change = max(abs(alpha*D_a), abs(alpha*D_b))
        print("Change: ", change)
        # COmpute and display current loss
        loss = loss_mse(a, b, x, y)
        losses.append(float(loss))
        print("Loss: ", loss)
        # Counter update, will break if iterations number exceeds max_count,
        # to prevent gradient descent from going indefinitely.
        # (Just a safety measure, for good practice, we would definitely prefer to see
        # the while loop break "naturally", because change eventually fell under the threshold stop.)
        counter += 1
        if(counter > max_count):
            print("Maximal number of iterations reached.")
            break
    return a, b, losses
```

Two possible stopping conditions, change < delta or counter > max_count

```python
def gradient_descent_linreg(a_0, b_0, x, y, alpha = 1e-5, delta = 1e-5, max_count = 1000):
    # Define the initial values of a and b as a_0 and b_0
    a, b = a_0, b_0
    # Define N as the number of elements in the dataset
    N = len(x)
    # Keep track of how much a and b changed on each iteration
    change = float("Inf")
    # Counter as safety to prevent infinite looping
    counter = 0
    # List of losses, to be used for display later
    losses = []
    while change > delta:
        # Helper to visualize iterations of while loop
        print("-----")
        # Use gradident descent update rules for a and b
        D_a = -2/N*(sum([x_i*(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        D_b = 2/N*(sum([(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        a = a - alpha*D_a
        b = b - alpha*D_b
        print("Gradients: ", D_a, D_b)
        print("New values for (a, b): ", a, b)
        # Compute change
        change = max(abs(alpha*D_a), abs(alpha*D_b))
        print("Change: ", change)
        # COmpute and display current loss
        loss = loss_mse(a, b, x, y)
        losses.append(float(loss))
        print("Loss: ", loss)
        # Counter update, will break if iterations number exceeds max_count,
        # to prevent gradient descent from going indefinitely.
        # (Just a safety measure, for good practice, we would definitely prefer to see
        # the while loop break "naturally", because change eventually fell under the threshold stop.)
        counter += 1
        if(counter > max_count):
            print("Maximal number of iterations reached.")
            break
    return a, b, losses
```

Update using our GD update rules from earlier

```python
def gradient_descent_linreg(a_0, b_0, x, y, alpha = 1e-5, delta = 1e-5, max_count = 1000):
    # Define the initial values of a and b as a_0 and b_0
    a, b = a_0, b_0
    # Define N as the number of elements in the dataset
    N = len(x)
    # Keep track of how much a and b changed on each iteration
    change = float("Inf")
    # Counter as safety to prevent infinite looping
    counter = 0
    # List of losses, to be used for display later
    losses = []
    while change > delta:
        # Helper to visualize iterations of while loop
        print("-----")
        # Use gradident descent update rules for a and b
        D_a = -2/N*(sum([x_i*(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        D_b = 2/N*(sum([(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        a = a - alpha*D_a
        b = b - alpha*D_b
        print("Gradients: ", D_a, D_b)
        print("New values for (a, b): ", a, b)
        # Compute change
        change = max(abs(alpha*D_a), abs(alpha*D_b))
        print("Change: ", change)
        # COmpute and display current loss
        loss = loss_mse(a, b, x, y)
        losses.append(float(loss))
        print("Loss: ", loss)
        # Counter update, will break if iterations number exceeds max_count,
        # to prevent gradient descent from going indefinitely.
        # (Just a safety measure, for good practice, we would definitely prefer to see
        # the while loop break "naturally", because change eventually fell under the threshold stop.)
        counter += 1
        if(counter > max_count):
            print("Maximal number of iterations reached.")
            break
    return a, b, losses
```

Compute change on this iteration (to decide on early stopping or not)

```python
def gradient_descent_linreg(a_0, b_0, x, y, alpha = 1e-5, delta = 1e-5, max_count = 1000):
    # Define the initial values of a and b as a_0 and b_0
    a, b = a_0, b_0
    # Define N as the number of elements in the dataset
    N = len(x)
    # Keep track of how much a and b changed on each iteration
    change = float("Inf")
    # Counter as safety to prevent infinite looping
    counter = 0
    # List of losses, to be used for display later
    losses = []
    while change > delta:
        # Helper to visualize iterations of while loop
        print("-----")
        # Use gradident descent update rules for a and b
        D_a = -2/N*(sum([x_i*(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        D_b = 2/N*(sum([(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        a = a - alpha*D_a
        b = b - alpha*D_b
        print("Gradients: ", D_a, D_b)
        print("New values for (a, b): ", a, b)
        # Compute change
        change = max(abs(alpha*D_a), abs(alpha*D_b))
        print("Change: ", change)
        # COmpute and display current loss
        loss = loss_mse(a, b, x, y)
        losses.append(float(loss))
        print("Loss: ", loss)
        # Counter update, will break if iterations number exceeds max_count,
        # to prevent gradient descent from going indefinitely.
        # (Just a safety measure, for good practice, we would definitely prefer to see
        # the while loop break "naturally", because change eventually fell under the threshold stop.)
        counter += 1
        if(counter > max_count):
            print("Maximal number of iterations reached.")
            break
    return a, b, losses
```

Calculate new MSE value using the new parameters a and b. We also keep track of these losses for display later

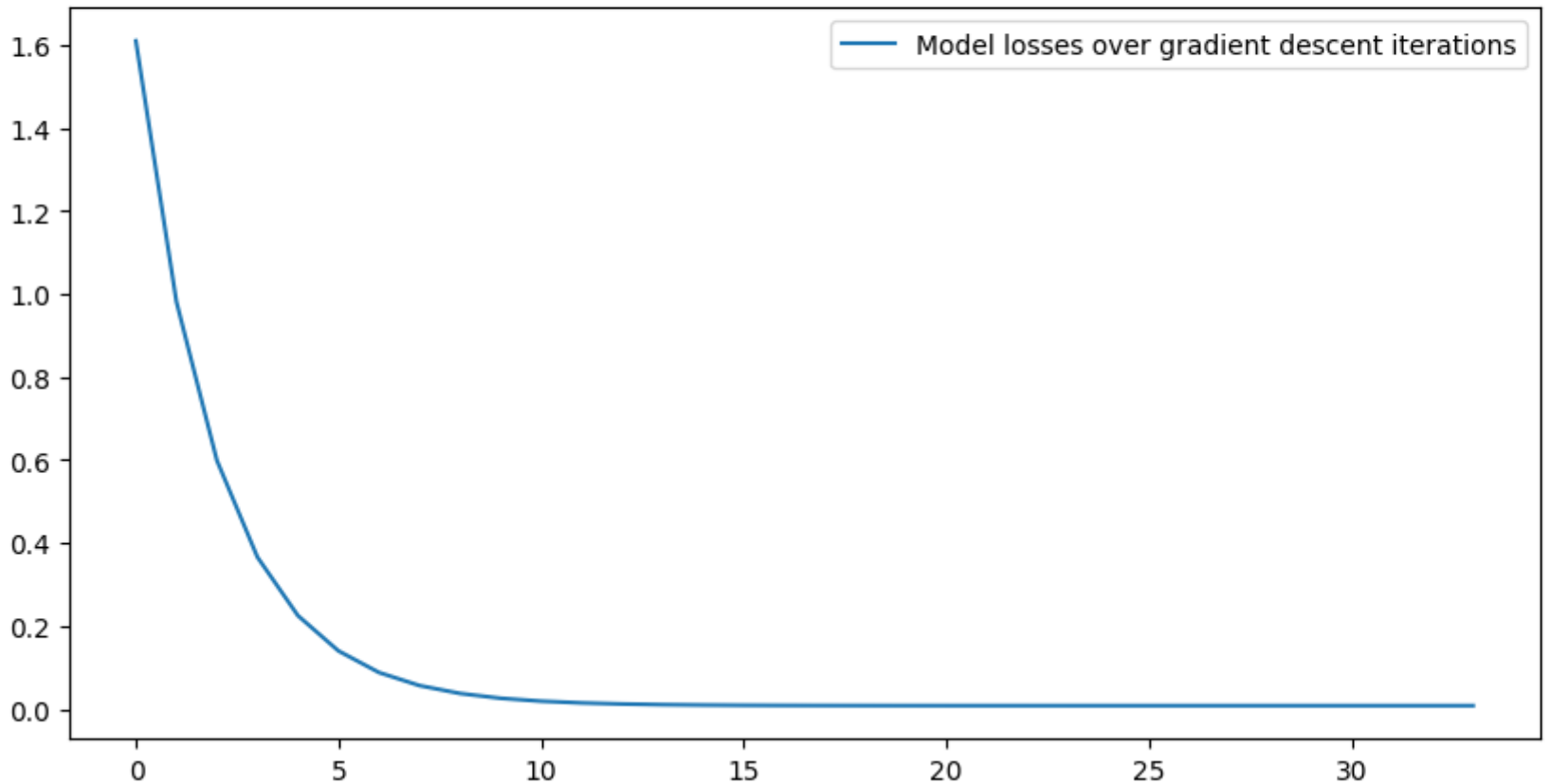
```python
def gradient_descent_linreg(a_0, b_0, x, y, alpha = 1e-5, delta = 1e-5, max_count = 1000):
    # Define the initial values of a and b as a_0 and b_0
    a, b = a_0, b_0
    # Define N as the number of elements in the dataset
    N = len(x)
    # Keep track of how much a and b changed on each iteration
    change = float("Inf")
    # Counter as safety to prevent infinite looping
    counter = 0
    # List of losses, to be used for display later
    losses = []
    while change > delta:
        # Helper to visualize iterations of while loop
        print("-----")
        # Use gradident descent update rules for a and b
        D_a = -2/N*(sum([x_i*(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        D_b = 2/N*(sum([(y_i - (a*x_i + b)) for x_i, y_i in zip(x, y)]))
        a = a - alpha*D_a
        b = b - alpha*D_b
        print("Gradients: ", D_a, D_b)
        print("New values for (a, b): ", a, b)
        # Compute change
        change = max(abs(alpha*D_a), abs(alpha*D_b))
        print("Change: ", change)
        # COmpute and display current loss
        loss = loss_mse(a, b, x, y)
        losses.append(float(loss))
        print("Loss: ", loss)
        # Counter update, will break if iterations number exceeds max_count,
        # to prevent gradient descent from going indefinitely.
        # (Just a safety measure, for good practice, we would definitely prefer to see
        # the while loop break "naturally", because change eventually fell under the threshold stop.)
        counter += 1
        if(counter > max_count):
            print("Maximal number of iterations reached.")
            break
    return a, b, losses
```

Return trained parameters and losses evolution on each iteration

```
: 1 a_gd, b_gd, losses = gradient_descent_linreg(a_0 = 0, b_0 = 0, x = inputs, y = outputs, alpha = 1e-5, delta = 1e-6)
```

```
-----
Gradients:  -342.3996226384001 3.105429920000001
New values for (a, b):  0.0034239962263840013 -3.105429920000001e-05
Change:  0.0034239962263840013
Loss:  1.61e+00
-----
Gradients:  -266.62829357559235 2.4212214483389536
New values for (a, b):  0.006090279162139925 -5.526651368338955e-05
Change:  0.0026662829357559236
Loss:  9.82e-01
-----
Gradients:  -207.62478780001163 1.8884249597020164
New values for (a, b):  0.008166527040140042 -7.415076328040972e-05
Change:  0.0020762478780001164
Loss:  5.99e-01
-----
Gradients:  -161.6784683033283 1.4735337252735503
New values for (a, b):  0.009783311723173324 -8.888610053314522e-05
Change:  0.001616784683033283
```

```python
# Display dataset
plt.figure(figsize = (10, 5))
plt.plot(losses, label = "Model losses over gradient descent iterations")

# Display
plt.legend(loc = 'best')
plt.show()
```

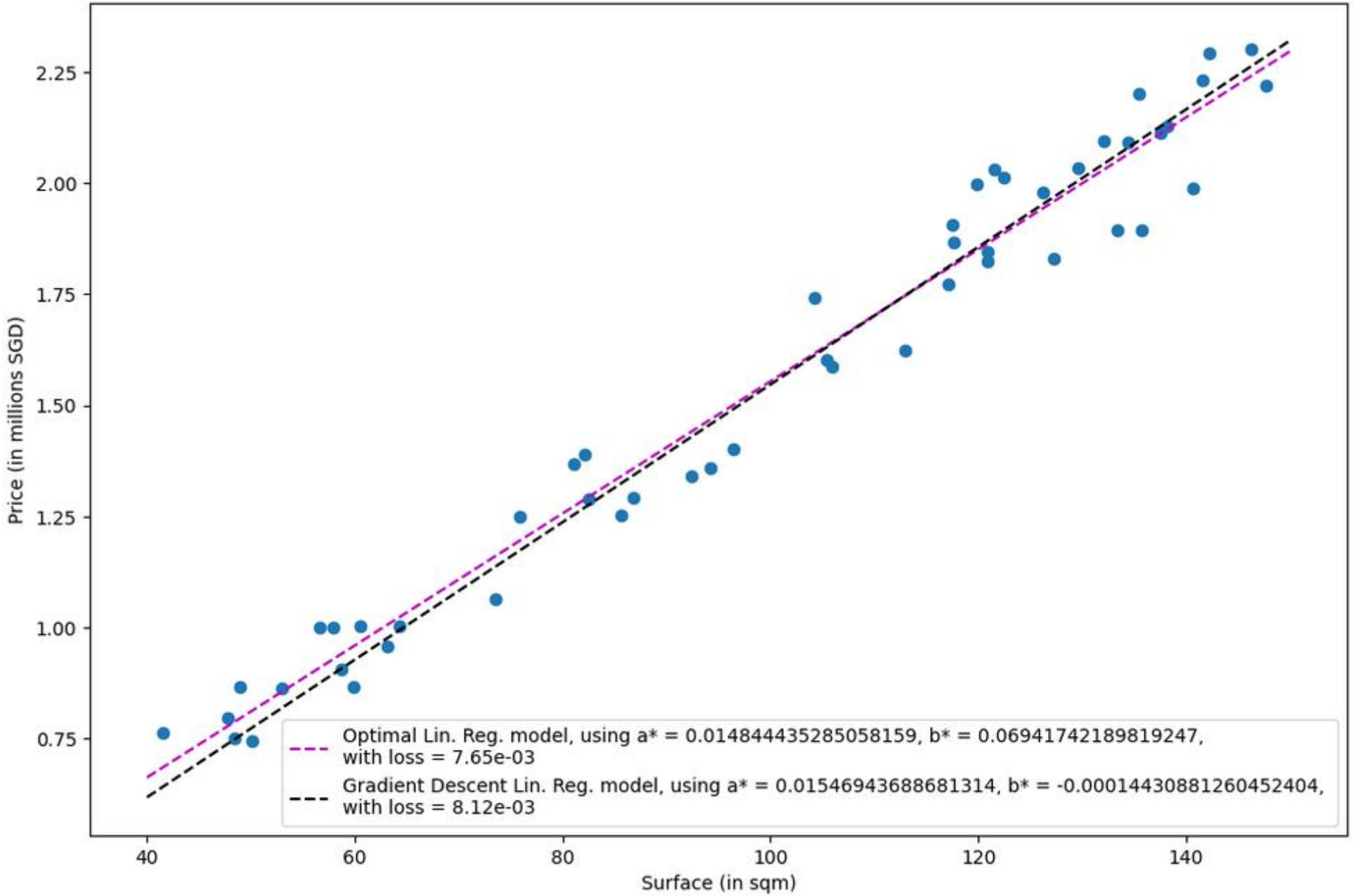


Checking for optimal parameters

- We have generated the dataset ourselves, so we know what should be the values for a and b!

```
print("Optimal a_star value: ", a_star)
print("Value for a_star, found by gradient descent: ", a_gd)
print("We used 14373/1000000 in the mock dataset generation, which is: ", 14373/1000000)
print("Optimal b_star value: ", b_star)
print("Value for b_star, found by gradient descent: ", b_gd)
print("The value we used in the mock dataset generation: ", 100000/1000000)
```

```
Optimal a_star value:  0.014844435285058159
Value for a_star, found by gradient descent:  0.015469436886813l4
We used 14373/1000000 in the mock dataset generation, which is:  0.014373
Optimal b_star value:  0.069417421898l9247
Value for b_star, found by gradient descent:  -0.0001443088126045240l
The value we used in the mock dataset generation:  0.1
```



Using Sklearn for Linear regression

- In practice, we never implement the linear regression model ourselves (but it is a good practice to try it at least once!)
- It is often faster to rely on the **sklearn** library and use the **LinearRegression** object!

```
# Creating a sklearn Linear Regression model.
# It uses the same analytical formula from earlier, i.e. W^* = (X^T X)^{-1} X^T Y.
reg = LinearRegression().fit(sk_inputs, sk_outputs)
# The coefficients for a* and b* are found using coeff_ and intercept_ respectively.
a_sk = reg.coef_[0]
b_sk = reg.intercept_
print("Optimal a_star value: ", a_star)
print("Value for a_star, found by sklearn: ", a_sk)
print("Optimal b_star value: ", b_star)
print("Value for b_star, found by sklearn: ", b_sk)
```

```
Optimal a_star value:  0.014844435285058159
Value for a_star, found by sklearn:  0.014844435285058166
Optimal b_star value:  0.06941742189819247
Value for b_star, found by sklearn:  0.06941742189818956
```

Predicting using Sklearn Linear regression

- After training, it is also good practice to check if the predictor makes sense, by asking it to predict the price of an apartment it has never seen before.
- Confirm the value manually, if possible.

```
# We can later use this Linear Regression model, to predict the price of
# a new appartment with surface 105 sqm (price in millions SGD).
new_surf = 105
pred_price = reg.predict(np.array([[new_surf]]))[0]
print(pred_price)
```

```
1.628083126829297
```

```
avg_price = 14373*new_surf + 100000
min_val = 0.9*avg_price
max_val = 1.1*avg_price
print("Min, max, avg prices: ", [min_val, max_val, avg_price])
```

```
Min, max, avg prices:  [1448248.5, 1770081.5000000002, 1609165]
```

Extension of multi-parameter Linear Regression

Let us now consider that the inputs consist of more than one parameter, e.g. each sample x_i consists of:

- The surface x_i^1,
- The distance to closest MRT x_i^2,
- Etc.

The linear regression can be simply transposed.

S$ 1,680,000 Negotiable

3 3 1184 sqft S$ 1,418.92 psf

Details

Property Type Executive Condominium For Sale	Floor Size 1184 sqft
Developer Tampines EC Pte Ltd	PSF S$ 1,418.92 psf
Furnishing Partially Furnished	Floor Level High Floor
Tenure 99-year Leasehold	TOP January, 2016

Extension of multi-parameter Linear Regression

Definition (Multi-parameter Linear Regression):

Multi-parameter Linear Regression is a model, which **assumes that there is a linear relationship between inputs and outputs.**

It therefore consists of several trainable parameters $(a_1, a_2, \dots a_K, b)$, to be freely chosen.

These will connect any input $x_i = \left(x_i^1, x_i^2, \dots, x_i^K\right)$ to its respective output y_i, with the following equation:

$$y_i \approx \sum_{k=1}^{K} a_k \, x_i^k \; + \; b$$

Extension of multi-parameter Linear Regression

The MSE loss then becomes:

$$L = \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{k=1}^{K} a_k x_i^k + b - y_i\right)^2$$

And the optimization problem related to training is then:

$$(a_1^*, \ldots, a_K^*, b^*) = \arg\min_{a_1,\ldots,a_K,b} L$$

Extension of multi-parameter Linear Regression

We could then try to compute the new normal equation for this problem (good practice for your optimization skills, try it out!)

Or, and it is often preferable, we could then define gradient descent update rules with respect to every trainable parameter $(\boldsymbol{a_1}, \boldsymbol{a_2}, \ldots \boldsymbol{a_K}, \boldsymbol{b})$ like before.

However, we leave this implementation for students who would like to practice. (or it might be the homework for this week!)

From Linear to Polynomial Regression

Definition (Polynomial Regression with degree K):

Polynomial Regression is a model, which **assumes that there is a polynomial relationship between inputs and outputs,** which can be defined as a **polynomial function of degree K.**

It therefore consists of several trainable parameters $(a_1, a_2, \dots a_K, b)$, to be freely chosen.

These will connect any input $x_i = \left(x_i^1, x_i^2, \dots, x_i^K\right)$ to its respective output y_i, with the following equation:

$$y_i \approx \sum_{k=1}^{K} a_k \, (x_i)^k \; + \; b$$

The hyperplane concept

Definition (Hyperplanes in Linear Algebra):

A **hyperplane** is a subspace of one dimension less than the ambient space.

In the case of linear regression with one input feature and one output feature, the ambient space is of dimension 2 (that is 1+1).

The Linear Regression model equation, is defined as

$$y = ax + b$$

This is the equation of a line, which is a 1D subspace, therefore an hyperplane of the 2D ambient space.

The same happens with the polynomial regression, but at a higher degree.

From Linear to Polynomial Regression

We will follow the same steps as before with linear regression, starting with a mock dataset.

- This will be different compared than what has been implemented in notebooks 1 and 2, as we will generate prices y_i as a polynomial function of the surfaces x_i.
- We will assume that the function $y = f(x)$, giving the price of an apartment with surface x, is defined as a polynomial function with degree 3.

$$y = f(x) = 100000 + 14373x + 3x^3$$

- In addition, we will add a random noise to the final pricing, with a +/- 10% drift as before.

From Linear to Polynomial Regression

We will follow the same steps as before with linear regression, starting with a mock dataset.

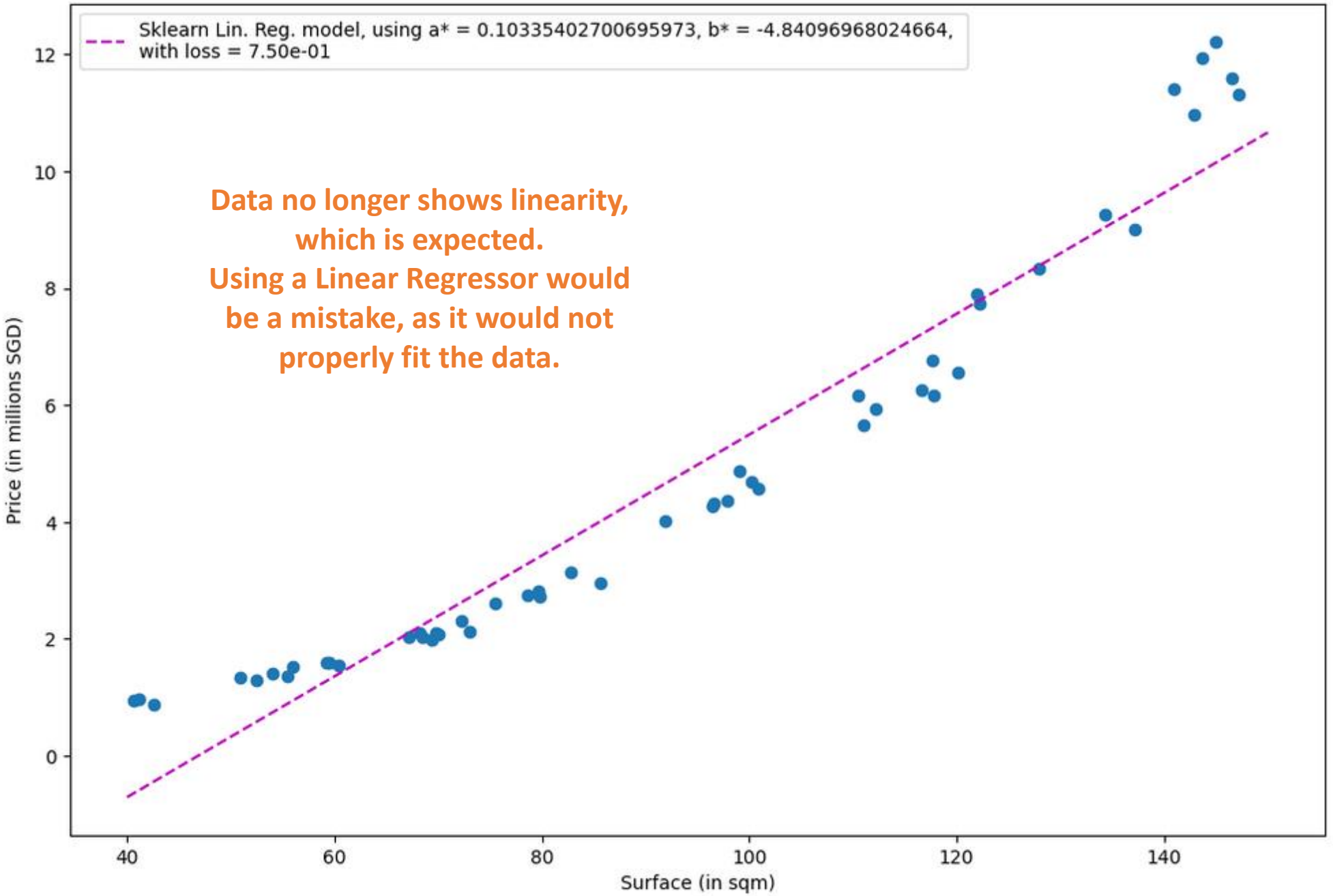
- This will be different compared than what has been implemented in notebooks 1 and 2, as we will generate prices y_i as a polynomial function of the surfaces x_i.
- We will assume that the function $y = f(x)$, giving the price of an apartment with surface x, is defined as a polynomial function with degree 3.

$$y = f(x) = 100000 + 14373x + 3x^3$$

- In addition, we will add a random noise to the final pricing, with a +/- 10% drift as before.

From Linear to Polynomial Regression

```python
# All helper functions
def surface(min_surf, max_surf):
    return round(np.random.uniform(min_surf, max_surf), 2)
def price(surface):
    # Note: this has changed and is now a polynomial function.
    return round((100000 + 14373*surface + 3*surface**3)*(1 + np.random.uniform(-0.1, 0.1)))/1000000
def generate_datasets(n_points, min_surf, max_surf):
    x = np.array([surface(min_surf, max_surf) for _ in range(n_points)])
    y = np.array([price(i) for i in x])
    return x, y
def linreg_matplotlib(a, b, min_surf, max_surf, n_points = 50):
    x = np.linspace(min_surf, max_surf, n_points)
    y = a*x + b
    return x, y
def loss_mse(a, b, x, y):
    val =  np.sum((y - (a*x + b))**2)/x.shape[0]
    return '{:.2e}'.format(val)
```



Polynomial Regression with sklearn

Sklearn treats polynomial regression as a **multi-parameter linear regression**, with **polynomial features**.

Polynomial features: reworking the inputs so that

$$\tilde{X} = \begin{pmatrix} x_1 & \cdots & (x_1)^K \\ \vdots & \ddots & \vdots \\ x_N & \cdots & (x_N)^K \end{pmatrix}$$

And $\tilde{x}_{i,j} = (x_i)^j$

```
1 # Preparing polynomial features for our dataset
2 n_degree = 3
3 sk_poly = PolynomialFeatures(degree = n_degree, include_bias = False)
4 sk_poly_inputs = sk_poly.fit_transform(sk_inputs.reshape(-1, 1))
5 print(sk_poly_inputs)
```

```
[[5.24800000e+01 2.75415040e+03 1.44537813e+05]
 [1.47190000e+02 2.16648961e+04 3.18885606e+06]
 [1.20160000e+02 1.44384256e+04 1.73492122e+06]
 [7.86600000e+01 6.18739560e+03 4.86700538e+05]
 [1.17840000e+02 1.38862656e+04 1.63635754e+06]
 [1.27960000e+02 1.63737616e+04 2.09518653e+06]
 [1.11010000e+02 1.23232201e+04 1.36800066e+06]
 [8.56100000e+01 7.32907210e+03 6.27441862e+05]
 [1.17660000e+02 1.38438756e+04 1.62887040e+06]
 [6.71300000e+01 4.50643690e+03 3.02517109e+05]
 [6.81600000e+01 4.64578560e+03 3.16656746e+05]
 [4.26400000e+01 1.81816960e+03 7.75267517e+04]
 [5.08600000e+01 2.58673960e+03 1.31561576e+05]
 [7.30500000e+01 5.33630250e+03 3.89816898e+05]
 [1.10490000e+02 1.22080401e+04 1.34886635e+06]
 [7.54400000e+01 5.69119360e+03 4.29343645e+05]
 [6.04000000e+01 3.64816000e+03 2.20348864e+05]
```

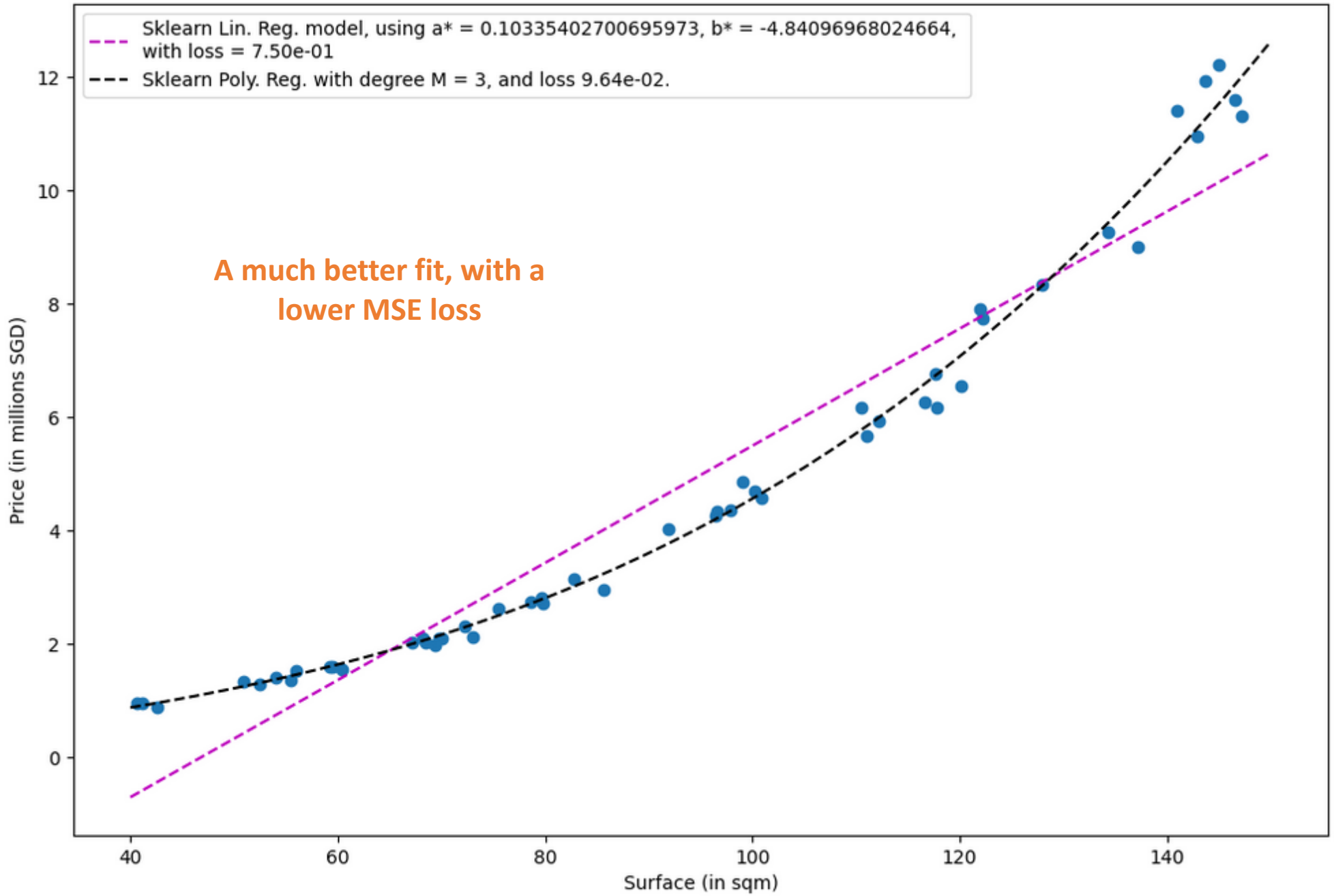
Polynomial Regression with sklearn

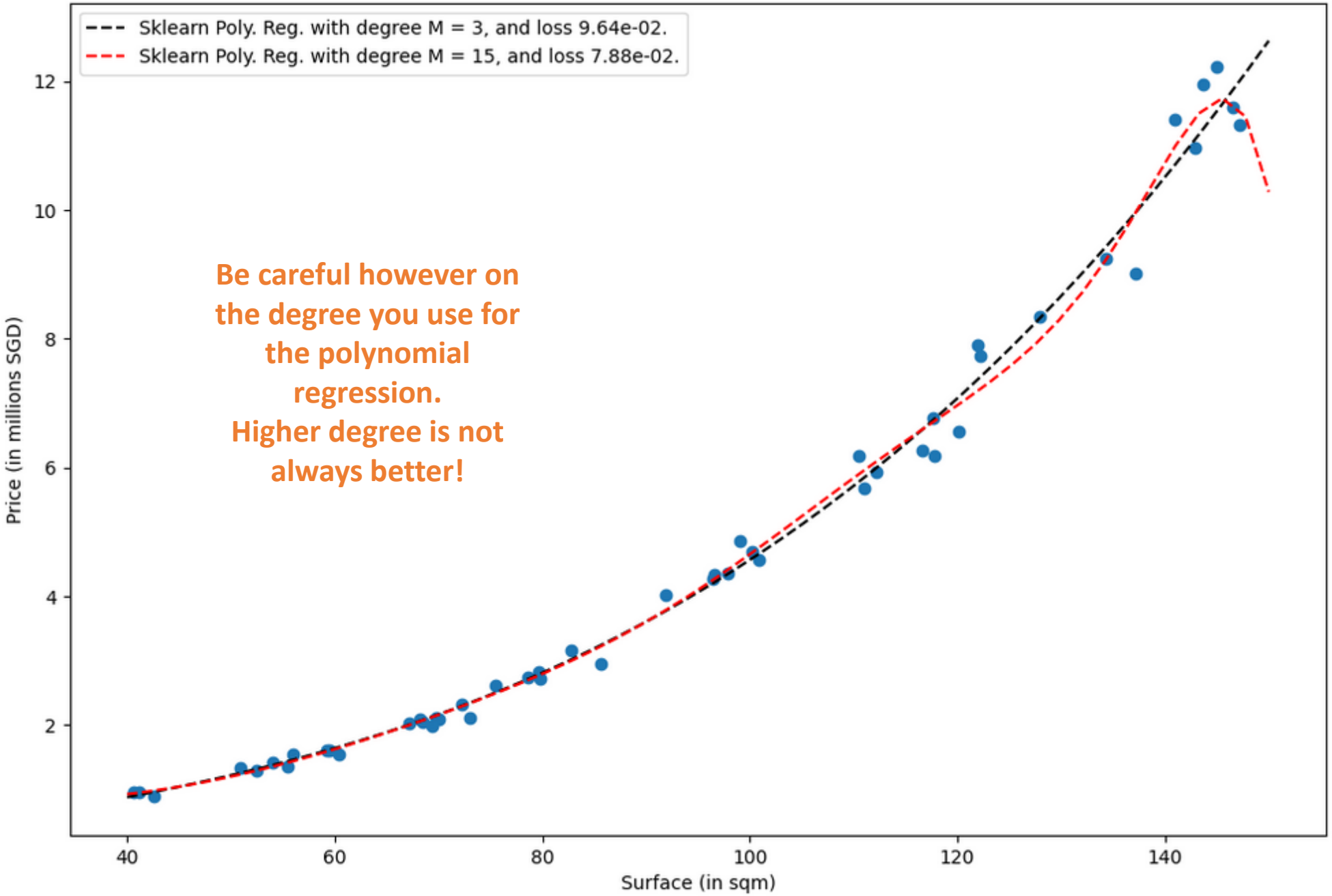
We then proceed normally, assuming $\tilde{X}$ are our new inputs.

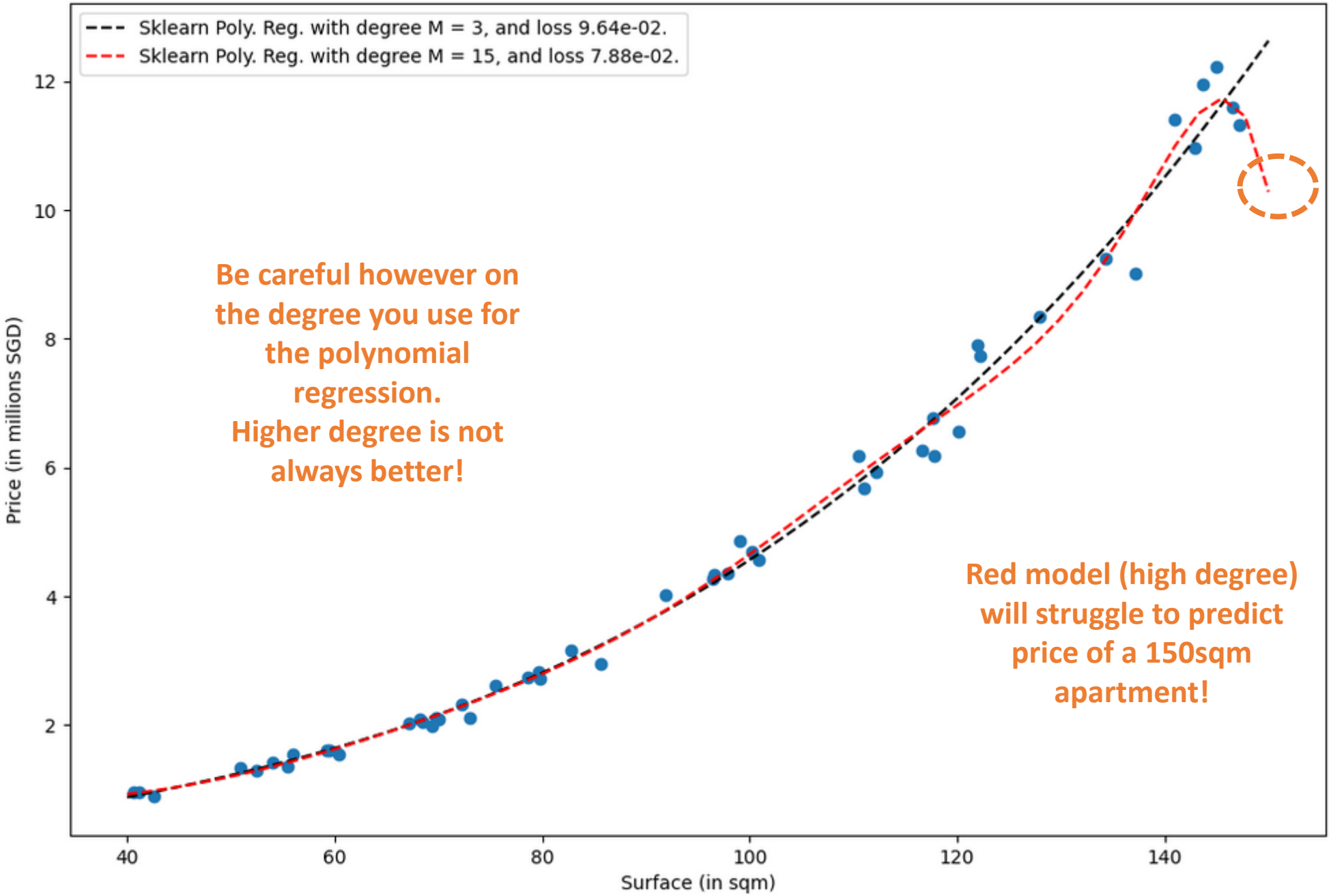
Sklearn then automatically adjusts and produces the right number of parameters for the model.

```
1  # Training a Polynomial Regressor
2  poly_reg_model = LinearRegression()
3  poly_reg_model.fit(sk_poly_inputs, sk_outputs)
4  a_sk_poly = poly_reg_model.coef_
5  b_sk_poly = poly_reg_model.intercept_
6  print(a_sk_poly, b_sk_poly)
```

```
[ 2.38010878e-02 -1.30213791e-04  3.58102988e-06] -0.09785310239196843
```



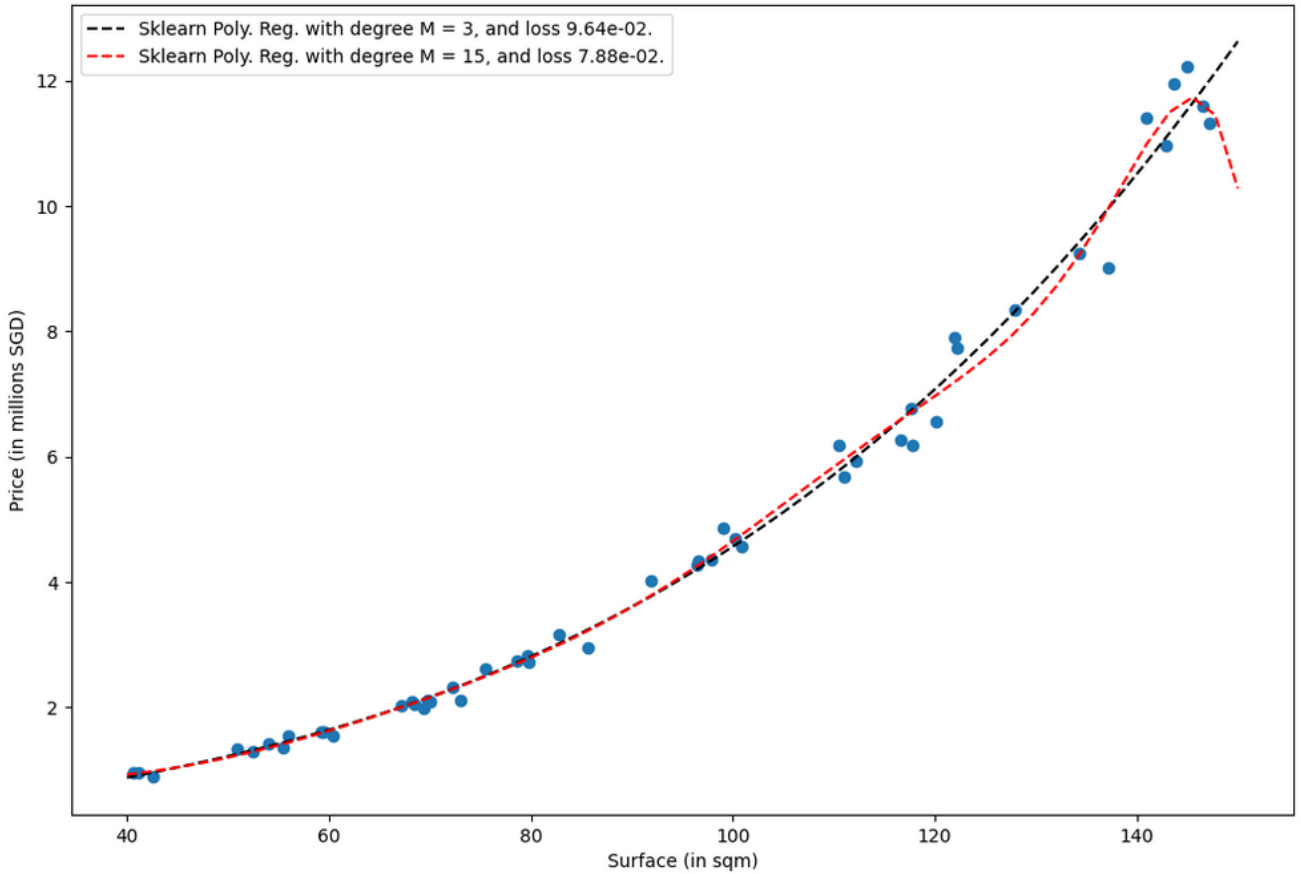




Overfitting vs. degree of polynomial

- Red curve shows what happens when using a degree K that is too high compared to the relationship connecting inputs and outputs (it was 3).

- This phenomenon is called **overfitting**.

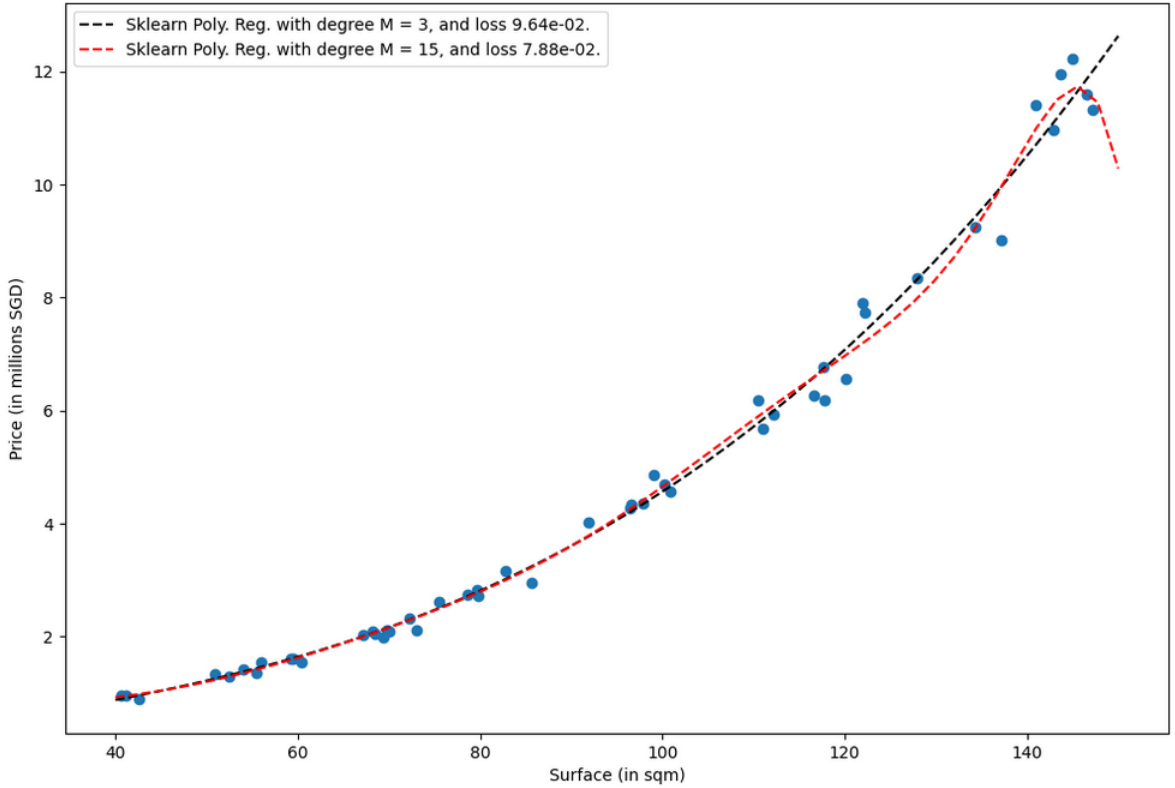


Overfitting vs. degree of polynomial

Definition (Overfitting):

Overfitting is a phenomenon that occurs in machine learning when a model is trained too well on the training data.

As a result, it performs poorly on new, unseen data (here an apartment with 150sqm surface, which we had not in the training dataset).



Overfitting vs. degree of polynomial

Definition (Overfitting):

Overfitting is a phenomenon that occurs in machine learning when a model is trained too well on the training data.

As a result, it performs poorly on new, unseen data (here an apartment with 150sqm surface, which we had not in the training dataset).

This typically happens when

- a model is trained with too many features
- or too much data,

It becomes too complex for the task at hand, resulting in poor **generalization** to new data.

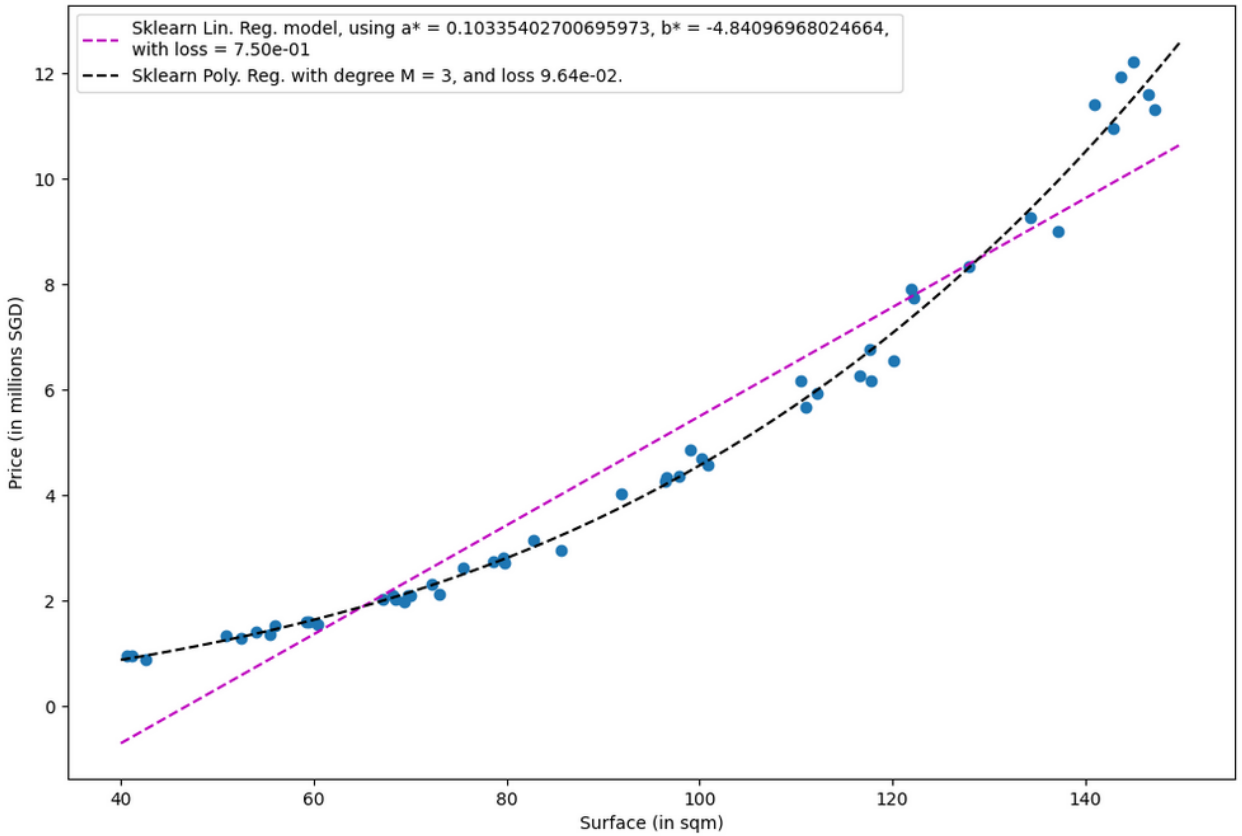
In other words, it memorizes the noise in the training data rather than learning the correct underlying pattern.

Underfitting vs. degree of polynomial

Definition (Underfitting):

Underfitting is a phenomenon that occurs in machine learning when a model is not capable enough to capture the underlying pattern of the data.

For instance, here, Linear Regression (which is Polynomial Regression with $K = 1$) could not fit the data well.



Underfitting vs. degree of polynomial

Definition (Underfitting):

Underfitting is a phenomenon that occurs in machine learning when a model is not capable enough to capture the underlying pattern of the data.

For instance, here, Linear Regression (which is Polynomial Regression with $K = 1$) could not fit the data well.

This can happen when a model

- is trained with too few features
- or too little data,
- or when the model is not powerful enough to capture the complexity of the task.

Generalization

Definition (Generalization):

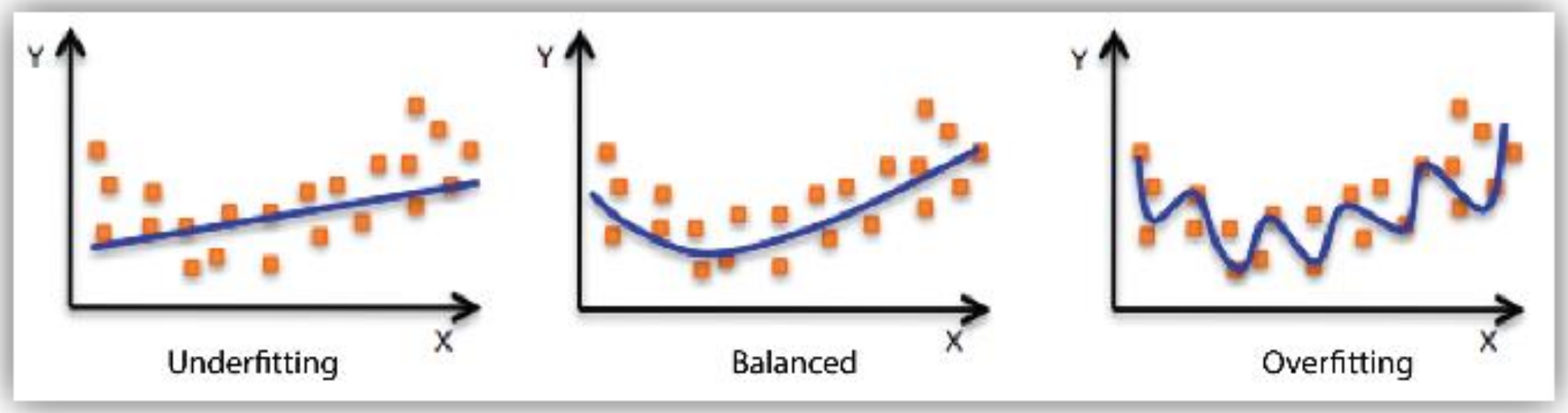
Generalization is considered the "holy grail" of machine learning because it is the ultimate goal of any machine learning model.

Even though the model was trained on seen data we collected, the purpose of a machine learning model is to make accurate predictions on new, unseen data.

If a model can make accurate predictions on data it has not seen before, it is said to have **generalized well**, and can then be used in the real world for solving the problem it was designed to solve.

Generalization vs. Overfitting/Underfitting

- In general, having a model that is **underfitting** or **overfitting** will lead to poor **generalization**.
- We would very much prefer to fit the data “just right”.



While we are at it…

In general, we want to:

- Train the model was trained on data we collected,
- Confirm that the model can generalize and make accurate predictions on new, unseen data, after training.

Problem: We would prefer to test the model before releasing it in the wild (what if it is wrong?)

Problem #2: Testing its generalization capabilities on the same data that was used for training would NOT confirm generalization.

Solution: Use some of our seen data to train the model, and some of the data (that has not been used for training and is therefore unseen to the model yet) for evaluating its generalization.

Train and test split

Definition (Train and Test Split):

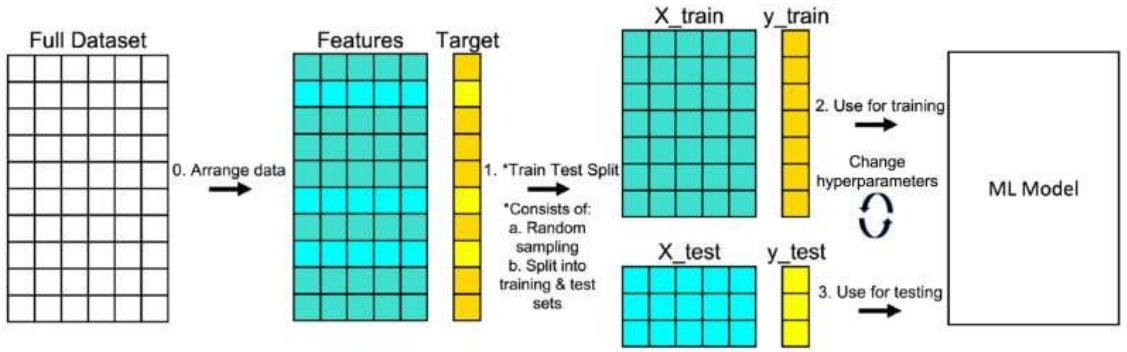
Train and test split is a method used in machine learning to divide a dataset into two subsets:

- The **training set** is used to train the model
- The **test set** is used to evaluate the performance of the trained model.

The idea is to use a portion of the dataset for training and a separate portion for testing so that the model can be evaluated on unseen data.

The typical split is to use 80% of the data for training and 20% for testing.

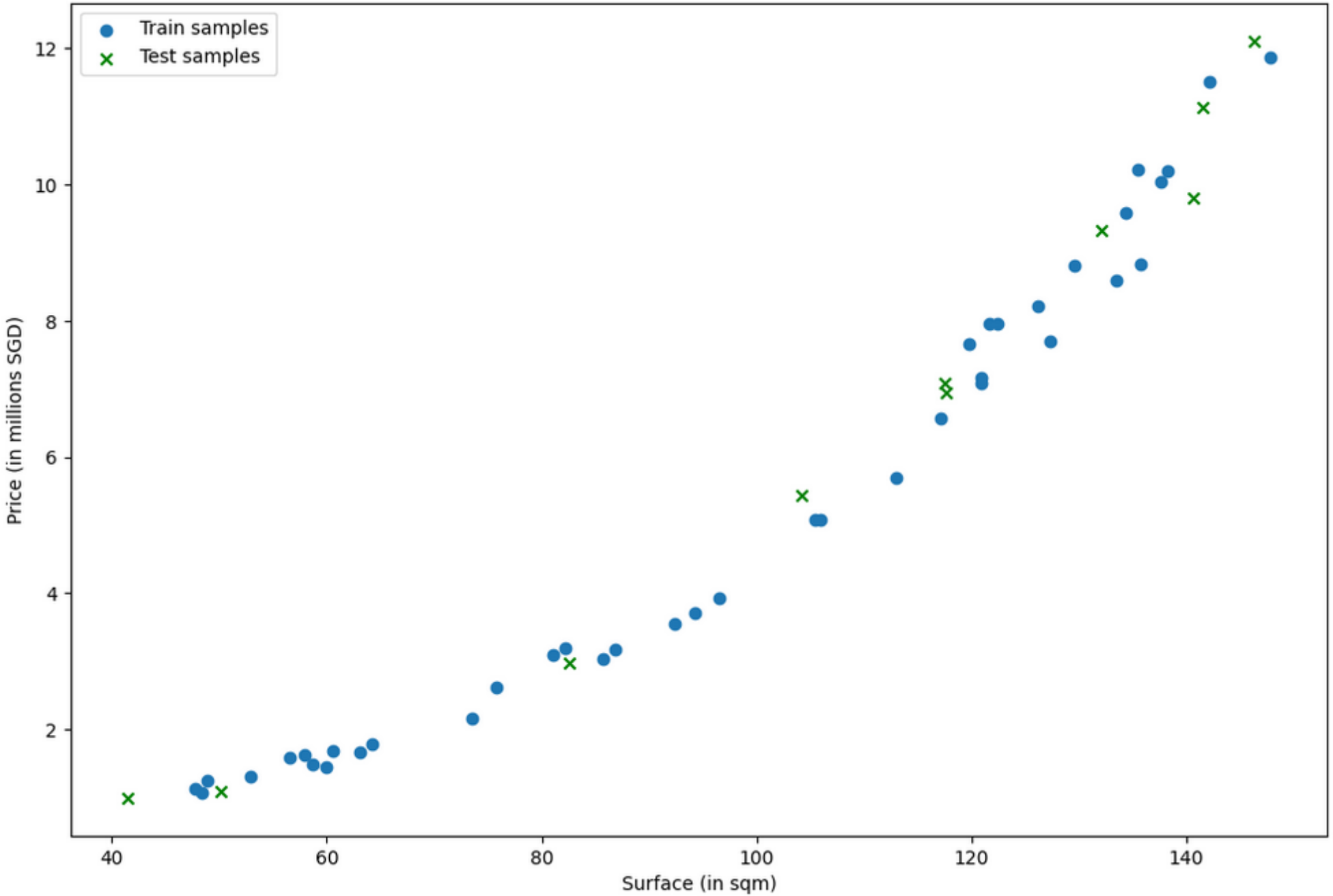
This can however vary depending on the specific use case and the size of the dataset.



Train and test split

```
# 80% of the samples will be used for training,
# and the remaining 20% will be used to evaluate generalization/overfitting.
ratio_train = 0.8
split_index = int(n_points*ratio_train)
# Training inputs and outputs
train_inputs = inputs[:split_index]
train_outputs = outputs[:split_index]
# Testing inputs and outputs
test_inputs = inputs[split_index:]
test_outputs = outputs[split_index:]
# Display
print(train_inputs)
print(train_outputs)
print(test_inputs)
print(test_outputs)
```

```
[ 86.83 129.6  120.89 135.48  82.17 147.74 138.25  63.07 121.6  112.95
 137.55 134.38 122.42 135.72  60.54  75.81  81.02 127.31  56.62  58.69
  48.93  73.57 126.16  57.92  47.77 117.12  59.91 105.88  85.68  96.49
  64.27 119.81 133.44 142.18 120.95  92.42  94.22 105.4   48.36  52.92]
[ 3.171633  8.805667  7.166722 10.224944  3.195151 11.87205  10.206911
  1.676214  7.954078  5.696169 10.049157  9.588631  7.968783  8.827882
  1.693317  2.6239    3.094831  7.700582  1.597057  1.489182  1.247223
  2.166055  8.212138  1.627403  1.12722   6.560241  1.450404  5.072051
  3.031968  3.938955  1.782147  7.652644  8.58658  11.518957  7.088832
  3.563134  3.702923  5.087156  1.071143  1.308982]
[146.31 104.17  50.17  41.5  132.06 140.63 117.5   82.57 117.63 141.56]
[12.111945  5.438864  1.088438  0.998857  9.333922  9.801835  7.095279
  2.982004  6.953392 11.129658]
```

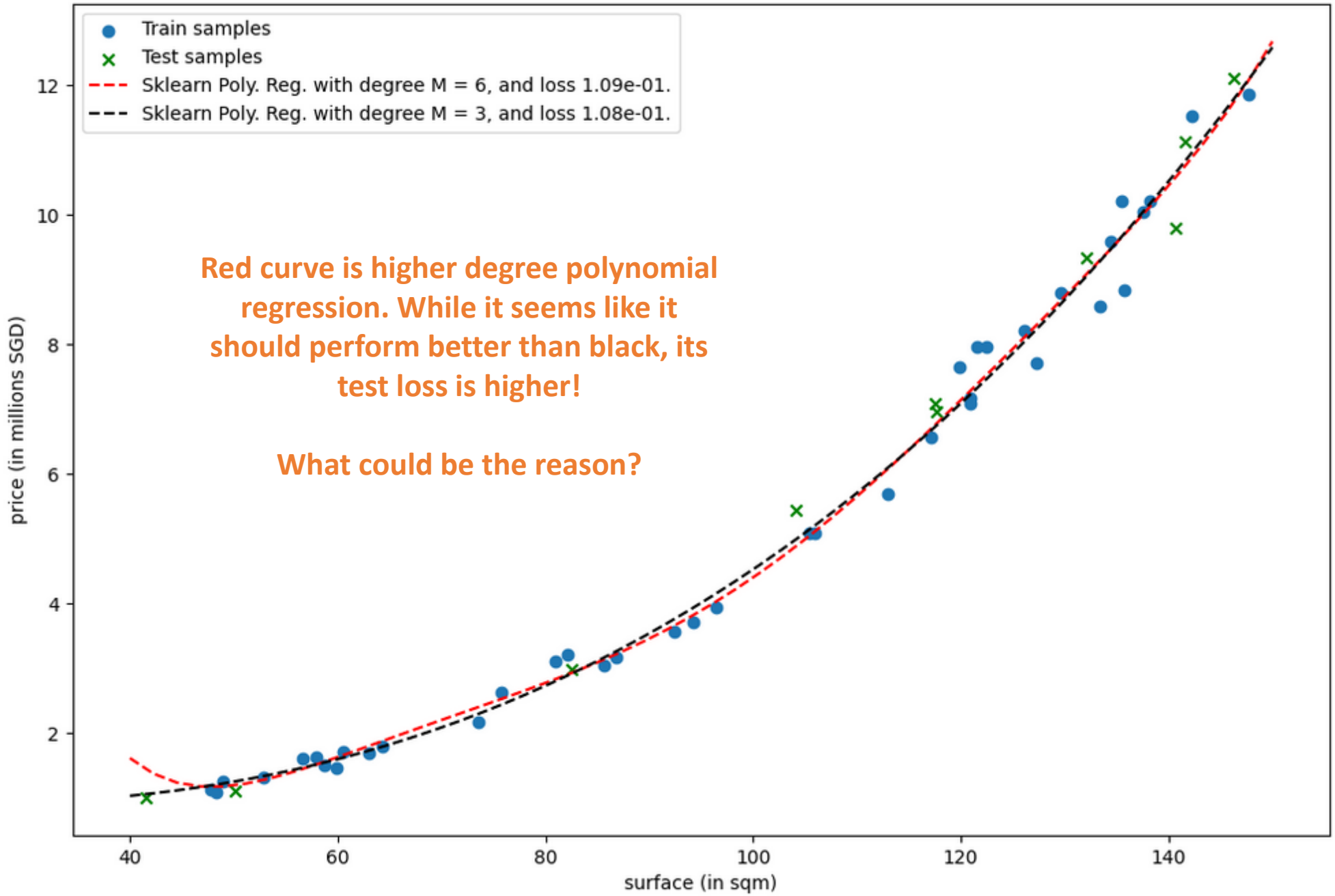


Training the Polynomial Regressor, again

- This time, we only train the polynomial regressor on the training data.
- After training, evaluate loss using the test data.

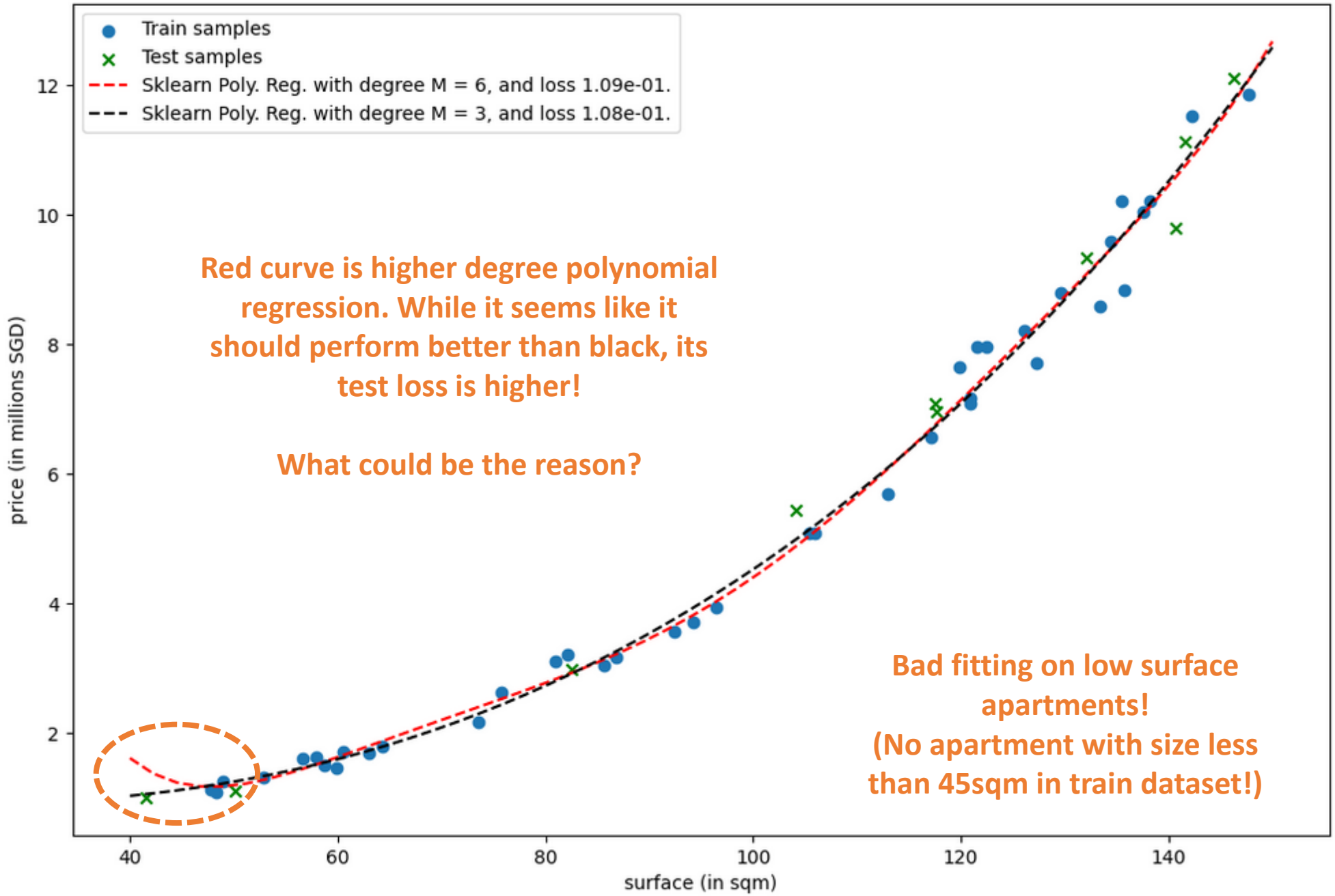
```
# Training a Polynomial Regressor
n_degree = 6
sk_inputs = np.array(train_inputs).reshape(-1, 1)
sk_outputs = np.array(train_outputs)
sk_poly = PolynomialFeatures(degree = n_degree, include_bias = False)
sk_poly_inputs = sk_poly.fit_transform(sk_inputs.reshape(-1, 1))
poly_reg_model = LinearRegression()
poly_reg_model.fit(sk_poly_inputs, sk_outputs)
a_sk = poly_reg_model.coef_
b_sk = poly_reg_model.intercept_
print(a_sk, b_sk)
```

```
[-5.31103644e+00  1.55941662e-01 -2.34706297e-03  1.92370550e-05
 -8.12548986e-08  1.38734052e-10] 73.25495081585393
```



Red curve is higher degree polynomial regression. While it seems like it should perform better than black, its test loss is higher!

What could be the reason?



Generalization vs. Train/Test Distributions

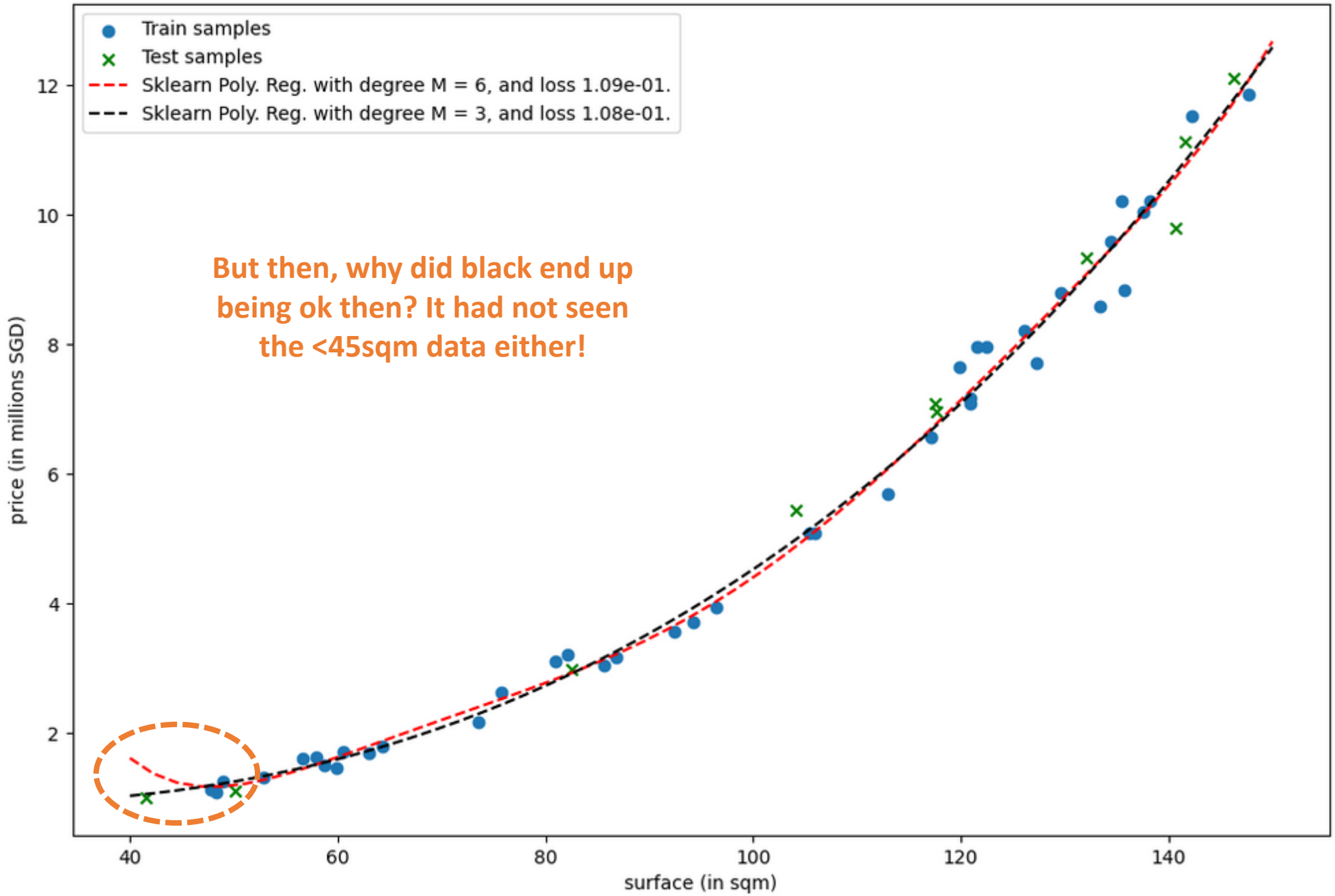
Definition (The need for train and test distribution similarity):

In order to generalize well in is important that the **train and test set are following similar distributions**.

$$P_{train} \approx P_{test}$$

Here, we ended up having a problem, as our training dataset did not contain any apartments with size lower than 45sqm.

I mean, you would not train a model on Singaporean apartments and use it to predict the price of apartments in Kuala Lumpur, right?



Regularization

Definition (Regularization):

Regularization consists of helping the model avoid overfitting (which is usually more prominent than underfitting).

It can typically be done by making sure the model is not too complex for the data, but many other approaches exist.

In general,

- the more complex the data is (in terms of features), the more complex the model can be.
- the more data you have (in terms of quantity of individual samples in dataset), the more complex the model can be.

Regularization

Definition (Regularization):

Regularization consists of helping the model avoid overfitting (which is usually more prominent than underfitting).

It can typically be done by making sure the model is not too complex for the data, but many other approaches exist.

A more common approach consists of **adding a penalty term to the loss function**.

The penalty term, also known as a **regularization term**, discourages the model from assigning too much weight to any one feature, i.e. making a certain parameter a_k too prominent (leading to overfitting).

Regularization

Definition (Ridge Regression):

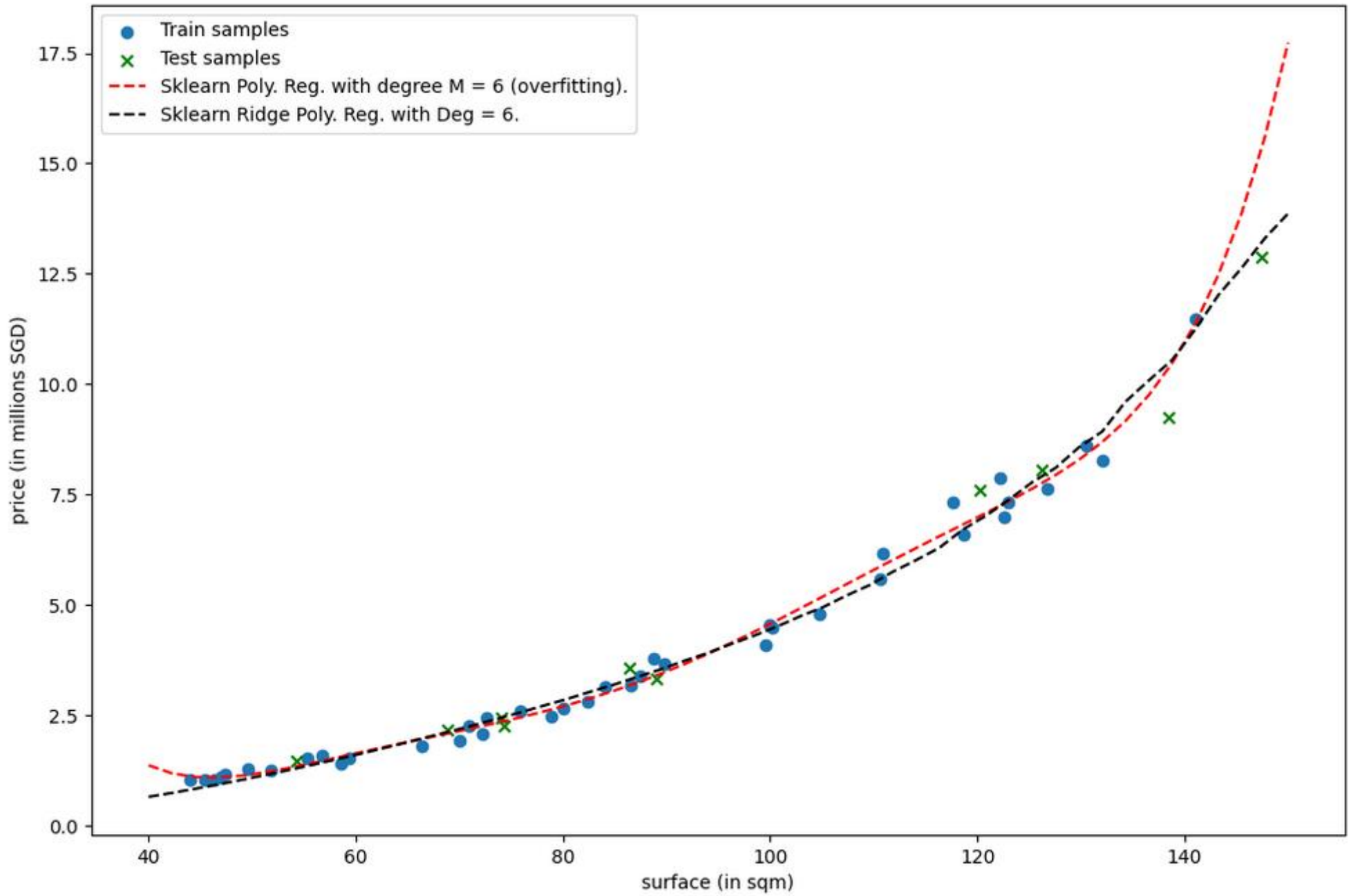
The **Ridge Polynomial Regressor is a polynomial regressor, like described earlier.**

Its only difference is **the loss function includes an additional regularization term $\lambda\, R(a, b)$.**

This regularization term consists of the **sum of the squared values for the trainable parameters**, i.e.:

$$\lambda\, R(a,b) = \frac{\lambda}{2N}\left(\sum_{k}^{M} (a_k)^2 \; + \; (b)^2\right)$$

- This regularization term encourages the model to assign low values to the coefficients of the model.
- This will, in turn, lead to less overfitting. Indeed, overfitting often occurs when high values are assigned to coefficients for high degrees (i.e. coefficients a_k with for high k values).



Regularization

Definition (Ridge Regression):

The **Ridge Polynomial Regressor is a polynomial regressor, like described earlier.**

Its only difference is **the loss function includes an additional regularization term $\lambda\, R(a, b)$.**

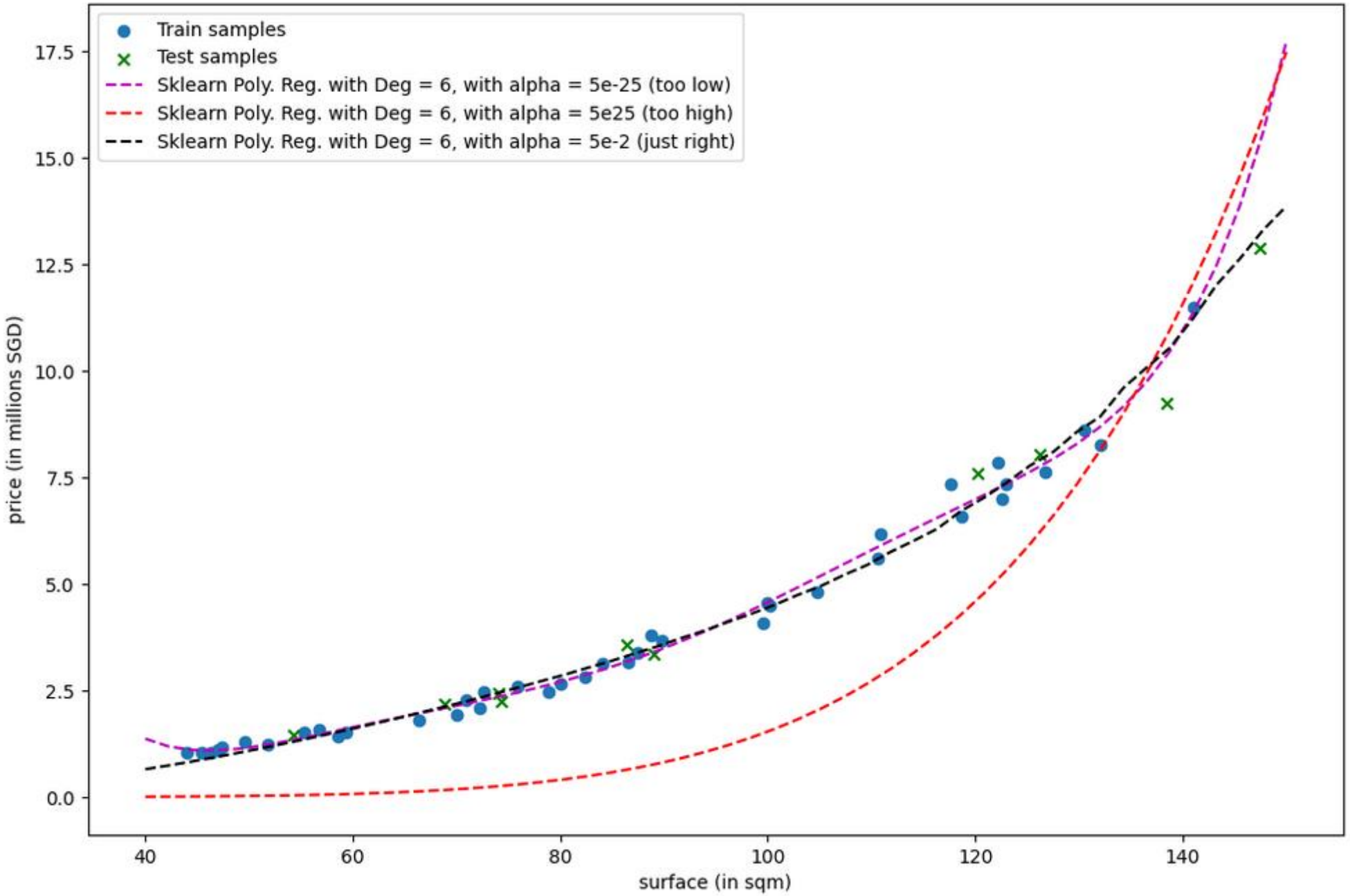
This regularization term consists of the **sum of the squared values for the trainable parameters**, i.e.:

$$\lambda\, R(a,b) = \frac{\lambda}{2N}\left(\sum_{k}^{M} (a_k)^2 \; + \; (b)^2\right)$$

Note: Just like the learning rate in GD, the parameter λ is manually decided and is used to weight the importance of the regularization term compared to the MSE.

If the **value for λ is small**, the loss function will be **mostly MSE** and our model will therefore suffer from the same **overfitting** problems as earlier.

If the **value of λ is too high** however, the model will **mostly ignore the MSE** part of the loss function, which will lead to a model not fitting (or **underfitting**) the data.



Regularization

Definition (Ridge Regression):

The **Ridge Polynomial Regressor is a polynomial regressor, like described earlier.**

Its only difference is **the loss function includes an additional regularization term $\lambda\, R(a, b)$.**

This regularization term consists of the **sum of the squared values for the trainable parameters**, i.e.:

$$\lambda\, R(a,b) = \frac{\lambda}{2N}\left(\sum_{k}^{M} (a_k)^2 \; + \; (b)^2\right)$$

Note #2: it is also possible to use the sum of **absolute** values, making a **Lasso Regressor** instead.

Why use L1 regularization over L2 regularization?

- L1 regularization results in some weights being exactly equal to zero, which can be used for feature selection.
- L2 regularization will not produce sparse models, and it will only shrink the weights.

Regularization

Definition (Ridge Regression):

The **Ridge Polynomial Regressor is a polynomial regressor, like described earlier.**

Its only difference is **the loss function includes an additional regularization term $\lambda\, R(a, b)$.**

This regularization term consists of the **sum of the squared values for the trainable parameters**, i.e.:

$$\lambda\, R(a,b) = \frac{\lambda}{2N}\left(\sum_{k}^{M} (a_k)^2 \; + \; (b)^2\right)$$

Note #3: it is also possible to use the sum of **absolute** values **AND squared** values as regularization

Essentially getting the best of both worlds, and making an **Elastic Regressor** instead!

Check it out!

Introducing the sigmoid function

Definition (the sigmoid function):

The **sigmoid function** is an important function used in Machine Learning. It is defined as

$$s(x) = \frac{1}{1 + \exp(-x)}$$

Or, equivalently

$$s(x) = \frac{\exp(x)}{1 + \exp(x)}$$

Note: Sometimes, the notation $\sigma(x)$ is used instead of $s(x)$.

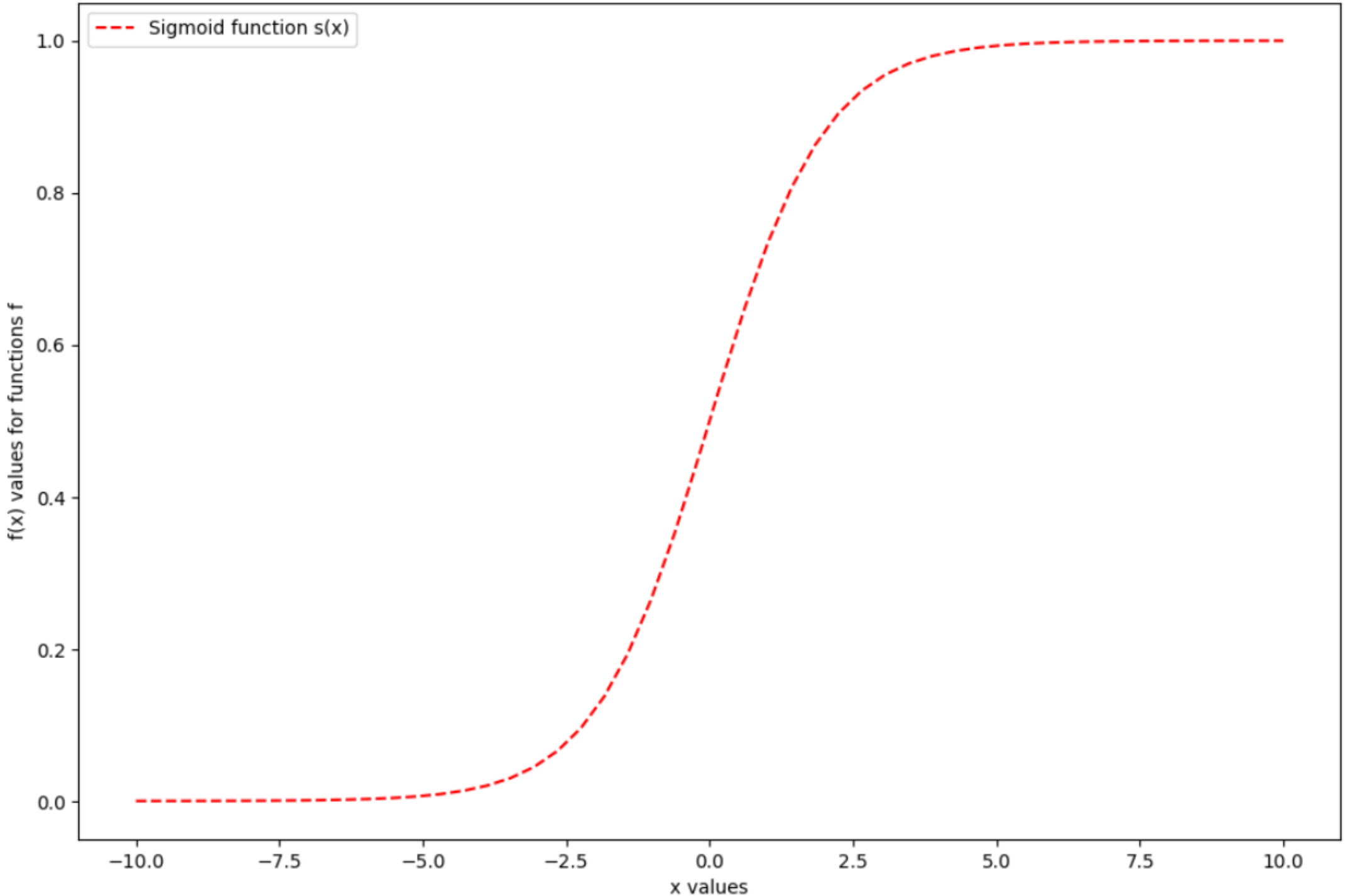
The sigmoid function has the following properties:

$$\forall\, x, \qquad 0 < s(x) < 1$$

$$\lim_{x \to -\infty} s(x) = 0$$

$$\lim_{x \to \infty} s(x) = 1$$

$$s(0) = 0.5$$


Sigmoid function s(x)
1.0
0.8
0.6
0.4
0.2
0.0
f(x) values for functions f
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x values

Introducing two logistic functions

Definition (the logistic functions):

Similarly, let us introduce two **logistic functions**, below:

$$l(x) = \ln(x)$$

$$l_2(x) = \ln(1 - x)$$

These two functions also have interesting properties.

$$\lim_{x \to 0} l(x) = -\infty$$

$$\lim_{x \to 1} l(x) = 0$$

$$\lim_{x \to 0} l_2(x) = 0$$

$$\lim_{x \to 1} l_2(x) = -\infty$$

$$l(0.5) = l_2(0.5) = -\ln(2)$$

To recap

Definition (the sigmoid function):

$$s(x) = \frac{1}{1 + \exp(-x)}$$

Or, equivalently

$$s(x) = \frac{\exp(x)}{1 + \exp(x)}$$

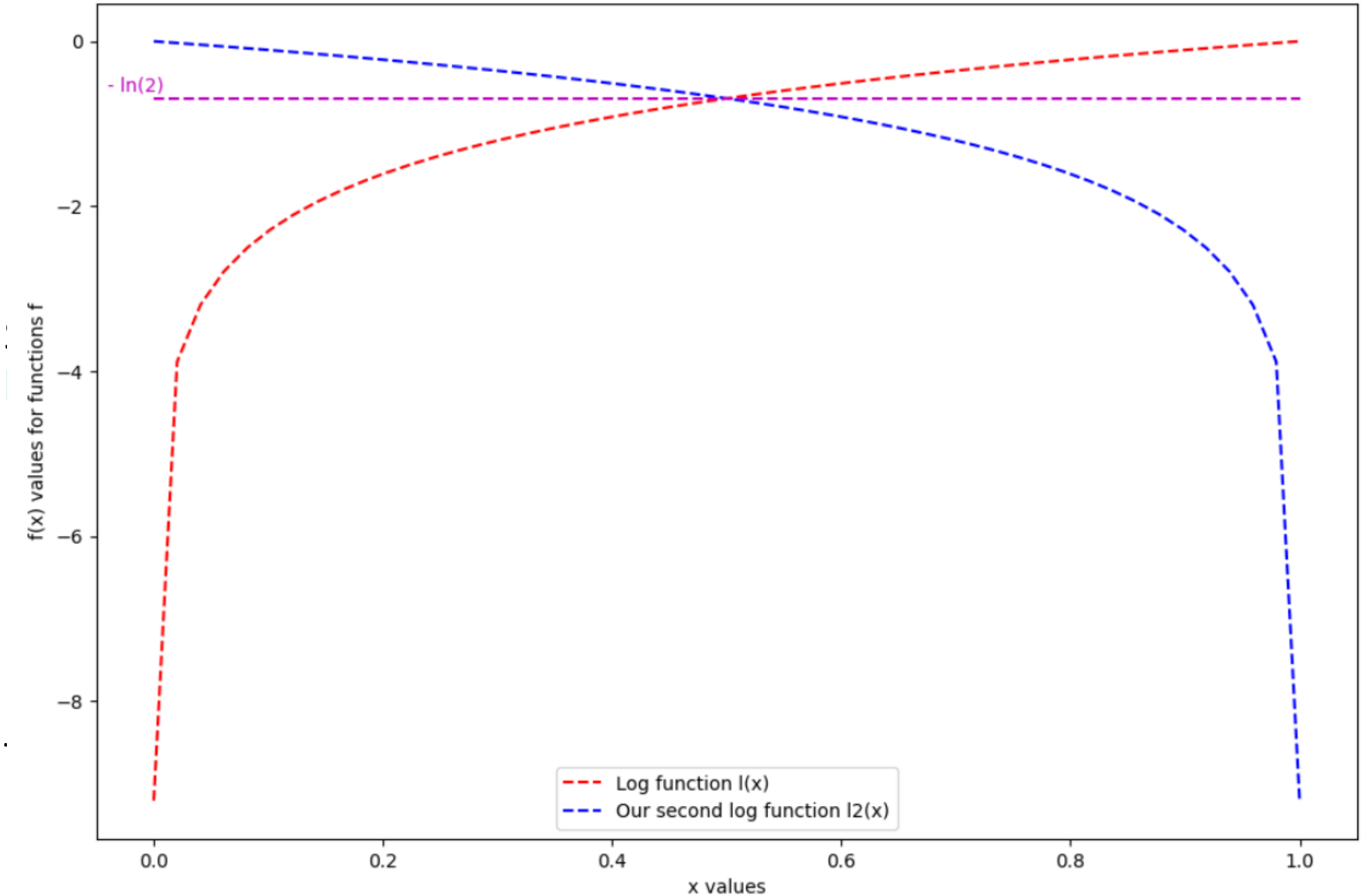
Definition (the logistic functions):

$$l(x) = \ln(x)$$

And,

$$l_2(x) = \ln(1 - x)$$

```
1  def s(x):
2      return 1/(1 + np.exp(-x))
3  def l(x):
4      return np.log(x)
5  def l2(x):
6      return np.log(1 - x)
```


0
- ln(2)
−2
−4
−6
−8
f(x) values for functions f
0.0
0.2
0.4
0.6
0.8
1.0
x values
Log function l(x)
Our second log function l2(x)

Introducing two logistic functions

In addition, the following properties hold (some of them require using L'Hospital rule to prove them, try it out?).

$$\lim_{x \to 0} x \, ln(x) = 0$$

$$\lim_{x \to 1} x \, ln(x) = 0$$

$$\lim_{x \to 0} (1 - x) \, ln(1 - x) = 0$$

$$\lim_{x \to 1} (1 - x) \, ln(1 - x) = 0$$

Introducing two logistic functions

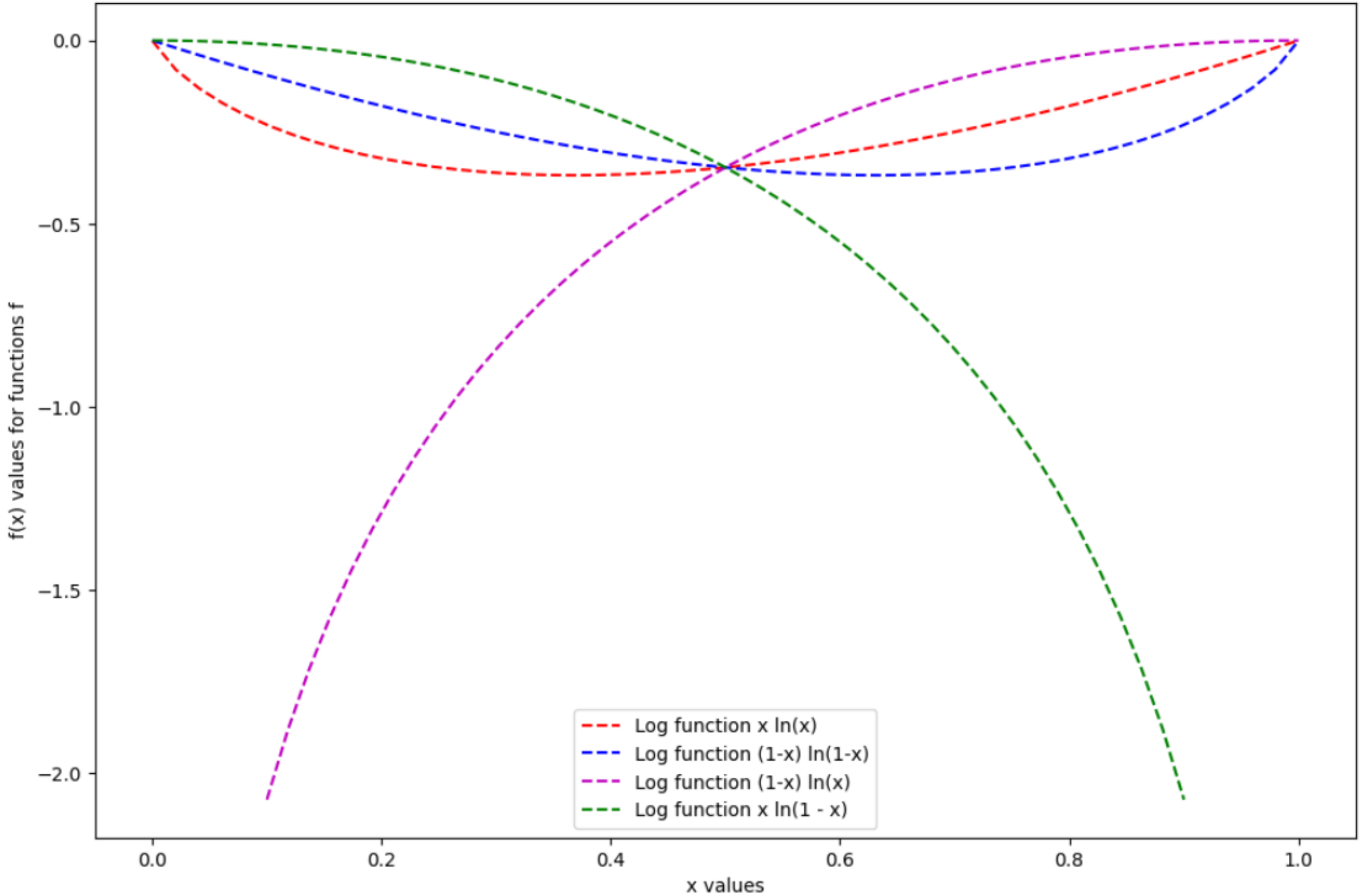
In addition, the following properties hold (some of them require using L'Hospital rule to prove them, try it out?).

$$\lim_{x \to 0} (1 - x)\, ln(x) = -\infty$$

$$\lim_{x \to 1} (1 - x)\, ln(x) = 0$$

$$\lim_{x \to 0} x\, ln(1 - x) = 0$$

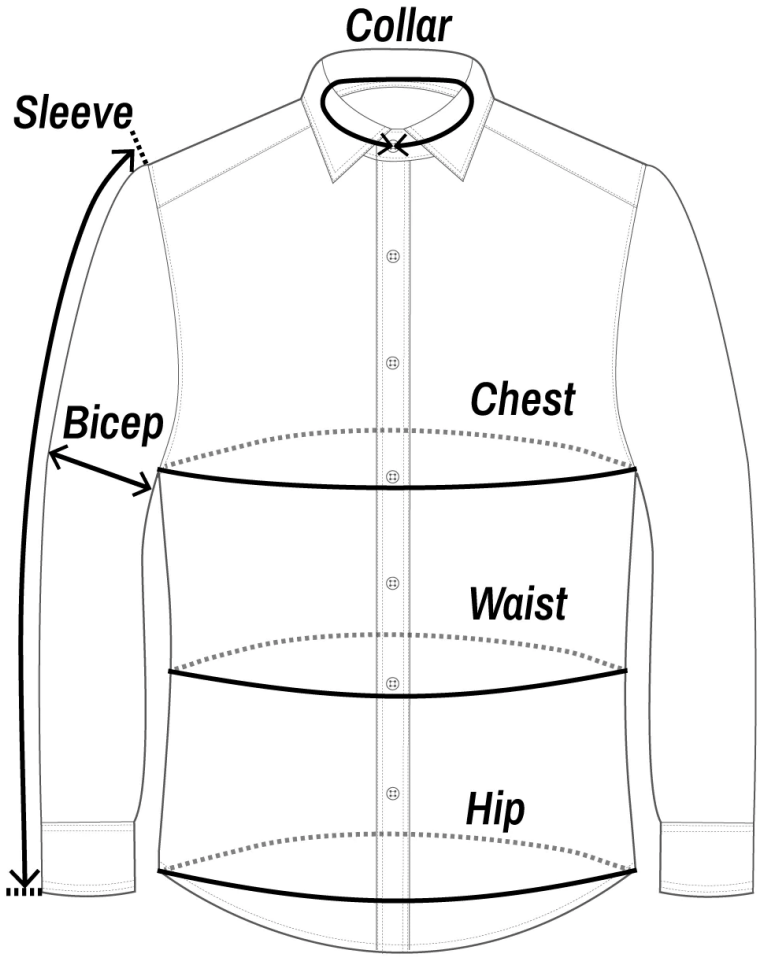
$$\lim_{x \to 1} x\, ln(1 - x) = -\infty$$


0.0
−0.5
−1.0
−1.5
−2.0
f(x) values for functions f
0.0
0.2
0.4
0.6
0.8
1.0
x values
Log function x ln(x)
Log function (1-x) ln(1-x)
Log function (1-x) ln(x)
Log function x ln(1 - x)

Mock dataset generation

In notebook 6, we will prepare a dataset for a **single-variable binary classification.**

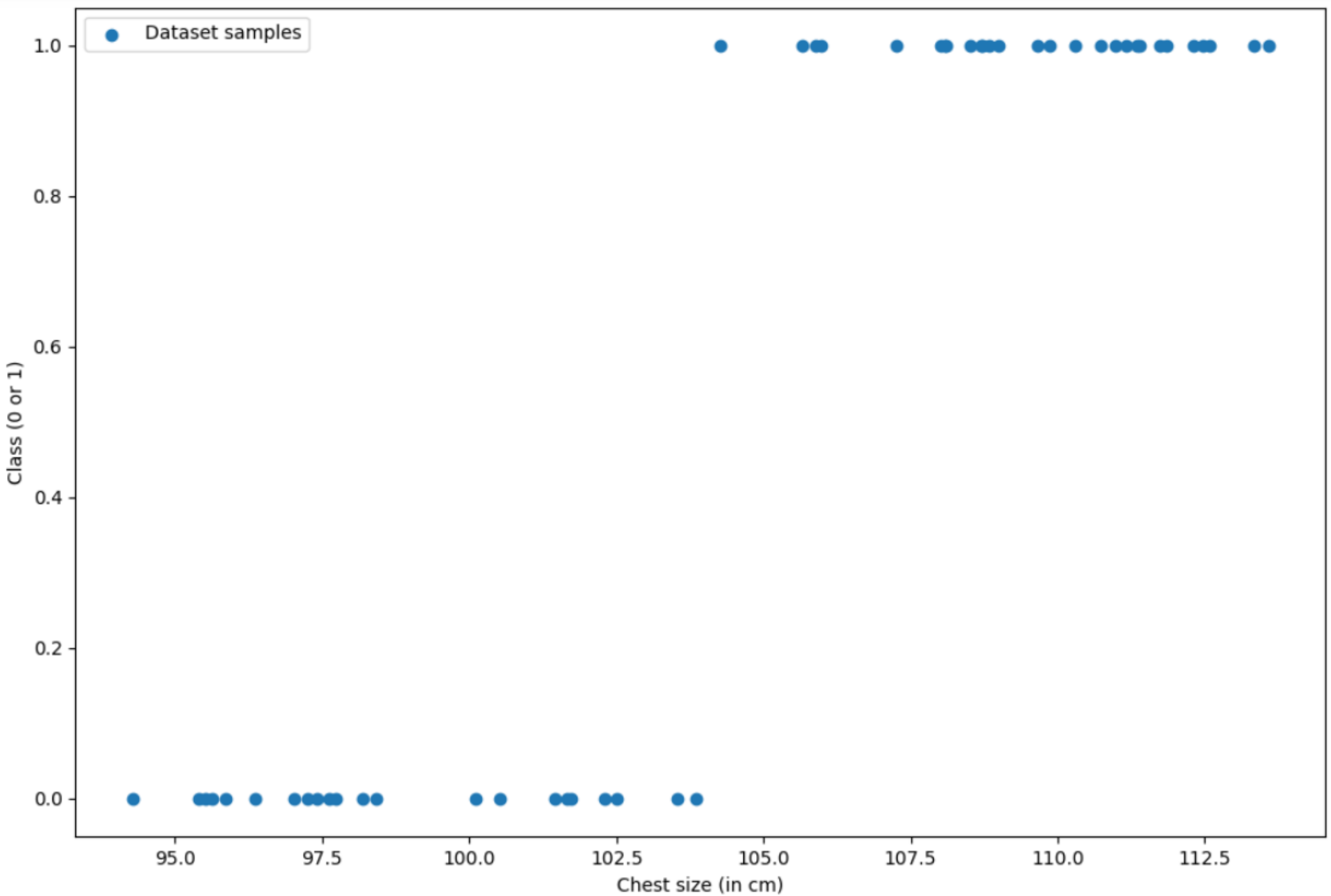
- The **inputs**, x, will consist of chest sizes for male between 94cm and 114cm, randomly generated.
- The **outputs**, y, will consists of **two classes**, corresponding to the shirt size to use, with two possible values:
  - M (class with value 0, if chest size is below 104 cm) size
  - and L size (class with value 1, if chest size is above 104 cm).



```python
# All helper functions
def chest_size(min_size, max_size):
    return round(np.random.uniform(min_size, max_size), 2)
def shirt_size(size, threshold):
    return int(size >= threshold)
def generate_datasets(n_points, min_size, max_size, threshold):
    x = np.array([chest_size(min_size, max_size) for _ in range(n_points)])
    y = np.array([shirt_size(i, threshold) for i in x])
    return x, y
```

```python
# Dataset generation (n_points points will be generated).
# We will use a seed for reproducibility.
min_size = 94
max_size = 114
threshold = 104
np.random.seed(27)
n_points = 50
inputs, outputs = generate_datasets(n_points, min_size, max_size, threshold)
print(inputs)
print(outputs)
```

```
[102.51 110.29 108.71 111.36 101.67 113.59 111.86  98.19 108.84 107.26
 111.74 111.16 108.99 111.4   97.74 100.51 101.46 109.87  97.02  97.4
  95.62 100.1  109.67  97.26  95.41 108.02  97.62 105.98 102.31 104.27
  98.41 108.51 110.99 112.58 108.72 103.53 103.86 105.89  95.52  96.35
 113.33 105.67  95.85  94.27 110.74 112.3  108.09 101.74 108.11 112.47]
[0 1 1 1 0 1 1 0 1 1 1 1 1 1 0 0 0 1 0 0 0 0 1 0 0 1 0 1 0 1 0 1 1 1 1 0 0
 1 0 0 1 1 0 0 1 1 1 0 1 1]
```


Dataset samples
1.0
0.8
0.6
0.4
0.2
0.0
Class (0 or 1)
95.0
97.5
100.0
102.5
105.0
107.5
110.0
112.5
Chest size (in cm)

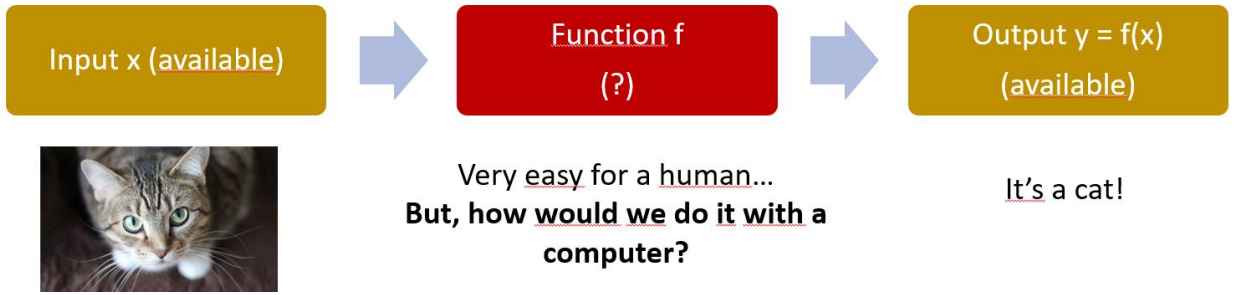
A quick word about expert systems

Definition (expert systems):

An **expert system** is a computer program that mimics the decision-making abilities of a human expert in a specific domain or field.

It uses knowledge represented in a symbolic form, such as rules or ontologies, to make inferences and decisions based on the data it receives.

Expert system provide a high level of performance and accuracy in solving problems or making decisions but these are difficult to design as they require human expertise.



A quick word about expert systems

Definition (expert systems):

An **expert system** is a computer program that mimics the decision-making abilities of a human expert in a specific domain or field.

It uses knowledge represented in a symbolic form, such as rules or ontologies, to make inferences and decisions based on the data it receives.

In our case, we could define as expert system like so, but most of the time the datasets will not have a clear logic and you will have to rely on ML algorithms instead, basically “praying for the AI to figure out a logic that works”.

```
1  # A simple expert system
2  def expert_system(inputs):
3      return [int(size > 104) for size in inputs]
```

```
1  # Try it and compare results to see we have 100% accuracy here!
2  pred = expert_system(inputs)
3  print((pred == outputs).all())
```

```
True
```

From Linear Regression to Logistic Regression

Definition(Logistic Regression):

The **logistic regression** model assumes that the classes y_i for every sample x_i are connected via the formula below.

$$\forall i, y_i = \begin{cases} 1 & if\ p(x_i) > 0.5 \\ 0 & otherwise \end{cases}$$

This value $p(x)$ is often referred to as the **probability** of sample with value x being of class 1 (that is L size).

Respectively, $1 - p(x)$ is therefore the **probability** of sample with value x being of class 0 (that is M size).

The **probability** function is then simply defined as

$$p(x) = s(ax + b)$$

The function p always has values between 0 and 1, thanks to the sigmoid function s we defined earlier.

From Linear Regression to Logistic Regression

Definition(Logistic Regression):

The **logistic regression** model assumes that the classes y_i for every sample x_i are connected via the formula below.

$$\forall i, y_i = \begin{cases} 1 & if\ s(a\,x\ +\ b) > 0.5 \\ 0 & otherwise \end{cases}$$

This value $s(ax + b)$ is often referred to as the **probability** of sample with value x being of class 1 (that is L size).

This model has **two trainable parameters**, $\boldsymbol{a}$ and $\boldsymbol{b}$, to be decided like in the linear regression.

In fact, we could see the logistic regression as the combination of linear regression and sigmoid together.

In addition, note that the function $f(x) = a\,x + b$ is also often referred to as the **logit** function with value x.

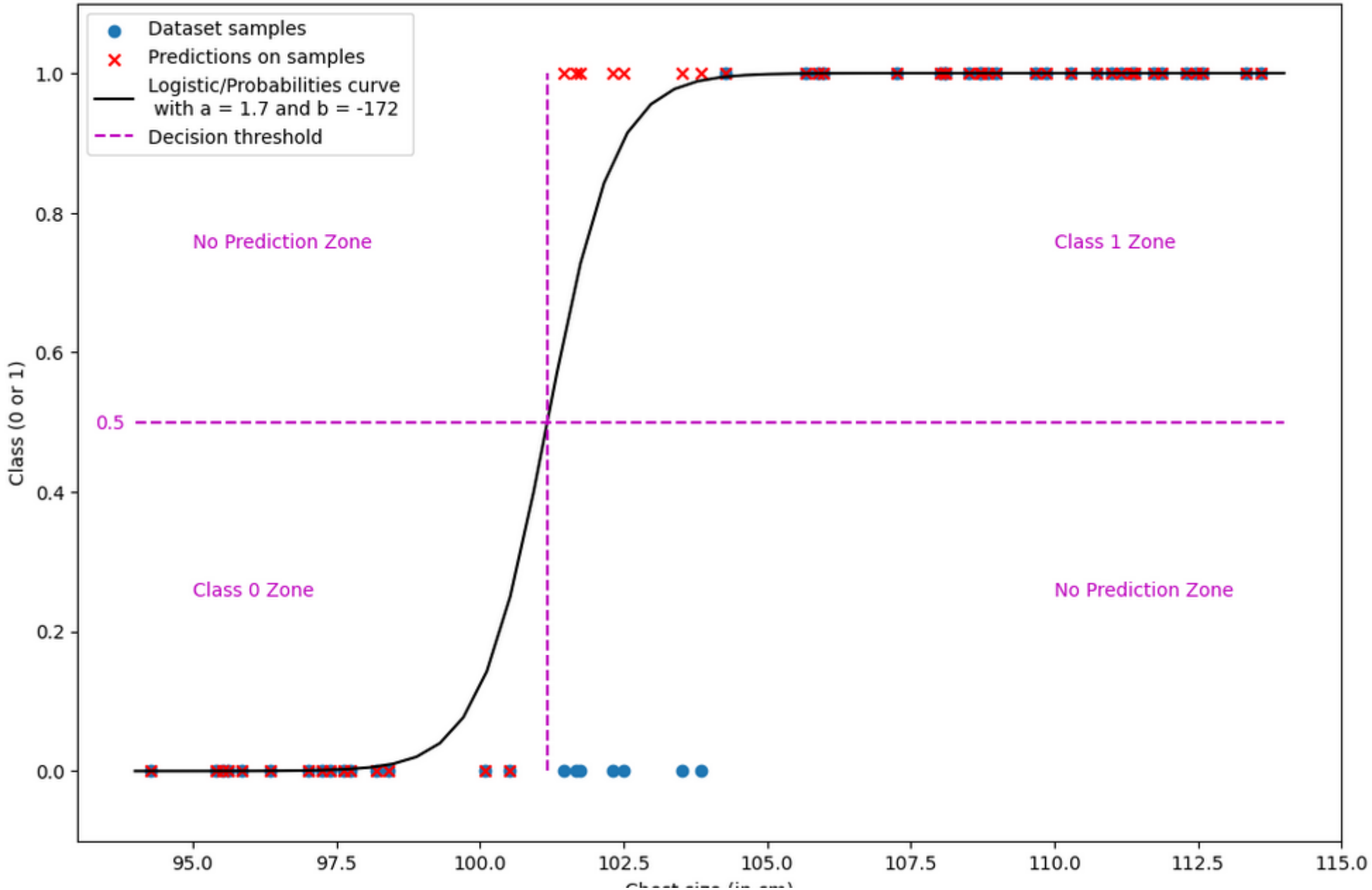
From Linear Regression to Logistic Regression

We can implement the logistic regression below and try it out with some values a and b!

```
def logistic_regression(x, a, b):
    return s(a*x + b)
def predict_samples(inputs, a, b):
    return [int(logistic_regression(x, a, b) >= 0.5) for x in inputs]
def logistic_regression_matplotlib(a, b, min_size, max_size):
    x_plt = np.linspace(min_size, max_size, 50)
    y_plt = np.array([s(a*x + b) for x in x_plt])
    return x_plt, y_plt
```

```
# Trying to predict with our logistic regression
# model, for two given values of a and b.
a = 1.7
b = -172
pred_y = predict_samples(inputs, a, b)
print(pred_y)
```

```
[1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0,
1, 1, 0, 0, 1, 1, 1, 1, 1, 1]
```



Training a logistic regressor

- As before, in linear regression, we will have to **train the model**, i.e. choose values for a and b, that fit the dataset.
- While **MSE** seemed to work as a **loss function** to minimize for the **linear regression** model, it will simply not do here.
- We need something else.

- In logistic regression, we **minimize** a function, called the **log-likelihood** function, instead (after it has been multiplied by -1).
- This function has to do with the two logistic functions we have introduced earlier.

The log-likelihood function and loss

Definition (the log-likelihood function):

The **log-likelihood function** is defined as

$$L = \frac{1}{N}\left[x_i \ln\big(s(ax_i + b)\big) + (1 - x_i)\ln\big(1 - s(ax_i + b)\big)\right]$$

This is a good function to use for our problem for the two reasons:

- When $p(x_i)$ is close to the ground truth value y_i, both terms of the loss function, will have values close to 0. This follows from the properties we defined earlier for the logistic functions.
- On the opposite, the log-likelihood loss function will have a different behaviour when $p(x_i)$ and the ground truth value y_i are different, producing larger non-zero negative values.

The log-likelihood function and loss

Definition (using the log-likelihood function as a loss):

This log-likelihood loss function is therefore a good choice of a performance metric to measure the performance of our models.

Indeed, **maximizing this log-likelihood** loss function would be equivalent to finding the best choice of parameters a and b, and therefore the best fit.

In practice however, we **prefer to minimize loss functions**. This requires a simple adjustment, simply **multiplying the loss function by -1**.

$$a^*, b^* = \arg\min_{a,b} \left[\frac{-1}{N} \left[x_i \ln\big(s(ax_i + b)\big) + (1 - x_i) \ln\big(1 - s(ax_i + b)\big) \right] \right]$$

Implementing the log-likelihood loss

- We can simply implement the log likelihood loss as shown below.
- We could look for the normal equation for logistic regression, but this simply will not work.
- Indeed, it is simply impossible to solve the logistic regression problem analytically.

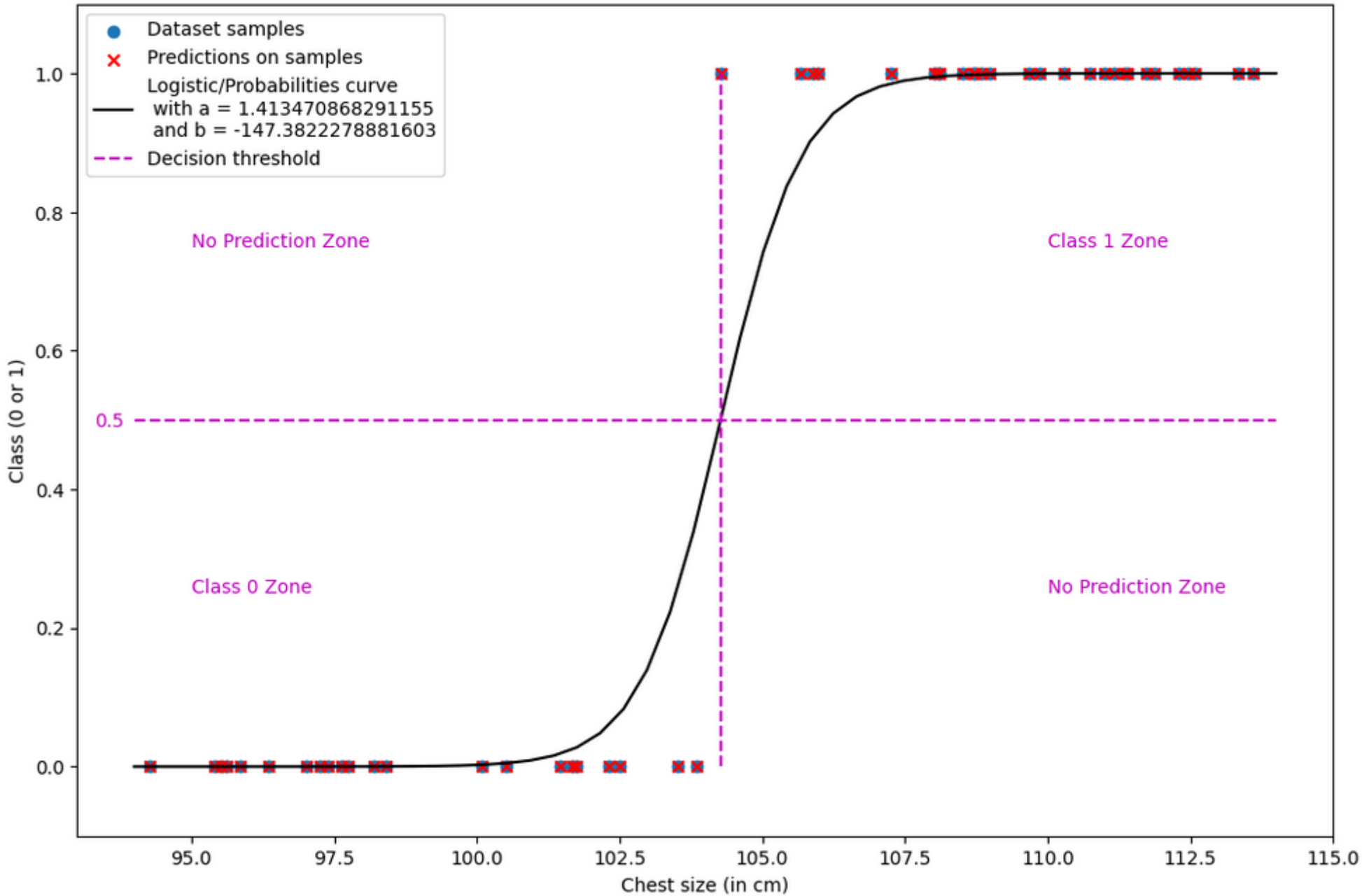
```
1  def log_likelihood_loss(a, b, x, y):
2      # Using Numpy broadcasting
3      return -1*np.mean(y*np.log(logistic_regression(x, a, b)) \
4                        + (1 - y)*np.log(1 - logistic_regression(x, a, b)))
```

Training a logistic regressor

- **Instead, it is often preferable to rely on heuristics, such as the gradient descent method, or the Newton method.**
- Typically, this is what the **LogisticRegression()** object in sklearn uses for training!
- (Not covered but feel free to try it out?)

```
# Reshaping inputs as a 2D array
# (As before, this is a requirement for sklearn models)
inputs_re = inputs.reshape(-1, 1)

# Create a logistic regression model, use the Newton method
# to find the best parameters a and b
logreg_model = LogisticRegression(solver = 'newton-cg')
logreg_model.fit(inputs_re, outputs)

# Test your model and inputs
pred_outputs = logreg_model.predict(inputs_re)
print(pred_outputs)
print(outputs)

# Retrieving coefficients a and b
a_sk = logreg_model.coef_[0, 0]
b_sk = logreg_model.intercept_[0]
print(a_sk, b_sk)
```

```
[0 1 1 1 0 1 1 0 1 1 1 1 1 1 0 0 0 1 0 0 0 0 1 0 0 1 0 1 0 1 0 1 1 1 1 0 0
 1 0 0 1 1 0 0 1 1 1 0 1 1]
[0 1 1 1 0 1 1 0 1 1 1 1 1 1 0 0 0 1 0 0 0 0 1 0 0 1 0 1 0 1 0 1 1 1 1 0 0
 1 0 0 1 1 0 0 1 1 1 0 1 1]
1.413470868291155 -147.3822278881603
```



Evaluating the logistic regression model

- When attempting to evaluation a classification model, it is often a good idea to first display a confusion matrix for the predicted values of the dataset samples.
- We can simply calculate it by using the confusion_matrix() function from sklearn. The non-diagonal elements have zero values, which is the sign that our model classifies perfectly.
- Learn more, here: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html.

```
1  # Using a confusion matrix.
2  print(confusion_matrix(outputs, pred_outputs))
```

```
[[22  0]
 [ 0 28]]
```

```
1  # Using a classification report.
2  print(classification_report(outputs, pred_outputs))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        22
           1       1.00      1.00      1.00        28

    accuracy                           1.00        50
   macro avg       1.00      1.00      1.00        50
weighted avg       1.00      1.00      1.00        50
```

Evaluating the logistic regression model

- We can also ask for a classification report for the model.
- This shows the precision, recall and F1 scores for each class, as well as the overall accuracy of the model.
- For each one of these metrics, the best value is 1.0, and our model exhibits it, suggesting again that the model classifies perfectly.
- Learn more, here: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html#sklearn.metrics.classification_report.

```
1 # Using a confusion matrix.
2 print(confusion_matrix(outputs, pred_outputs))
```

```
[[22  0]
 [ 0 28]]
```

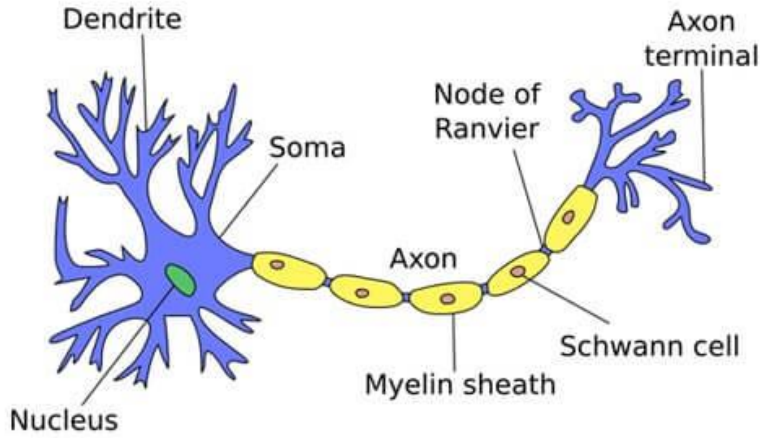
```
1 # Using a classification report.
2 print(classification_report(outputs, pred_outputs))
```

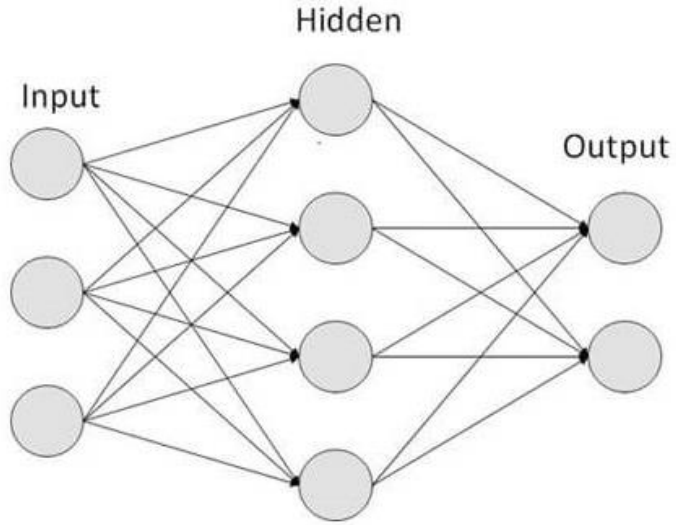
```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        22
           1       1.00      1.00      1.00        28

    accuracy                           1.00        50
   macro avg       1.00      1.00      1.00        50
weighted avg       1.00      1.00      1.00        50
```

A bit of biology

Question: are neurons in Neural Networks really close to the actual human brain?

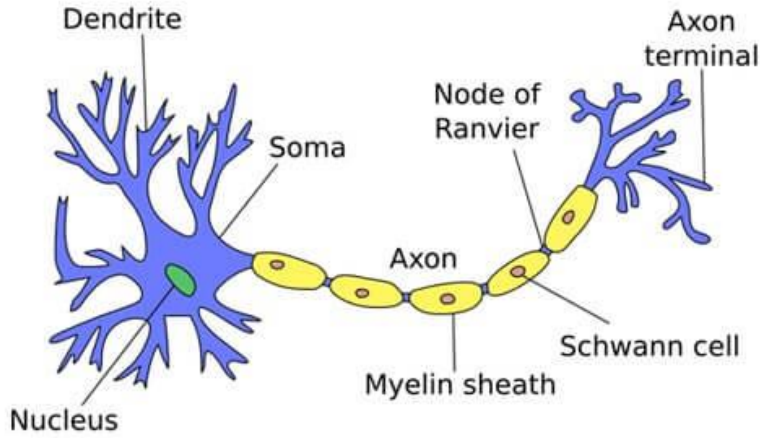
- Neural networks are inspired by the neural structure and processes of the human brain.
- Neural networks are comprised of interconnected nodes, which are called neurons.
- Neural networks are related to biology in that they are modeled after biological neural systems.

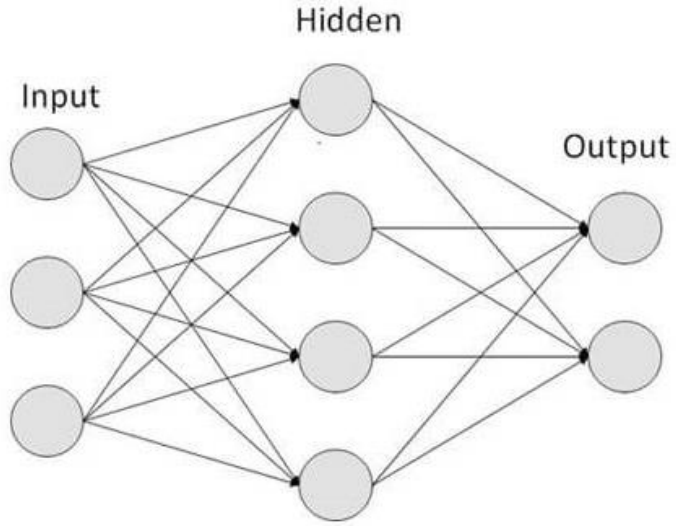




A bit of biology

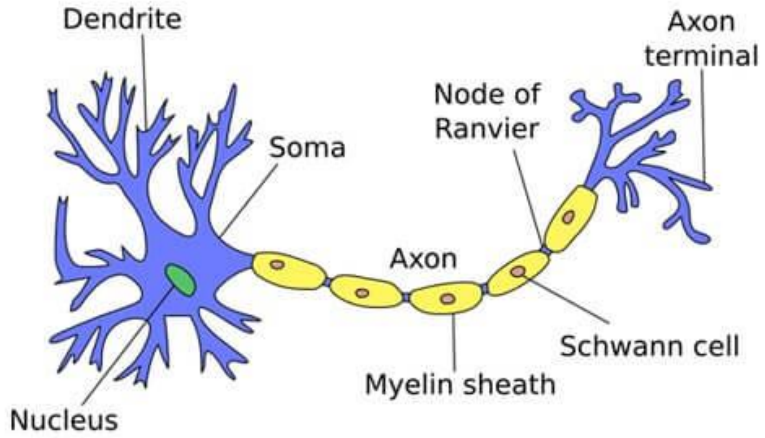
- The neurons in a neural network are similar to the neurons in the brain in that they are connected to each other and can send and receive signals.
- Furthermore, the training process of a neural network is analogous to the learning process of a biological neural network.
- The neurons are adjusted and adapted in order to learn, just as neurons in the brain are modified in order to learn new information.

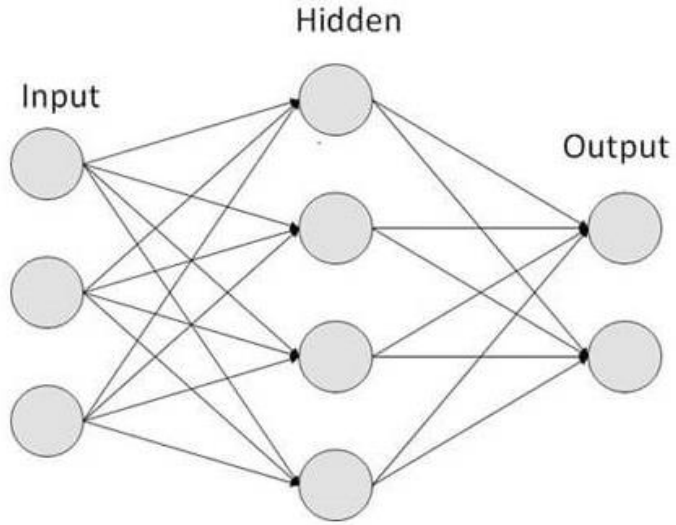




A bit of biology

- Research has also shown that artificial neural networks can be used to solve some of the same types of problems that biological neural networks can solve.
- For example, neural networks can be used to identify objects in images, just as biological neural networks can.
- Ongoing argument: see [Quanta2021] and [MITNews2022].





Starting with a mock dataset, again

As before, we will try to come up with a model capable of predicting the price of an apartment based on some of its features.

The inputs will this time consist of two parameters:

- the **surface** of the apartment, in sqm, just like before,
- and the **distance** between the apartment and the closest MRT station in meters.

We will generate a mock dataset by:

- randomly drawing **surfaces** between 40 sqm and 200sqm,
- randomly drawing **distances** between 50 and 1000m,
- generate prices by assuming that the average price is simply defined as $avgprice = 100000 + 14373 \times surface + (1000 - distance) \times 1286$

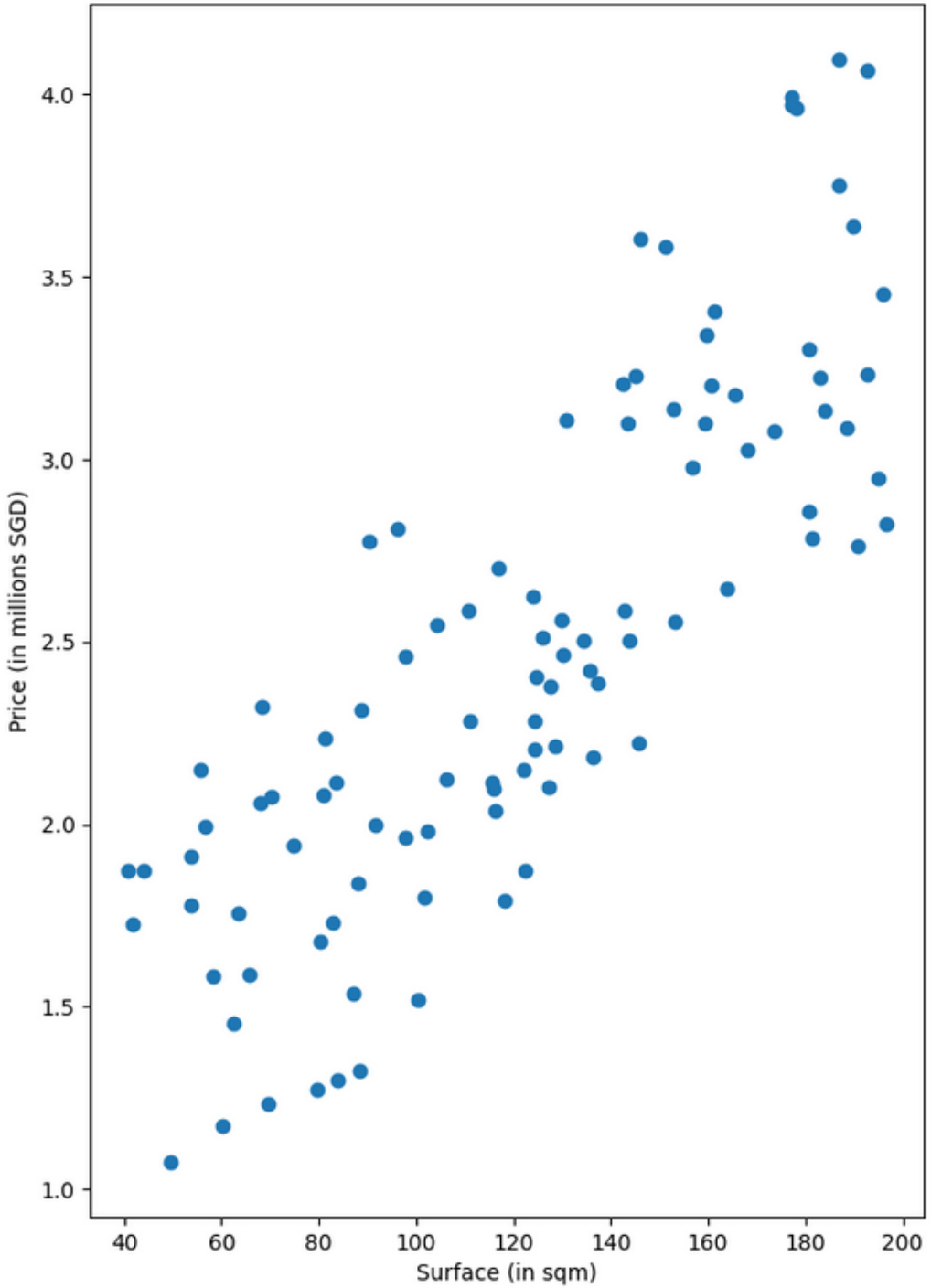
Finally, we will randomly apply a -/+ 10% variation on the average price to create some variance, and call that the output for a given sample.

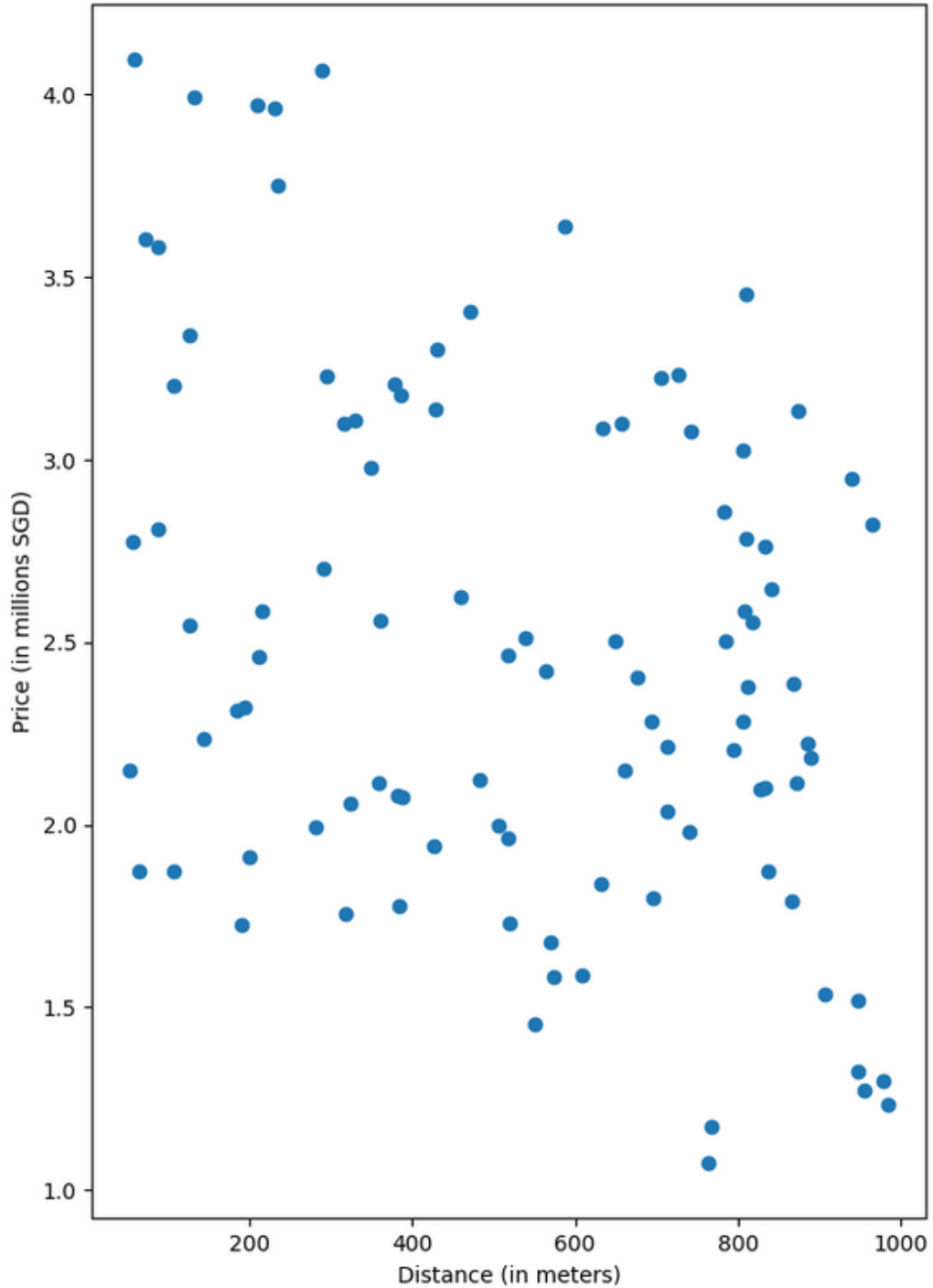
Starting with a mock dataset, again

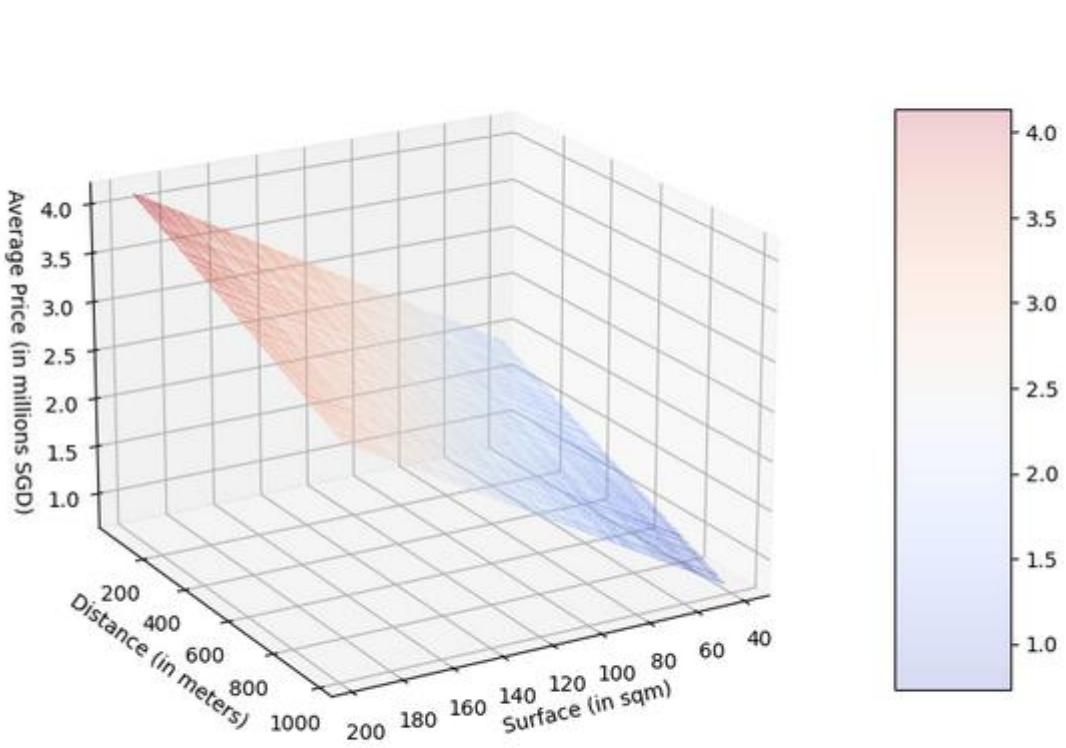
```python
# All helper functions
min_surf = 40
max_surf = 200
def surface(min_surf, max_surf):
    return round(np.random.uniform(min_surf, max_surf), 2)
min_dist = 50
max_dist = 1000
def distance(min_dist, max_dist):
    return round(np.random.uniform(min_dist, max_dist), 2)
def price(surface, distance):
    return round((100000 + 14373*surface + (1000 - distance)*1286)*(1 + np.random.uniform(-0.1, 0.1)))/1000000
n_points = 100
def create_dataset(n_points, min_surf, max_surf, min_dist, max_dist):
    surfaces_list = np.array([surface(min_surf, max_surf) for _ in range(n_points)])
    distances_list = np.array([distance(min_dist, max_dist) for _ in range(n_points)])
    inputs = np.array([[s, d] for s, d in zip(surfaces_list, distances_list)])
    outputs = np.array([price(s, d) for s, d in zip(surfaces_list, distances_list)]).reshape(n_points, 1)
    return surfaces_list, distances_list, inputs, outputs
```

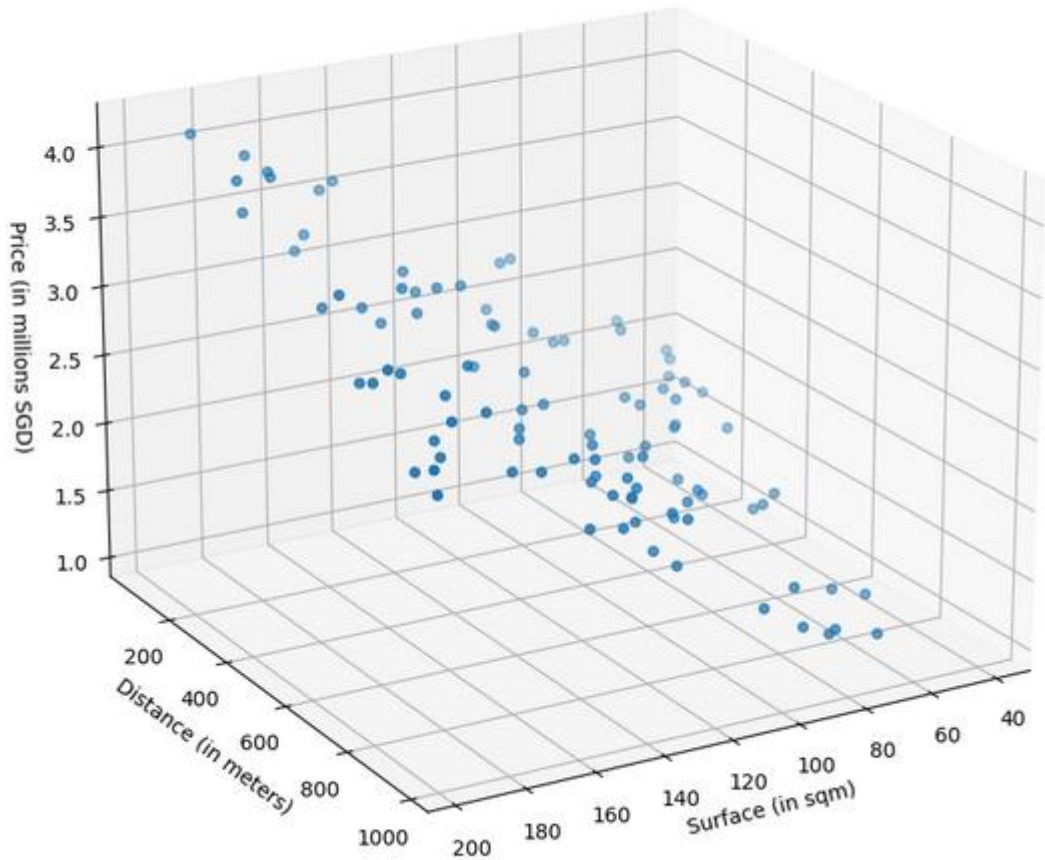
```python
# Generate dataset
np.random.seed(47)
surfaces_list, distances_list, inputs, outputs = create_dataset(n_points, \
                                                                min_surf, \
                                                                max_surf, \
                                                                min_dist, \
                                                                max_dist)
# Check a few entries of the dataset
print(surfaces_list.shape)
print(distances_list.shape)
print(inputs.shape)
print(outputs.shape)
print(inputs[0:10, :])
print(outputs[0:10])
```

```
(100,)
(100,)
(100, 2)
(100, 1)
[[ 58.16 572.97]
 [195.92 809.8 ]
 [156.6  349.04]
 [ 96.23  86.82]
 [153.22 817.92]
 [167.94 806.25]
 [143.29 315.92]
 [106.34 482.67]
 [152.96 427.77]
 [ 79.46 955.76]]
[[1.581913]
 [3.450274]
 [2.978769]
 [2.808258]
 [2.556398]
 [3.023983]
 [3.099523]
 [2.121069]
 [3.136544]
 [1.273443]]
```


Price (in millions SGD)
4.0
3.5
3.0
2.5
2.0
1.5
1.0
40
60
80
100
120
140
160
180
200
Surface (in sqm)


Price (in millions SGD)
4.0
3.5
3.0
2.5
2.0
1.5
1.0
200
400
600
800
1000
Distance (in meters)





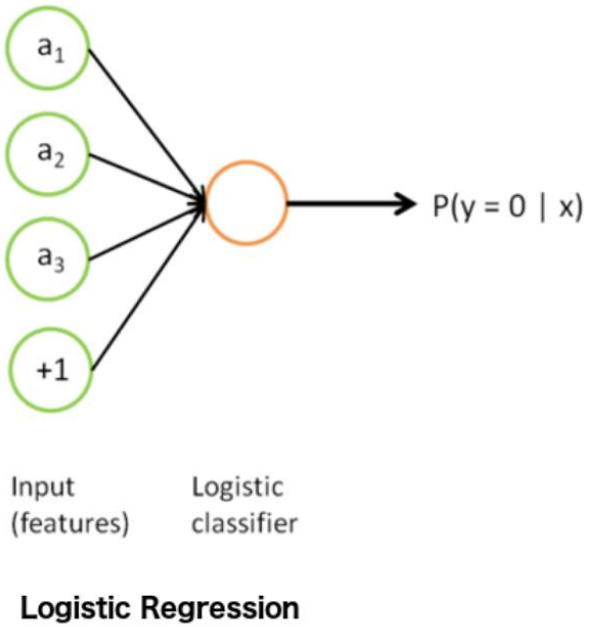
Coding a Neuron and a Neural Network

Definition (Neuron in Neural Networks):

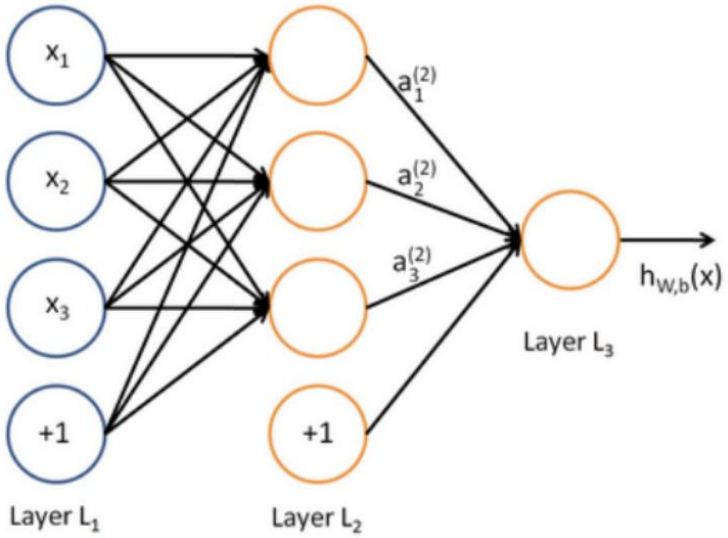
Each artificial neuron in a neural network is like a logistic regression model, which receives input, applies a transformation on that input, and produces an output.

And similar to the way that biological neurons are connected and communicate with each other, artificial neurons are connected to each other through a network of edges, which can be thought of as pathways for information to flow through the network.

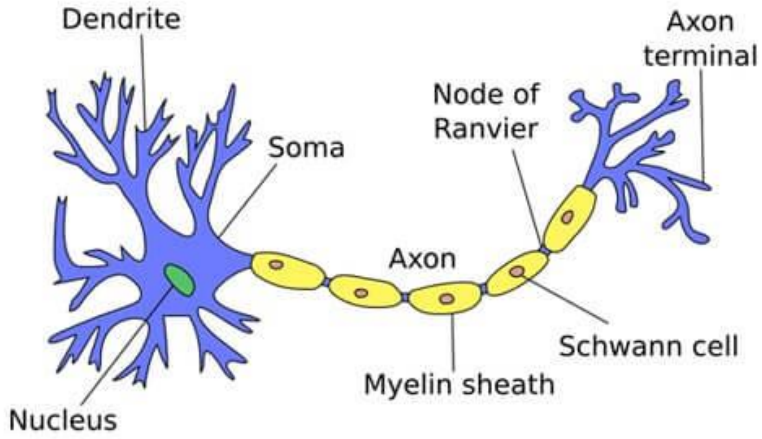
Coding a Neuron and a Neural Network



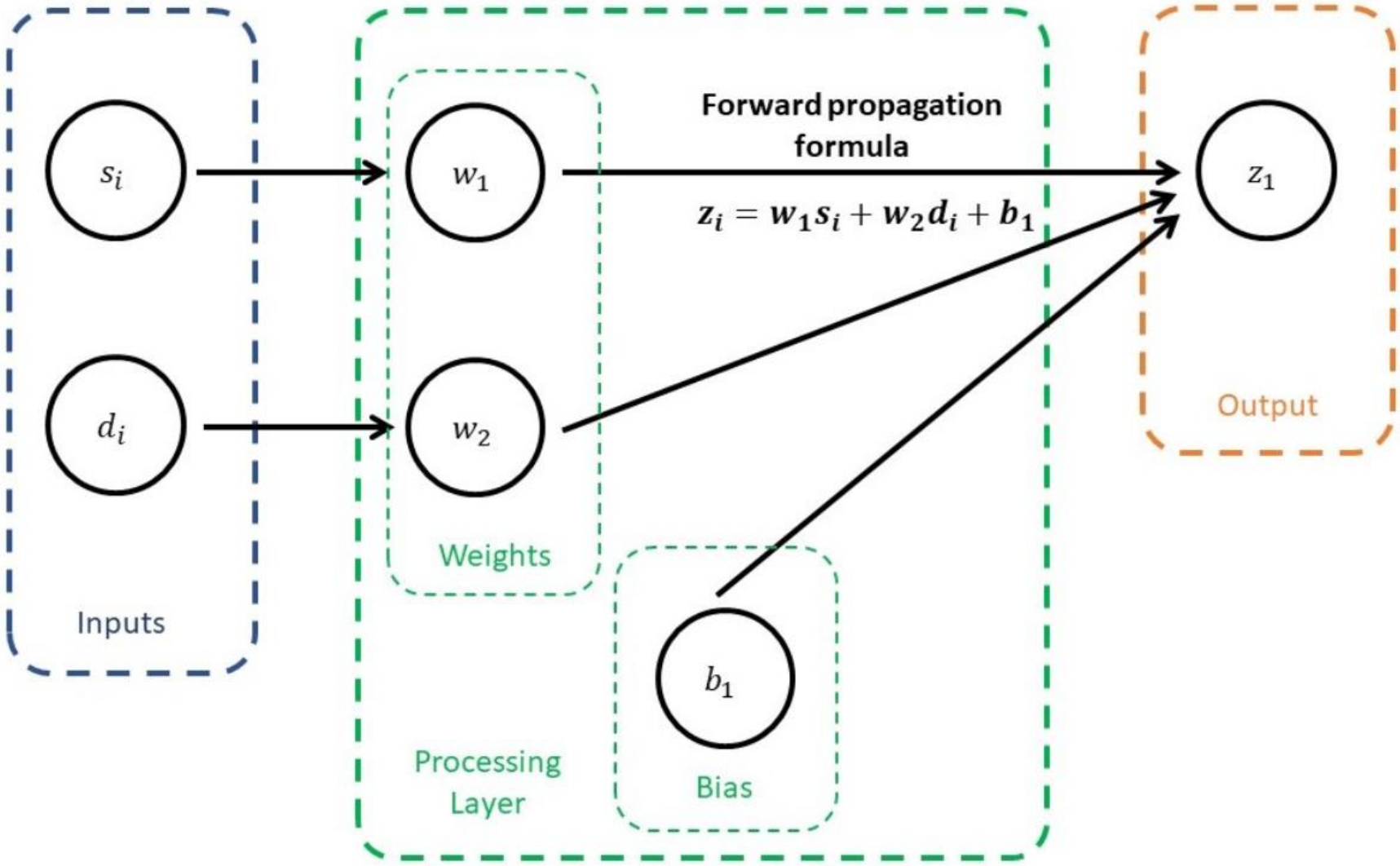
Logistic Regression



Neural Network



Coding a Neuron and a Neural Network



```python
# Let us define matrix W first, with two parameters w1 and w2.
# And the matrix b as well.
# Are these values randomly defined?
W = np.array([[14373/1000000], [-1286/1000000]]) # 14373/1000000 and -1286/1000000
b = np.ones(shape = (1, 1))*1.386 # 100000/1000000 + 1000*1286/1000000
```

```python
# We can then implement the multiplication operation
# we defined above by using the matmul operation.
Z = np.matmul(inputs, W)
pred = Z + b
# We can then check the shape of the matrices we obtained
print(inputs.shape)
print(W.shape)
print(b.shape)
print(Z.shape)
print(pred.shape)
print(outputs.shape)
```

```
(100, 2)
(2, 1)
(1, 1)
(100, 1)
(100, 1)
(100, 1)
```

Defining a minimal Neural Network

Definition (defining a minimal Neural Network):

In order to define a minimal neural network, we need two things:

- First, an **__init__ method,** listing the **trainable parameters** for the model.
- In our case that is a **weight vector** $W = (w_1, w_2)$ with 2 elements and a single scalar **bias** value b.
- Second, a **forward method**, which will be used to formulate predictions for any set of given inputs.
- The **prediction formula** is simply $y_i = w_1 s_i + w_2 d_i + b$.

```python
class SimpleNeuralNet():

    def __init__(self, W, b):
        # Weights and biases matrices
        self.W = W
        self.b = b

    def forward(self, x):
        # Wx + b operation as above
        Z = np.matmul(x, self.W)
        pred = Z + self.b
        return pred
```

```python
# Create a minimal neuron neural network using our class
simple_neural_net = SimpleNeuralNet(W = np.array([[14373/1000000], [-1286/1000000]]), \
                                    b = np.ones(shape = (1, 1))*1.386)
print(simple_neural_net.__dict__)
```

```
{'W': array([[ 0.014373],
       [-0.001286]]), 'b': array([[1.386]])}
```

```python
# Predict and show first five samples
pred = simple_neural_net.forward(inputs)
print(pred[:5])
```

```
[[1.48509426]
 [3.16055536]
 [3.18794636]
 [2.65746327]
 [2.53638594]]
```

Adding a loss

- This takes care of the first three elements of a machine learning model (task, dataset, model).
- The fourth element we need is a loss function.
- As this is a regression class again, we can reuse the MSE loss and add a method to our class.
- We will also reuse the forward method we just coded to make predictions and then compared said predictions with the expected outputs from our dataset.

Adding a loss

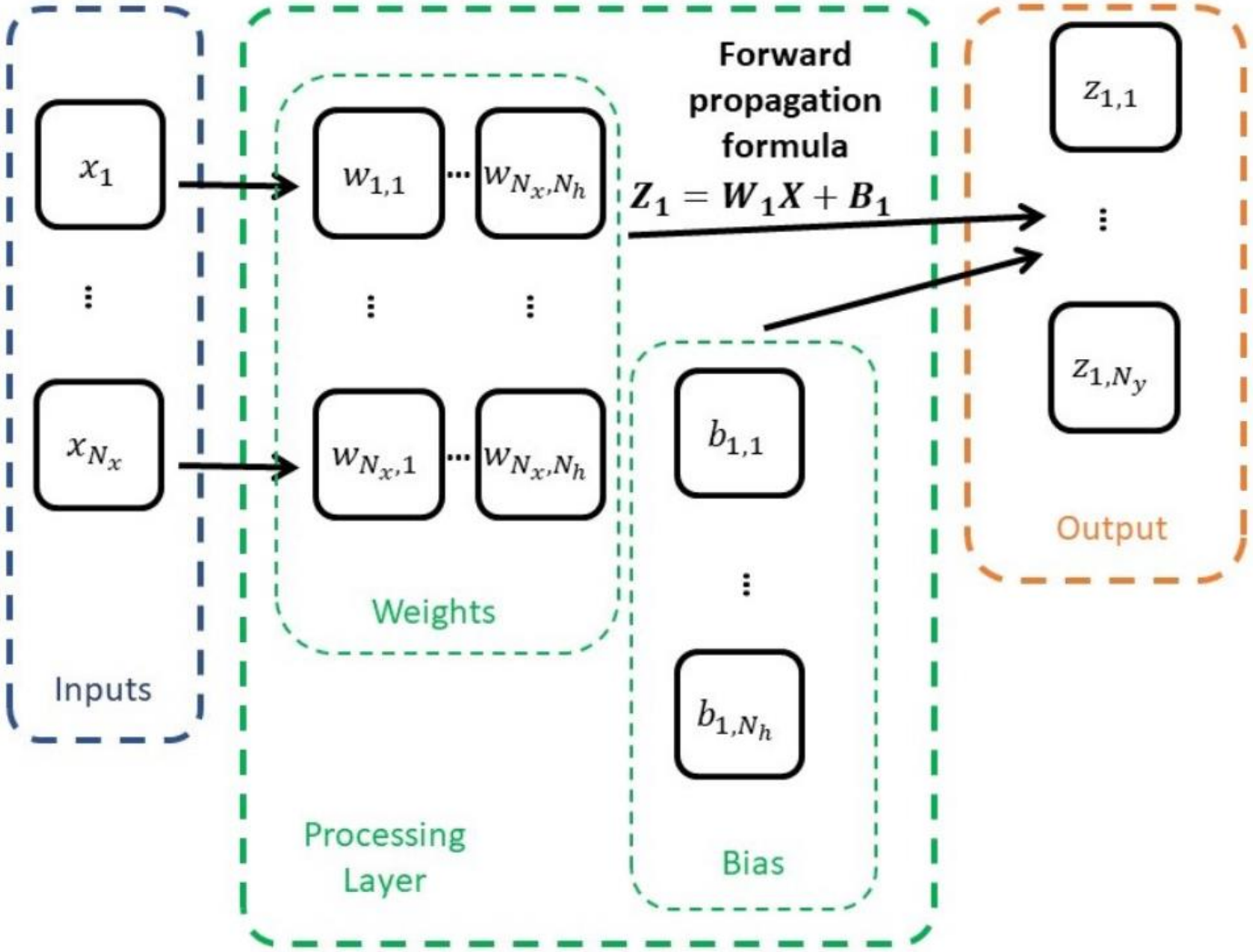
```python
class SimpleNeuralNet():

    def __init__(self, W, b):
        # Weights and biases matrices
        self.W = W
        self.b = b
        # Loss, initialized as infinity before first calculation is made
        self.loss = float("Inf")

    def forward(self, inputs):
        Z = np.matmul(inputs, self.W)
        pred = Z + self.b
        return pred

    def MSE_loss(self, inputs, outputs):
        outputs_re = outputs.reshape(-1, 1)
        pred = self.forward(inputs)
        losses = (pred - outputs_re)**2
        self.loss = np.sum(losses)/outputs.shape[0]
        return self.loss
```

Scaling up with more parameters

Technically, our Neural Network could operate with any number of inputs N_x and any number of outputs N_y.

We would simply need to adjust the sizes of the parameters, making:

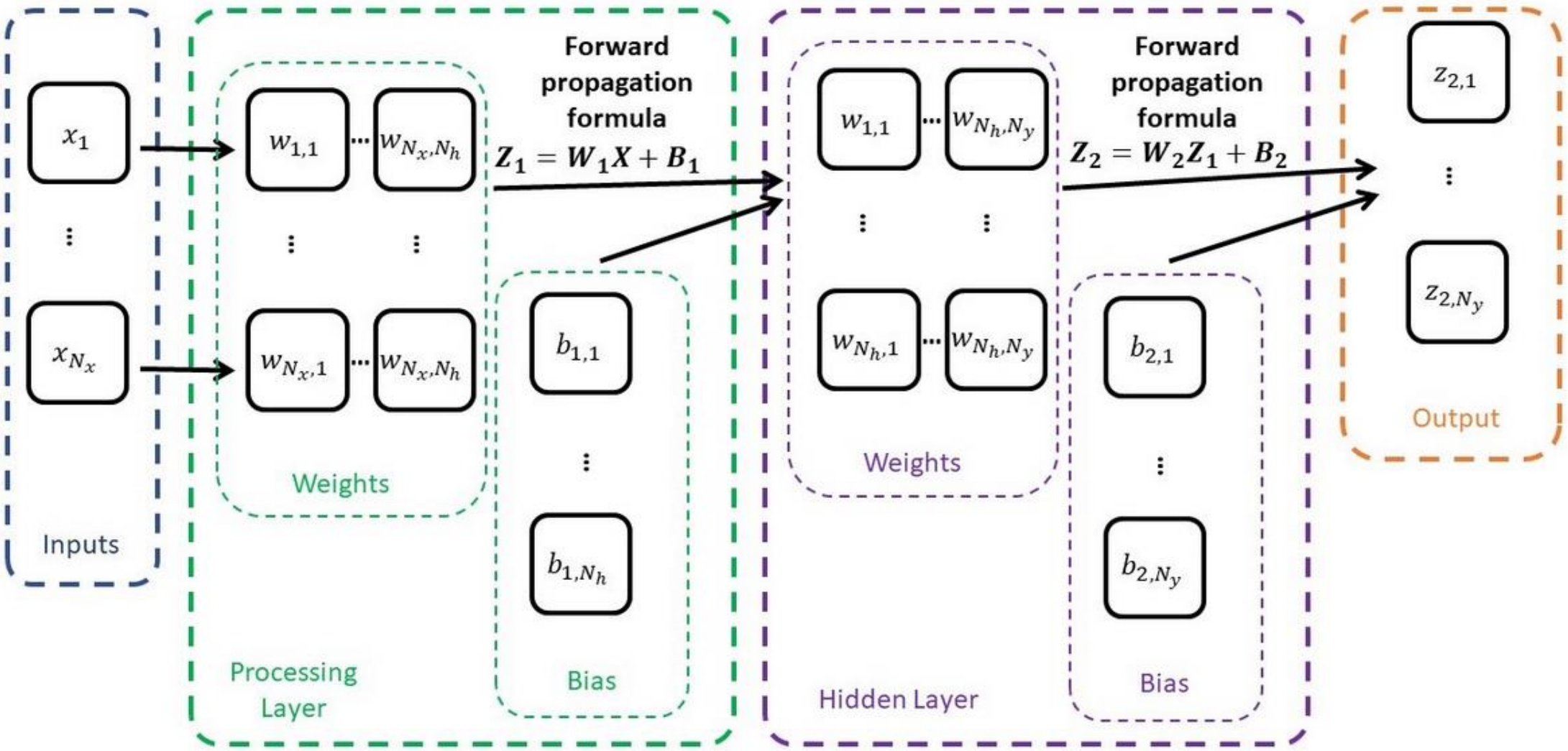
- W as a $N_x \times N_y$ matrix,
- B as a N_y vector.



Scaling up with more layers

- Following the logic of our SimpleNeuralNet, we could now implement a Shallow Neural Network, with its own class ShallowNeuralNet.
- It will simply include two processing layers instead of just one, both implementing a $WX + b$ operation of some sort.
- The first layer, will produce receive inputs with dimensionality n_x and produce outputs with dimensionality n_h. The matrix W for this layer will therefore be of size $n_x \times n_h$, and the matrix b will be a simple 1D vector with size n_h.
- The second layer, also called **hidden layer**, will produce receive inputs from the previous layer with dimensionality n_h and will produce outputs matching the dimensionality of the outputs in our dataset, that is n_y. The matrix W for this layer will therefore be of size $n_h \times n_y$, and the matrix b will be a simple 1D vector with size n_y.

Scaling up with more layers



Our NN class

Changes:

- Weights and biases are now randomly generated, instead of being passed to the init method.
- We pass the expected sizes n_x, n_h, n_y instead.
- Initializing as normal random with zero mean and variance 1.

```python
class ShallowNeuralNet():

    def __init__(self, n_x, n_h, n_y):
        # Network dimensions
        self.n_x = n_x
        self.n_h = n_h
        self.n_y = n_y
        # Weights and biases matrices
        self.W1 = np.random.randn(n_x, n_h)*0.1
        self.b1 = np.random.randn(1, n_h)*0.1
        self.W2 = np.random.randn(n_h, n_y)*0.1
        self.b2 = np.random.randn(1, n_y)*0.1
        # Loss, initialized as infinity before first calculation is made
        self.loss = float("Inf")

    def forward(self, inputs):
        # Wx + b operation for the first layer
        Z1 = np.matmul(inputs, self.W1)
        Z1_b = Z1 + self.b1
        # Wx + b operation for the second layer
        Z2 = np.matmul(Z1_b, self.W2)
        Z2_b = Z2 + self.b2
        return Z2_b

    def MSE_loss(self, inputs, outputs):
        # MSE loss function as before
        outputs_re = outputs.reshape(-1, 1)
        pred = self.forward(inputs)
        losses = (pred - outputs_re)**2
        self.loss = np.sum(losses)/outputs.shape[0]
        return self.loss
```

```python
# Define neural network structure
n_x = 2
n_h = 4
n_y = 1
shallow_neural_net = ShallowNeuralNet(n_x, n_h, n_y)
print(shallow_neural_net.__dict__)
```

```
{'n_x': 2, 'n_h': 4, 'n_y': 1, 'W1': array([[-0.10476816,  0.18570216,  0.03204007, -0.10951262],
       [-0.13867874, -0.03539496, -0.02856421,  0.20592501]]), 'b1': array([[ 0.0232776 , -0.16122469,  0.00718537,  0.0666335
1]]), 'W2': array([[ 0.03321156],
       [-0.0336505 ],
       [ 0.04977554],
       [-0.1794089 ]]), 'b2': array([[0.03460341]]), 'loss': inf}
```

1. Initialize model
2. Forward to predict
3. Compute loss to evaluate model (it is bad, because trainable parameters have received random values!)

```python
pred = shallow_neural_net.forward(inputs)
print(pred.shape)
print(outputs.shape)
print(pred[0:5])
print(outputs[0:5])
```

```
(100, 1)
(100, 1)
[[-23.24055489]
 [-31.54945952]
 [-12.75105332]
 [ -2.49026451]
 [-32.3803654 ]]
[[1.581913]
 [3.450274]
 [2.978769]
 [2.808258]
 [2.556398]]
```

```python
loss = shallow_neural_net.MSE_loss(inputs, outputs)
print(loss)
```

```
677.625448852107
```

The need for a training procedure

- At the moment, we have coded a model that can initialize trainable parameters randomly, formulate predictions and evaluate its own performance.
- It is great, but we have no way to compute the weights manually.
- We can only try a few different initialization and pray the RNG gods to give us trainable parameters with a good loss.
- **We need a training procedure!**

```
np.random.seed(963)
shallow_neural_net1 = ShallowNeuralNet(n_x, n_h, n_y)
loss1 = shallow_neural_net1.MSE_loss(inputs, outputs)
shallow_neural_net2 = ShallowNeuralNet(n_x, n_h, n_y)
loss2 = shallow_neural_net2.MSE_loss(inputs, outputs)
shallow_neural_net3 = ShallowNeuralNet(n_x, n_h, n_y)
loss3 = shallow_neural_net3.MSE_loss(inputs, outputs)
shallow_neural_net4 = ShallowNeuralNet(n_x, n_h, n_y)
loss4 = shallow_neural_net4.MSE_loss(inputs, outputs)
print(loss1, loss2, loss3, loss4)
```

```
21.318364917457647 58.190236579106184 4.770288142049728 0.2093294704434788
```

The backpropagation mechanism

Definition (Backpropagation):

Backpropagation is an **algorithm** used to **train** neural networks.

It is a type of **gradient descent** algorithm that allows the network to learn by **adjusting the weights** of the connections between the neurons in the network.

Introduced for the first time in [Rumelhart1986].

Or was it? [Medium2020]

Backpropagation will tell our model **how to adjust** the W and b matrices to improve its loss and prediction capabilities.

This process is called, **backpropagation** and will be implemented in the **backward()** method of our model.

Backpropagation in our model

While our Neural Network seems to be complicated in its architecture, at the end of the day, training it simply consists of solving the optimization problem below.

$$W_1^*, b_1^*, W_2^*, b_2^* = \arg\min_{W_1, b_1, W_2, b_2} \left[\frac{1}{M} \left(Y_{pred}(X, W_1, b_1, W_2, b_2) - Y \right)^2 \right]$$

$$W_1^*, b_1^*, W_2^*, b_2^* = \arg\min_{W_1, b_1, W_2, b_2} \left[\frac{1}{M} \left(W_2 (W_1 X + b_1) + b_2 - Y \right)^2 \right]$$

With M being the number of samples in the dataset. We could use gradient descent, like we did for the Linear Regression, except that this time we have 4 parameters and a loss function that is a bit more complex.

Chain rule, to the rescue!

Let us denote the error term ϵ.

$$\epsilon = Y_{pred}(X, W_1, b_1, W_2, b_2) - Y = W_2(W_1 X + b_1) + b_2 - Y.$$

Using the chain rule, we can simply compute $\frac{\partial L}{\partial W_2}$ as

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial Y_{pred}} \frac{\partial Y_{pred}}{\partial W_2}$$

Chain rule, to the rescue!

Since we have

$$\frac{\partial L}{\partial Y_{pred}} = \frac{2}{M} Y_{pred}(X, W_1, b_1, W_2, b_2) - Y = \frac{2\epsilon}{\mathrm{M}}$$

And

$$\frac{\partial Y_{pred}}{\partial W_2} = \mathrm{W_1 X + b_1}$$

The gradient descent update rule for W_2 is then

$$W_2 \leftarrow W_2 - \frac{2\epsilon\alpha}{\mathrm{M}} (\mathrm{W_1 X + b_1})$$

Chain rule, to the rescue!

Since we have

$$\frac{\partial L}{\partial Y_{pred}} = \frac{2}{M} Y_{pred}(X, W_1, b_1, W_2, b_2) - Y = \frac{2\epsilon}{\mathrm{M}}$$

$$And\ \frac{\partial Y_{pred}}{\partial W_2} = \mathrm{W_1 X + b_1}$$

The gradient descent update rule for $\mathrm{W_2}$ is then

$$\mathrm{W_2} \leftarrow \mathrm{W_2} - \frac{2\epsilon\alpha}{\mathrm{M}} (\mathrm{W_1 X + b_1})$$

Similarly, we have

$$\mathrm{b_2} \leftarrow \mathrm{b_2} - \frac{2\epsilon\alpha}{\mathrm{M}}$$

Chain rule, to the rescue!

Similarly, we have

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial Y_{pred}} \frac{\partial Y_{pred}}{\partial W_1} = \left(\frac{2\epsilon}{\mathrm{M}}\right)(\mathrm{W}_2\mathrm{X})$$

$$And\ \frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial Y_{pred}} \frac{\partial Y_{pred}}{\partial b_1} = \left(\frac{2\epsilon}{\mathrm{M}}\right)(\mathrm{W}_2)$$

And therefore, we have

$$\mathrm{W}_1 \leftarrow \mathrm{W}_1 - \frac{2\epsilon\alpha}{\mathrm{M}}\mathrm{W}_2 X$$

$$\mathrm{b}_1 \leftarrow \mathrm{b}_1 - \frac{2\epsilon\alpha}{\mathrm{M}}\mathrm{W}_2$$

Backpropagation in our model

Update rules for W_2 and b_2

$$W_2 \leftarrow W_2 - \frac{2\epsilon\alpha}{M}(W_1 X + b_1)$$

$$b_2 \leftarrow b_2 - \frac{2\epsilon\alpha}{M}$$

```python
def backward(self, inputs, outputs, alpha = 1e-5):
    # Get the number of samples in dataset
    m = inputs.shape[0]

    # Forward propagate
    Z1 = np.matmul(inputs, self.W1)
    Z1_b = Z1 + self.b1
    Z2 = np.matmul(Z1_b, self.W2)
    y_pred = Z2 + self.b2

    # Compute error term
    epsilon = y_pred - outputs

    # Compute the gradient for W2 and b2
    dL_dW2 = (2/m)*np.matmul(Z1_b.T, epsilon)
    dL_db2 = (2/m)*np.sum(epsilon, axis = 0, keepdims = True)

    # Compute the loss derivative with respect to the first layer
    dL_dZ1 = np.matmul(epsilon, self.W2.T)
```

Backpropagation in our model

Update rules for W_1 and b_1

$$W_1 \leftarrow W_1 - \frac{2\epsilon\alpha}{M} W_2 X$$

$$b_1 \leftarrow b_1 - \frac{2\epsilon\alpha}{M} W_2$$

```
# Compute the loss derivative with respect to the first layer
dL_dZ1 = np.matmul(epsilon, self.W2.T)

# Compute the gradient for W1 and b1
dL_dW1 = (2/m)*np.matmul(inputs.T, dL_dZ1)
dL_db1 = (2/m)*np.sum(dL_dZ1, axis = 0, keepdims = True)

# Update the weights and biases using gradient descent
self.W1 -= alpha*dL_dW1
self.b1 -= alpha*dL_db1
self.W2 -= alpha*dL_dW2
self.b2 -= alpha*dL_db2

# Update loss
self.MSE_loss(inputs, outputs)
```

Backpropagation in our model

Effect of the backpropagation

- Every time we call the backward() method, our model will update its parameters,
- Eventually becoming better and better at the task in question.
- This can be seen by looking at the loss values decreasing after each iteration of the backward() method.

```
# Define neural network structure
n_x = 2
n_h = 4
n_y = 1
np.random.seed(967)
shallow_neural_net = ShallowNeuralNet(n_x, n_h, n_y)
loss = shallow_neural_net.MSE_loss(inputs, outputs)
print(shallow_neural_net.loss)
```

```
13.1869714693353
```

```
shallow_neural_net.backward(inputs, outputs, 1e-5)
print(shallow_neural_net.loss)
```

```
8.576174190387835
```

```
shallow_neural_net.backward(inputs, outputs, 1e-5)
print(shallow_neural_net.loss)
```

```
5.818528741557871
```

```
shallow_neural_net.backward(inputs, outputs, 1e-5)
print(shallow_neural_net.loss)
```

```
4.14790610131927
```

Trainer function for our model

Train function implementation

- Iterate the backward() method for a given number of iteration $N_{\max}$.
- Display new loss values and append them in a list for display later.
- Also implemented an early stopping, which will break for loop if no change in parameters during an iteration of backward().

```python
# Our trainer function
def train(shallow_neural_net, inputs, outputs, N_max = 1000, \
          alpha = 1e-5, delta = 1e-5, display = True):
    # List of losses, starts with the current loss
    losses_list = [shallow_neural_net.loss]
    # Repeat iterations
    for iteration_number in range(1, N_max + 1):
        # Backpropagate
        shallow_neural_net.backward(inputs, outputs, alpha)
        new_loss = shallow_neural_net.loss
        # Update losses list
        losses_list.append(new_loss)
        # Display
        if(display):
            print("Iteration {} - Loss = {}".format(iteration_number, new_loss))
        # Check for delta value and early stop criterion
        difference = abs(losses_list[-1] - losses_list[-2])
        if(difference < delta):
            if(display):
                print("Stopping early - loss evolution was less than delta.")
            break
    else:
        # Else on for loop will execute if break did not trigger
        if(display):
            print("Stopping - Maximal number of iterations reached.")
    return losses_list
```

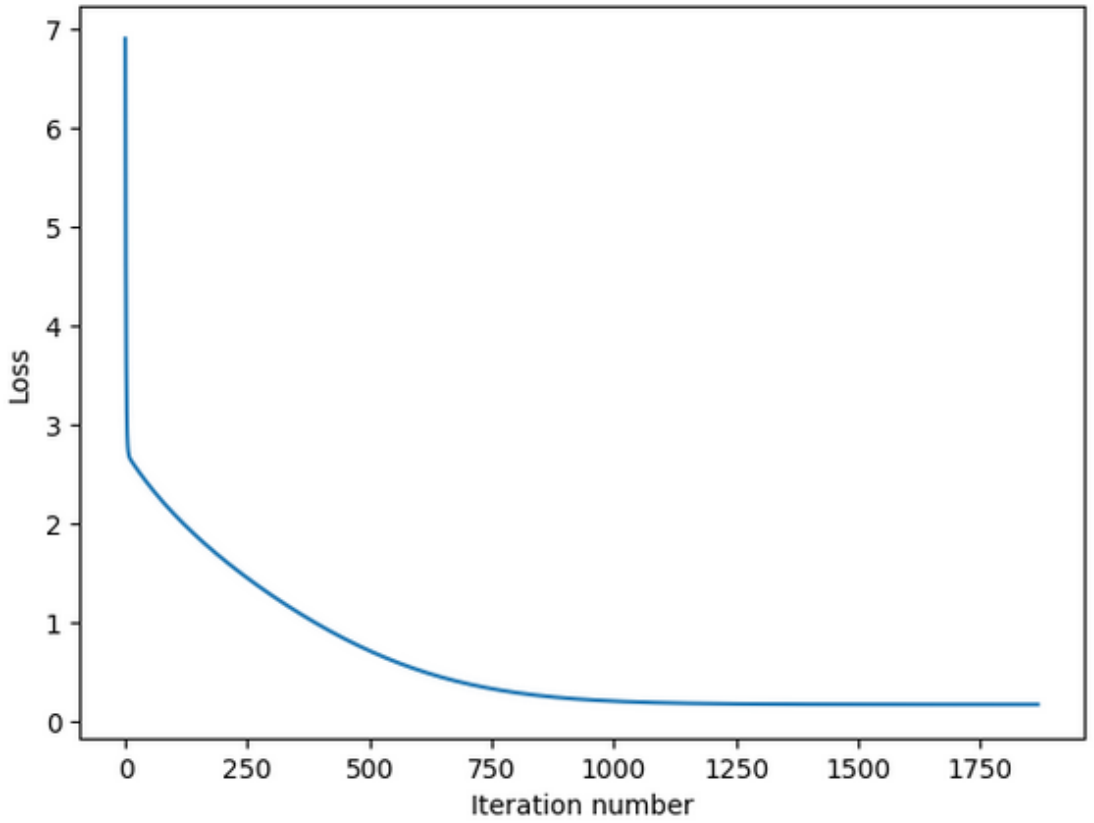
Trainer function for our model

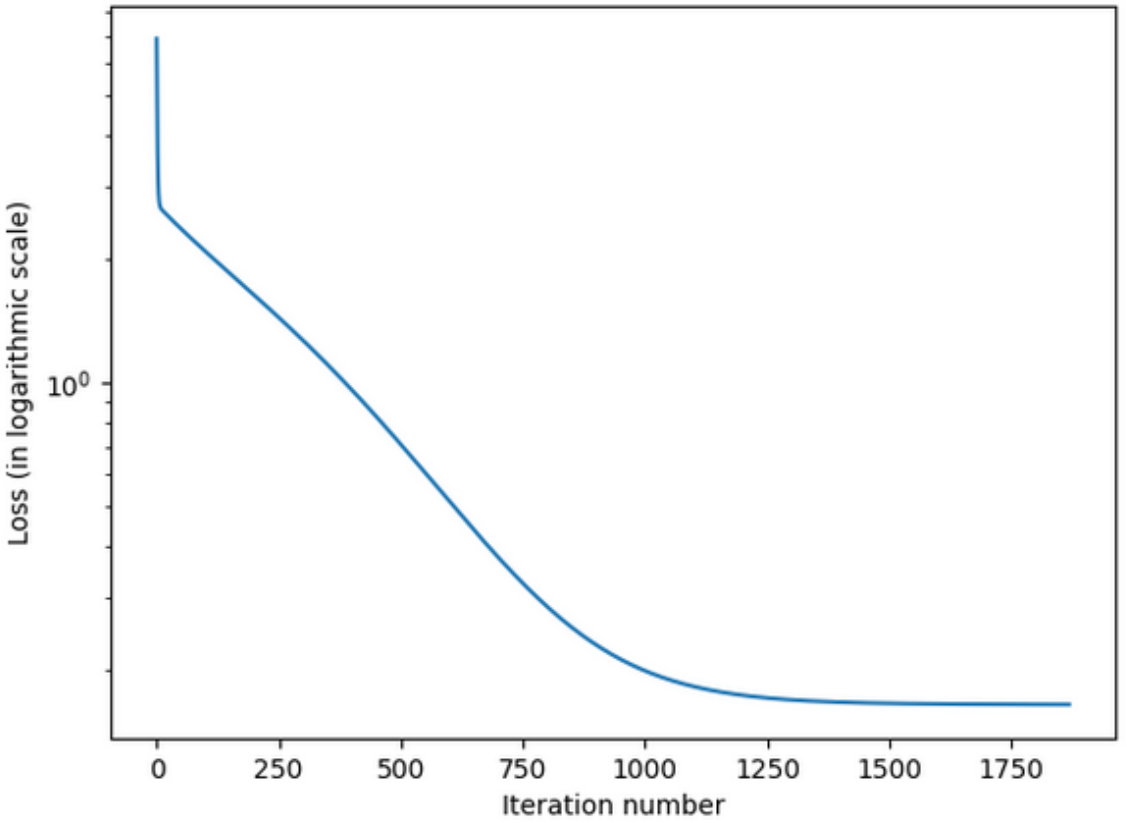
```
1 losses_list = train(shallow_neural_net, inputs, outputs, N_max = 10000, alpha = 1e-5, delta = 1e-6, display = True)
```

```
Iteration 1 - Loss = 4.639845655363769
Iteration 2 - Loss = 3.6055739235388646
Iteration 3 - Loss = 3.1219544752264055
Iteration 4 - Loss = 2.891370725402625
Iteration 5 - Loss = 2.778663798354082
Iteration 6 - Loss = 2.721333641596964
Iteration 7 - Loss = 2.6901646643755135
Iteration 8 - Loss = 2.671407441095785
Iteration 9 - Loss = 2.6585626629544303
Iteration 10 - Loss = 2.648548914291606
Iteration 11 - Loss = 2.639902127093024
Iteration 12 - Loss = 2.6319255476288577
Iteration 13 - Loss = 2.6242871305644138
Iteration 14 - Loss = 2.6168284145839156
Iteration 15 - Loss = 2.60947364178999
Iteration 16 - Loss = 2.6021864900028118
Iteration 17 - Loss = 2.5949494536811937
Iteration 18 - Loss = 2.58775401064343
Iteration 19 - Loss = 2.5805959297914653
```

Trainer function for our model

The curves below are called **performance/training curves** and show how the training went.





Finally, add the trainer function and the performance curves function as methods to our class.

```python
    def train(self, inputs, outputs, N_max = 1000, alpha = 1e-5, delta = 1e-5, display = True):
        # List of losses, starts with the current loss
        self.losses_list = [self.loss]
        # Repeat iterations
        for iteration_number in range(1, N_max + 1):
            # Backpropagate
            self.backward(inputs, outputs, alpha)
            new_loss = self.loss
            # Update losses list
            self.losses_list.append(new_loss)
            # Display
            if(display):
                print("Iteration {} - Loss = {}".format(iteration_number, new_loss))
            # Check for delta value and early stop criterion
            difference = abs(self.losses_list[-1] - self.losses_list[-2])
            if(difference < delta):
                if(display):
                    print("Stopping early - loss evolution was less than delta.")
                break
        else:
            # Else on for loop will execute if break did not trigger
            if(display):
                print("Stopping - Maximal number of iterations reached.")

    def show_losses_over_training(self):
        # Initialize matplotlib
        fig, axs = plt.subplots(1, 2, figsize = (15, 5))
        axs[0].plot(list(range(len(self.losses_list))), self.losses_list)
        axs[0].set_xlabel("Iteration number")
        axs[0].set_ylabel("Loss")
        axs[1].plot(list(range(len(self.losses_list))), self.losses_list)
        axs[1].set_xlabel("Iteration number")
        axs[1].set_ylabel("Loss (in logarithmic scale)")
        axs[1].set_yscale("log")
        # Display
        plt.show()
```

Conclusion (Week 1)

- Reminders of Machine Learning
- Linear
- Using a normal equation to train a linear regression
- Gradient descent as a training procedure for linear regression
- Polynomial Regression
- Regularization in Ridge/Lasso Regression
- Train-test split
- Overfitting and underfitting
- Generalization
- Sigmoid and Logistic functions
- From linear regression to logistic regression
- Using logistic regression for binary classification

Conclusion (Week 1)

- Implementing a shallow neural network in Numpy
- Forward propagation method, to formulate predictions
- The backpropagation mechanism, as the gradient descent on Neural Network
- Backward method for training and trainer functions to iterate backward iterations
- Performance/training curves

Next week?

- Initializations to break symmetries
- Exploding and vanishing gradients
- Activation functions and non-linearities in Neural Networks
- Advanced optimizers
- Validation sets, early stopping, saver and loader functions

Learn more about these topics

Out of class, supporting papers, for those of you who are curious.

- [Géron2019] A. Géron , “How Neural Networks Work”, 2019.
- [Yamins2016] Yamins et al., “Deep neural networks are robust computational models of the human visual system”, 2016.
- [Kriegeskorte2013] Kriegeskorte and Kievit, “Neural Network Models of the Human Brain”, 2013.
- [Rumelhart1986] D.E. **Rumelhart**, G. **Hinton**, Williams, “Learning representations by back-propagating errors“, 1986 https://www.nature.com/articles/323533a0

Learn more about these topics

Tracking important names (Track their works and follow them on Scholar, Twitter, or whatever works for you!)

- **Geoffrey Hinton: Professor** at **University of Toronto**, **one of the three Godfathers of Deep Learning** and **2018 Turing Award** winner (highest distinction in Computer Science).
  https://www.cs.toronto.edu/~hinton/
  https://scholar.google.co.uk/citations?user=JicYPdAAAAAJ&hl=en
- **David Rumelhart: Former professor** at **University of California**, credited for inventing **backpropagation along** with **Hinton**. Passed away in 2011.
  https://www.nytimes.com/2011/03/19/health/19rumelhart.html

Twitter, the theater for AI/DL drama and announcements

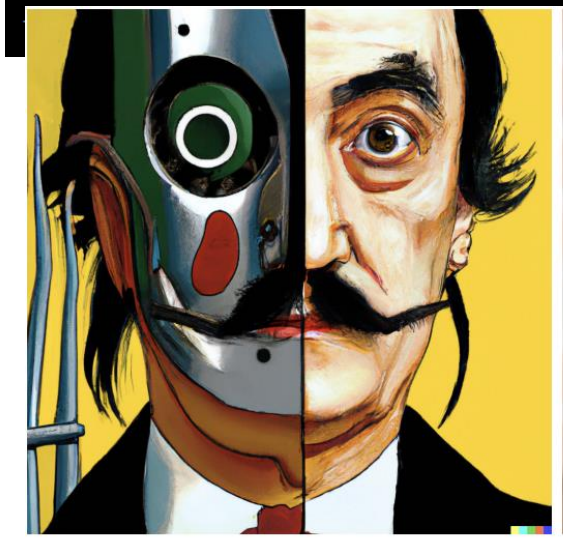
OpenAI ✓
@OpenAI

Here's a look at what DALL·E 2 can do. 👀🧵👇

Want to see more? Follow along on Instagram:
instagram.com/openaidalle/

Traduire le Tweet

Announcing Dall-E 2 and its future release



vibrant portrait painting of Salvador Dalí with a robotic half face

a shiba inu wearing a beret and black turtleneck

Gary Marcus @GaryMarcus · 8 mai
You obviously never read a single one of the papers of mine that you have been criticizing.

One thing for you to disagree, another to show a lack of awareness of what the person you disagree with is saying.

2 3 15

NYU Professor

Yann LeCun
@ylecun

En réponse à @GaryMarcus et @Tesla

You obviously never engineered nor built a working AI system.
Arguments in favor "hybrid systems" are non-falsifiable strawmen, whose only purpose to allow the author to declare victory regardless of the outcome.
As I said, this debate is vacuous.

Traduire le Tweet

12:18 AM · 9 mai 2022 · Twitter Web App

2 Retweets **9** Tweets cités **173** J'aime

NYU Professor and Chief of AI at Meta (Facebook)

Learn more about these topics

Some extra (easy) reading and videos for those of you who are curious.

- [Quanta2021] “Artificial Neural Nets Finally Yield Clues to How Brains Learn“, 2021. https://www.quantamagazine.org/artificial-neural-nets-finally-yield-clues-to-how-brains-learn-20210218/

- [MITNews2022] “Study urges caution when comparing neural networks to the brain”, 2022. https://news.mit.edu/2022/neural-networks-brain-function-1102

Learn more about these topics

Some extra (easy) reading and videos for those of you who are curious.

- [Medium2020] “Who Invented Backpropagation? Hinton Says He Didn’t, but His Work Made It Popular“, 2020. https://medium.com/syncedreview/who-invented-backpropagation-hinton-says-he-didnt-but-his-work-made-it-popular-e0854504d6d1/

- [MITNews2022] “Study urges caution when comparing neural networks to the brain”, 2022. https://news.mit.edu/2022/neural-networks-brain-function-1102