# Week 2 Class Notes

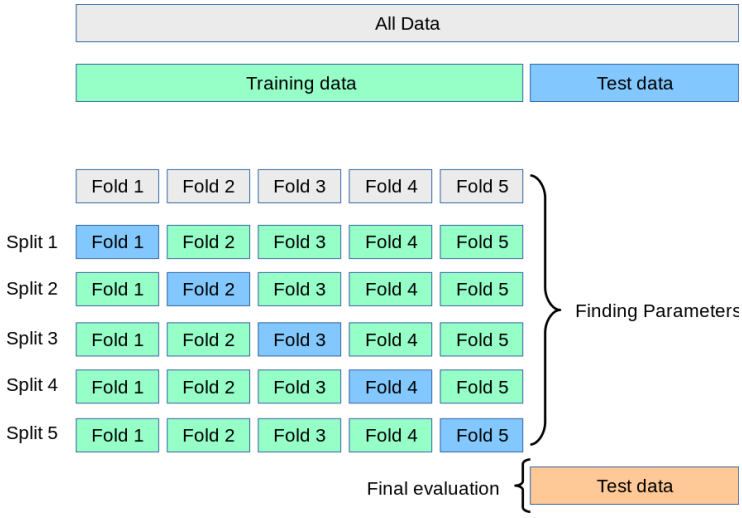
Objective: This document includes class notes: **summaries** of everything we learned, as well as **questions and answers** during the sessions, as well as any **resources** and links shared by students or otherwise. Also **assignments** and due dates. Other stuff might be included as well.

---

**Submissions Link:**
https://docs.google.com/forms/d/e/1FAIpQLScCZb7DXs2zcc6StZpurY2xLppEb0mjhUg6Imef9wGsj8rcbA/viewform?usp=sf_link

# Session 4

Getting Started in scikit-learn:
https://scikit-learn.org/stable/getting_started.html




K-fold Cross-validation:
https://scikit-learn.org/stable/modules/cross_validation.html#cross-validation
Note, Parameters in the figure really means: Hyper-parameters.

Random Seed. We tried the same random_seed on Windows, Linux, and MacOS machines, and ran a random event, and got exactly the same number back.

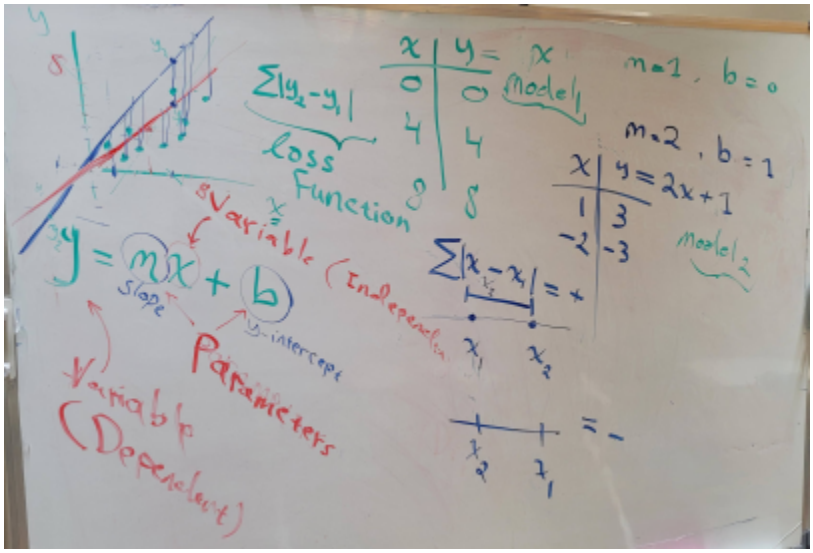
**Exercises (2 hours):**

1. Salary Prediction Dataset:
   https://www.kaggle.com/datasets/rkiattisak/salaly-prediction-for-beginer
2. Classification on Iris Dataset

**Resources**:

- **Regression:** Linear Regression (y is continuous): https://scikit-learn.org/stable/auto_examples/linear_model/plot_ols.html#sphx-glr-auto-examples-linear-model-plot-ols-py
- **Classification:** Logistic Regression (y is discrete): https://scikit-learn.org/stable/auto_examples/linear_model/plot_iris_logistic.html
- **Pipelines**: https://scikit-learn.org/stable/auto_examples/compose/plot_column_transformer_mixed_types.html#sphx-glr-auto-examples-compose-plot-column-transformer-mixed-types-py

# Session 3




**Linear Regression**

Line equation is: $y = mx + b$

- $y$ is the dependent variable
- $x$ is the independent variable
- $m$ is the slope
    - changing it changes the angle of the line
    - positive slope means you move up if you move right
    - negative slope means you move down if you move right
- $b$ is the y-intercept
    - it determines where the line intersects the y-axis
    - increase it to lift the line up (without changing its angle)
    - decrease it to lower the line down

Both $m, b$ are **Parameters** of the **model** (the line); meaning, to get another model, we simply change the parameters.

In the slide, we have:

- model A: $m = 1,\ b = 0$; which is the red line
- model B: $m = 2,\ b = 1$; which is the blue line

Sorry the drawing was not accurate; the blue should have been lifted up by 1 unit above the red line.

Note, in some other notation, the equation $y = mx + b$ might use other symbols such as:

- $f(x) = ax + b$
- $\hat{h}(x) = \theta_0 + \theta_1 x$

But, they are all just lines (models).

A **Machine Learning Algorithm** simply nudges (slightly changes) the parameters (increase/decrease) and calculates how bad the model becomes.

- If changing the parameters results in a line further away from the data, the loss increases
- if it gets closer to the data, the loss decreases

Sliders using Desmos: https://www.desmos.com/calculator/rih3fgne7h

**Loss / Cost Function**

We can decide whether model A or B is better than the other, by how close the line is to the (x,y) pairs; the data. We can quantify this by the distance. That is:

$$error = y_{true} - y_{pred}$$

Where both $y_{true}$ and $y_{pred}$ correspond to the same $x$ on the x-axis:

- $y_{true}$ is the point
- $y_{pred}$ is on the line vertical to the point

Let's add the absolute function to make the error positive all the time:

$$error = |y_{true} - y_{pred}|$$

Let's make sure the error increases more if the point is further away, by squaring it:

$$error^2 = |y_{true} - y_{pred}|^2 = (y_{true} - y_{pred})^2$$

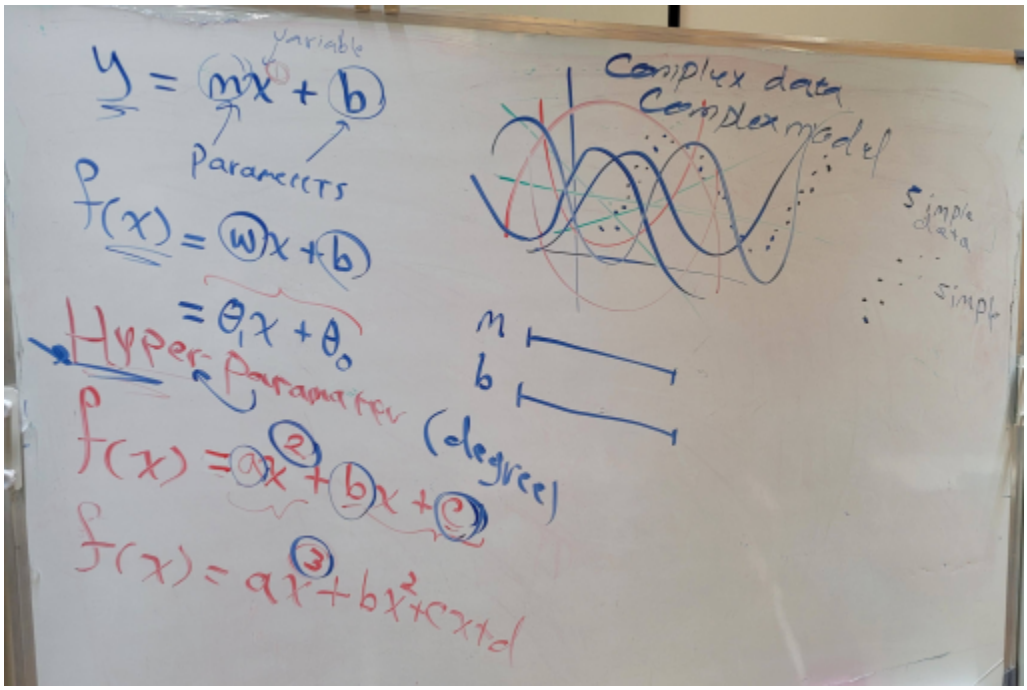
Let's now account for all the data points by summation and average over their number, to eliminate the effect of their number on the calculation. Let's call that, the **M**ean **S**quared **E**rror:

$$MSE = \frac{1}{m}\sum(y_{true} - y_{pred})^2$$

In Scikit-learn, the metric used is: `mean_squared_error(y_true, y_pred)`.

Example, if the target variable y is price. Then, for each point x (size of the house), we have a corresponding point y (price). Now, we give the model a point x, and ask it to predict the y, which we call `y_pred`, then compare it to the price we know: `y_true`.

**Hyper-parameters**:




Hyper-parameters of **Linear Regression** is the **Degree** of the **Polynomial**. The higher the degree:

1. the more curvy and complex the linear model becomes; and
2. the more parameters we have to fit

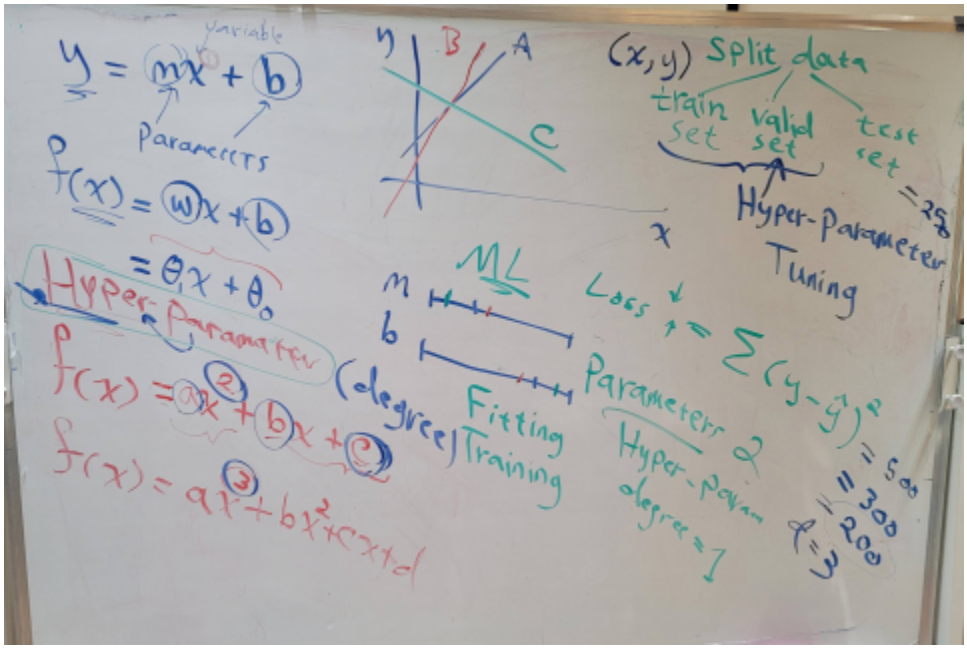
Example:

$$f(x) = \theta_5 x^5 + \theta_4 x^4 + \theta_3 x^3 + \theta_2 x^2 + \theta_1 x + \theta_0$$

When we say: **Training / Fitting**; we mean: closing the gap between the model curve and the data points. This, as we have indicated before, is done, usually, by adjusting the parameters slightly, in an iterative loop. Imagine having sliders for each parameter $\theta_i$ which the ML Algorithm moves left and right.

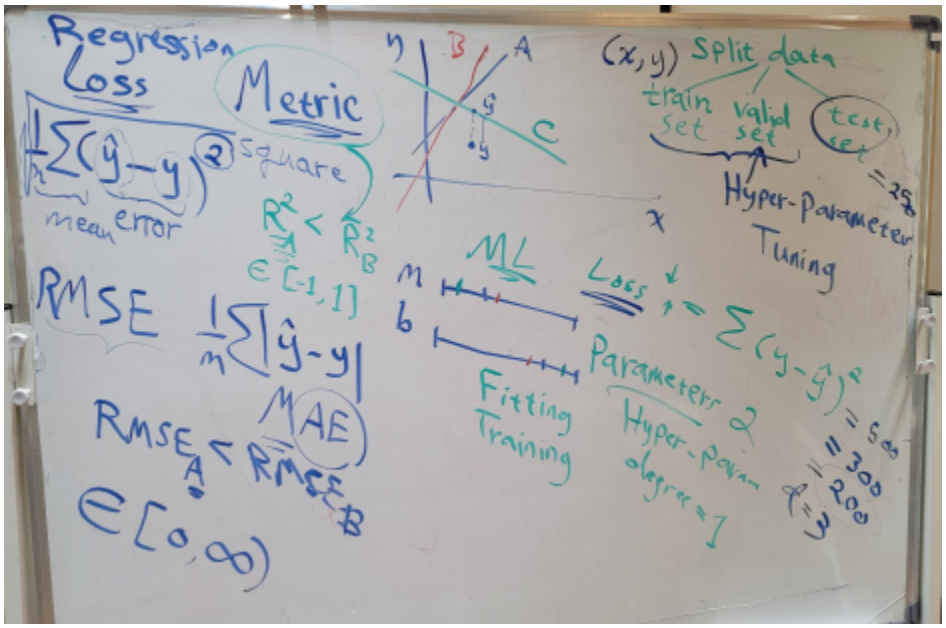
Note that each model **parameter** $\theta_i$ **is initialized at random**.

Complex data requires complex models.
Simple data requires simple models.
Using complex models for simple data results in **Overfitting**.
Using simple models for complex data results in **Underfitting**.




A **Dataset** needs to be **Split** into three:

1. **Training set**: used to fit model parameters
2. **Validation set**: used to search for best hyper-parameters
3. **Testing set**: used as a final test for the model



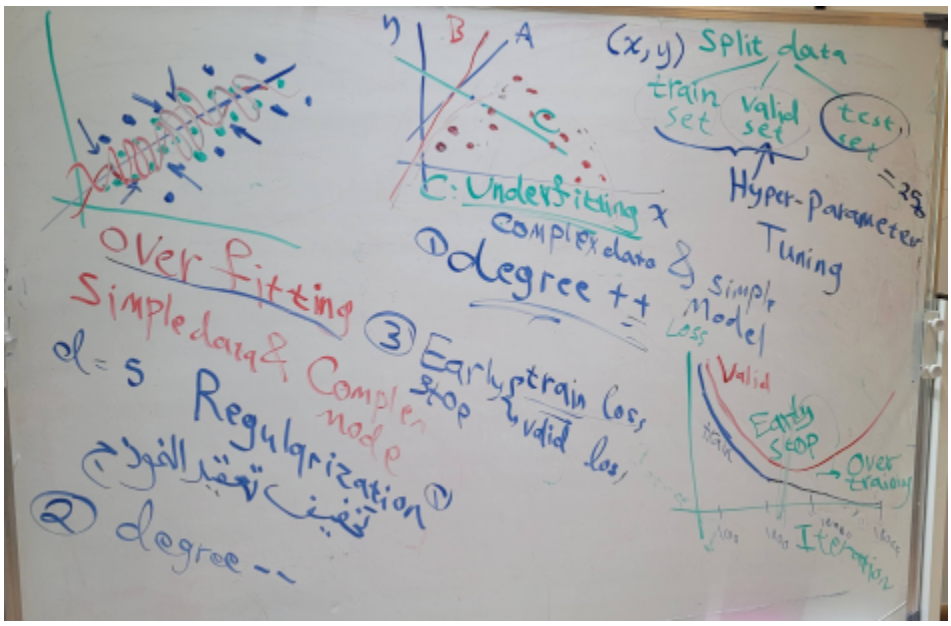

Usually the split is 80% for train and validation sets. And 20% for the test set.

**Hyper-parameter tuning**.

1. Select a set of hyper-parameters (e.g., degree=2)
    a. **RandomSearch**CV (pick at random from a range of values=[1 to 100])
    b. **GridSearch**CV (try each and every possible value within a set range: [1 to 100])
2. Fit the model on the training set
3. Evaluate the model on the validation set
4. Repeat step 1; continue this **loop for 5 times**; then break:

Then:

5. Choose the hyper-parameters that resulted in the highest score / lowest loss
6. Combine validation set to the training set
7. Fit the model on this combined set
8. Evaluate the model on the test set

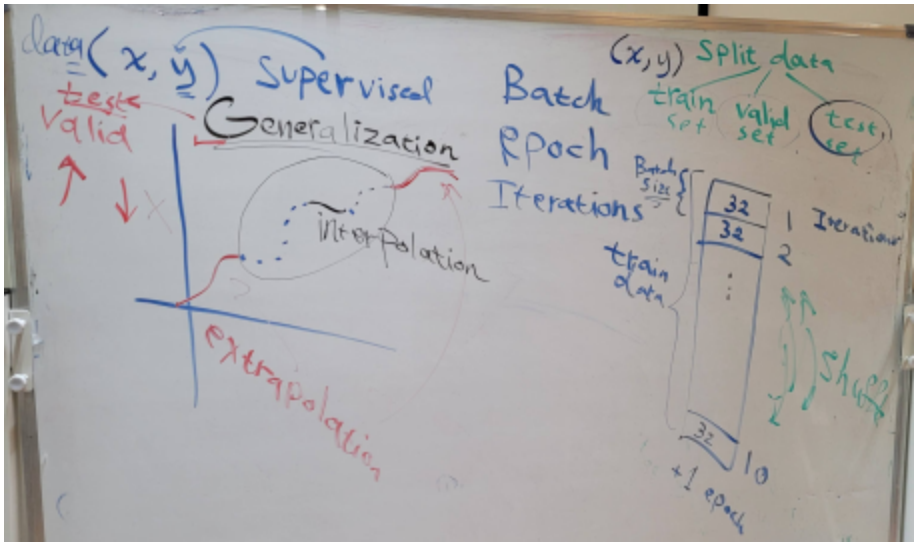



**Solutions to over-fitting**:

- **Get more data** (e.g., 100 examples → 1,000 examples)
- **Simplify the model** (lower the degree of the polynomial; e.g., d=5 → d=3)
- **Regularization** (L1 or L2): makes the model less curvy without reducing the degree of the polynomial; rather, by shrinking the parameters of the model.
- **Early Stopping:** don't over train the model. If you stop after 100 epochs; try stopping at 90 epochs or lower. (least preferred)

**Solutions to under-fitting**:

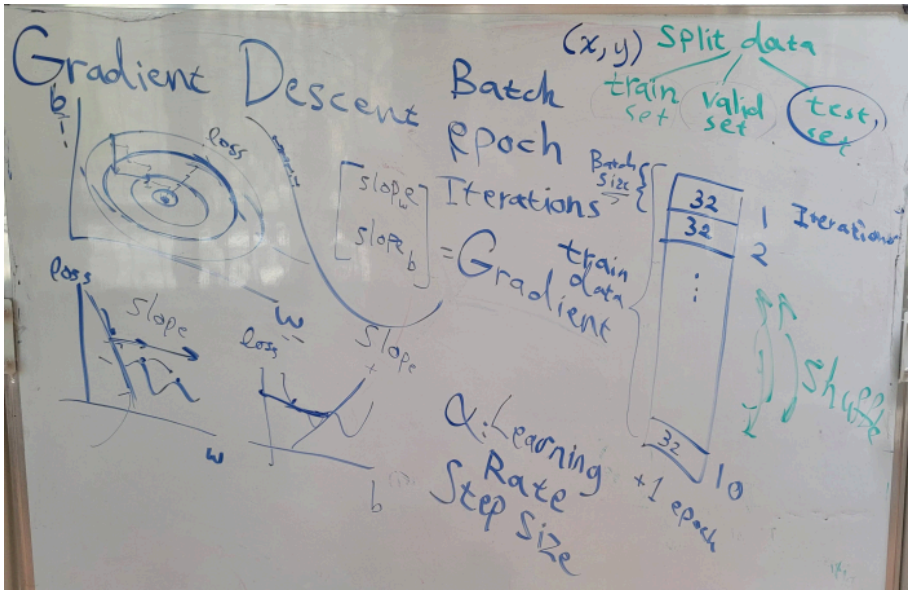
- **Complexify the model** (higher degree of the polynomial; e.g., d=3 → d=5)
- Decrease regularization




**Interpolation** is predicting patterns within the range of the data.
**Extrapolation** is predicting patterns beyond the range of the data.

A model is said to **Generalize** if it does well in a testing set (something it hasn't seen), and would be considered too good if this testing set requires extrapolation rather than just interpolation.

**Batch**: a set of examples fed together to a model in 1 **Iteration**.
**Batch Size**: how many examples per iteration are fed to the model. (default = 32)
One **Epoch** is a complete run over the entire dataset (i.e., the model sees all batches)
After each epoch, we **Shuffle** the batches before starting the next epoch. This also helps the model not to overfit by depending on the order in which the data occurs.




**Gradient Descent.**
A vector of slopes, for each variable, is called a **Gradient**.
If you go down the slope, you are **Descending**.

To minimize the loss, we want to go down the gradient with respect to the loss function.

*Gradient Descent* means that the parameter adjustment is always down the slope of each variable (negative to it); in order to lower the loss. In contrast, going up the slopes would be an Ascent.

**Learning Rate / Step Size**.
This hyper-parameter, often noted as $\alpha$ (alpha) or $\eta$ (eta), is how far we step down the slope for each variable when descending. This affects how long it takes to reach the lowest point, as well as if, at all, we would reach it or just keep jumping around (when the step is too big).

$$\theta := \theta - \alpha \nabla loss$$

Notes about the above **Update Rule**:

- The ":=" is assignment
- The minus "-" is needed to go down
- The step size $\alpha$ (alpha) dictates how much we go down
- The $\nabla loss$ is the gradient (slopes)

**Tutorials (Notebooks)**:

- Getting Started in scikit-learn: https://scikit-learn.org/stable/getting_started.html

- Linear Regression Example: https://scikit-learn.org/stable/auto_examples/linear_model/plot_ols.html#sphx-glr-auto-examples-linear-model-plot-ols-py
- Intermediate Machine Learning (Kaggle): https://www.kaggle.com/learn/intermediate-machine-learning

**Q&A:**

**Q: What batch size is best?**
A: 32 is a good default for most cases. However, some people suggest picking a number that fills the memory and not worry about what specific number it is.

**Q: If metrics show the same values for 2 different models: a Decision Tree and a Random Forest, which one should I pick?**
A: Unlike Random Forests (which are an ensemble of decision trees), a single Decision Tree is an interpretable model; i.e., I can tell which feature affected the prediction.

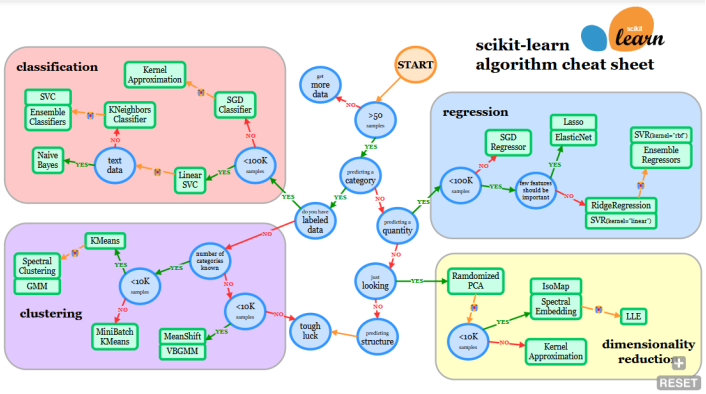
**Q: What is `random_state` in `train_test_split(X, y, random_state=0)`?**
A: Any random event after you set a seed of, say 14, will always produce the same random numbers after it. If two computers use the same seed, they will see the same random numbers. This is known as pseudo-randomness, contrary to true randomness.

The random state hyperparameter in the `train_test_split()` function **controls the shuffling process**.

- `With random_state=None`, we get different train and test sets across different executions and the shuffling process is out of control.
- `With random_state=0`, we get the same train and test sets across different executions.

**Resources**:

- **Coursera Machine Learning Specialization.**
- Scikit-learn ML Map: https://scikit-learn.org/stable/machine_learning_map.html#ml-map



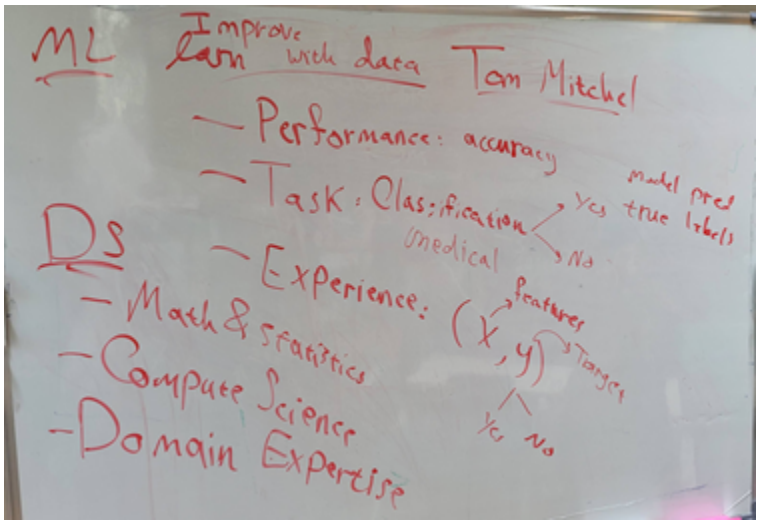
# Session 2

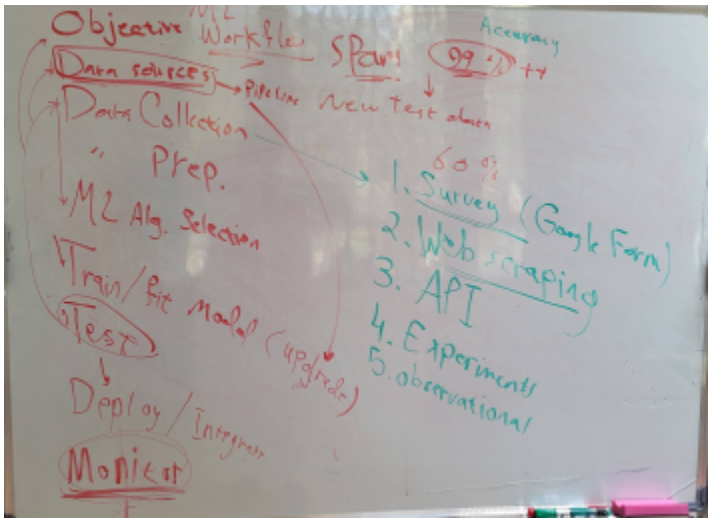
**Machine Learning**: A computer program is said to:

1. learn from experience **E**
2. with respect to some class of tasks **T** and
3. performance measure **P**

if its performance at tasks in **T**, as measured by **P**, improves with experience **E**.

**E**xperience: is data.
**T**ask: can be many things; including: Predicting House Price, or classifying an Image.
**P**erformance: can be (MSE or R-squared for regression) and (Accuracy for classification) and others, depending on the specific task.



**Web Scraping (Notebook):**
https://github.com/HassanAlgoz/data-analytics/blob/main/lecture/L08_web_scraping.ipynb

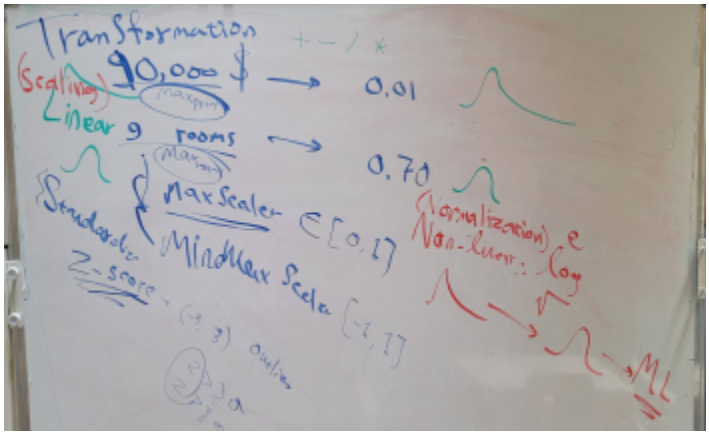
403 Access Denied (Unauthorized):

- `headers={'User-Agent': 'Ahmad Bot'}`

Static HTML page
Dynamic (Javascript) → Selenium.

**Regexp 101 (Regular Expressions)**: https://regex101.com/

**Regular Expressions (Python)**: https://www.w3schools.com/python/python_regex.asp




**Feature Engineering (Notebook)**:
https://github.com/HassanAlgoz/data-analytics/blob/main/techniques/feature_engineering.ipynb

**Penguins Exercise (Notebook)**:
https://github.com/HassanAlgoz/data-analytics/blob/main/scenario/penguins_exercise.ipynb

# Session 1

*Data science* is the field of study that combines:

1. domain expertise
2. programming skills
3. mathematics and statistics

to extract meaningful insights from data.

Domain experts would tell you what kind of features to collect, how to collect them, and what purpose they serve.

Before learning from data, programmers used to write algorithms to capture a long list of hand-crafted rules made by experts in their respective fields. Examples:

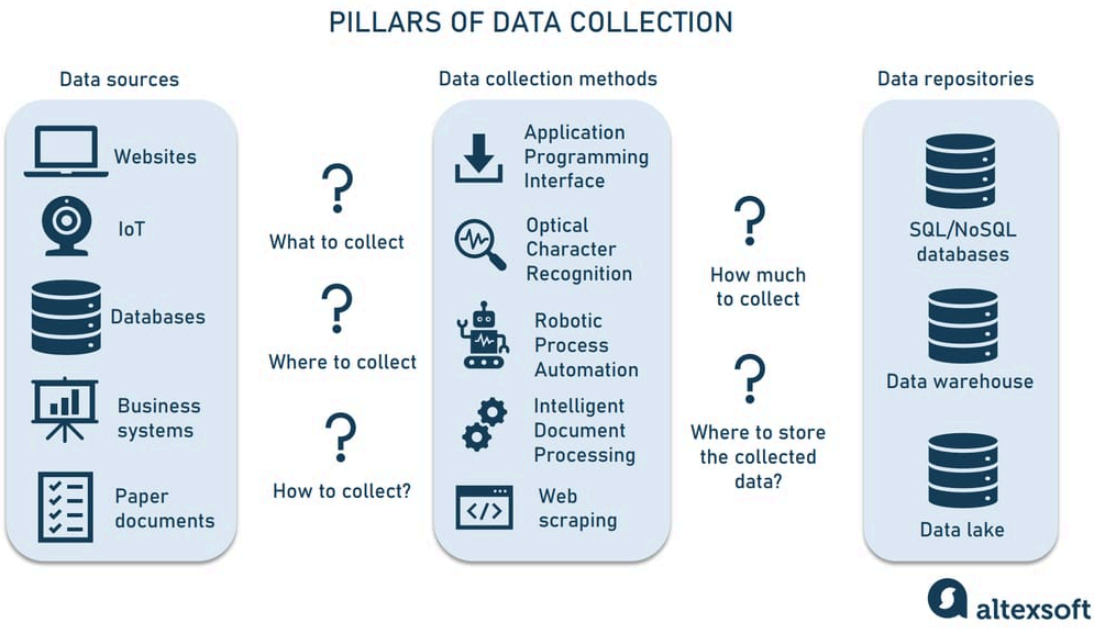
- Medical diagnosis: Systems would use a series of if-then rules based on symptoms, medical history, and test results to suggest possible diagnoses
- Robotics: Robot control systems were programmed with specific movements and actions based on sensor inputs

Problem: systems were often inflexible, and as problems became more complex, the number of rules grew exponentially, making maintenance difficult. Furthermore, expert knowledge wasn't enough to capture all possible scenarios.

*Machine Learning* automated algorithm writing, which is especially useful when we have:

- **too many patterns**: too many cases for the algorithm to cover
- **too complex patterns**: impossible to write an algorithm for
- **changing patterns**: too costly, or impossible, to keep rewriting the algorithm

**Pillars of Data Collection**




Source: https://www.altexsoft.com/blog/data-collection-machine-learning/

**5 Characteristics of Data Quality**

1. **Accuracy:** Is the information correct in every detail?
2. **Completeness:** How comprehensive is the information?
3. **Reliability:** Does the information contradict other trusted resources?

4. **Relevance:** Do you really need this information?
5. **Timeliness:** How up- to-date is the information? Can it be used for real-time reporting?

Source: https://www.precisely.com/blog/data-quality/5-characteristics-of-data-quality

**Words-per Minute (notebook):**
https://github.com/HassanAlgoz/data-analytics/blob/main/scenario/wpm.ipynb

**California Housing Dataset (notebook):**
https://github.com/HassanAlgoz/data-analytics/blob/main/scenario/Ex03_eda_housing.ipynb