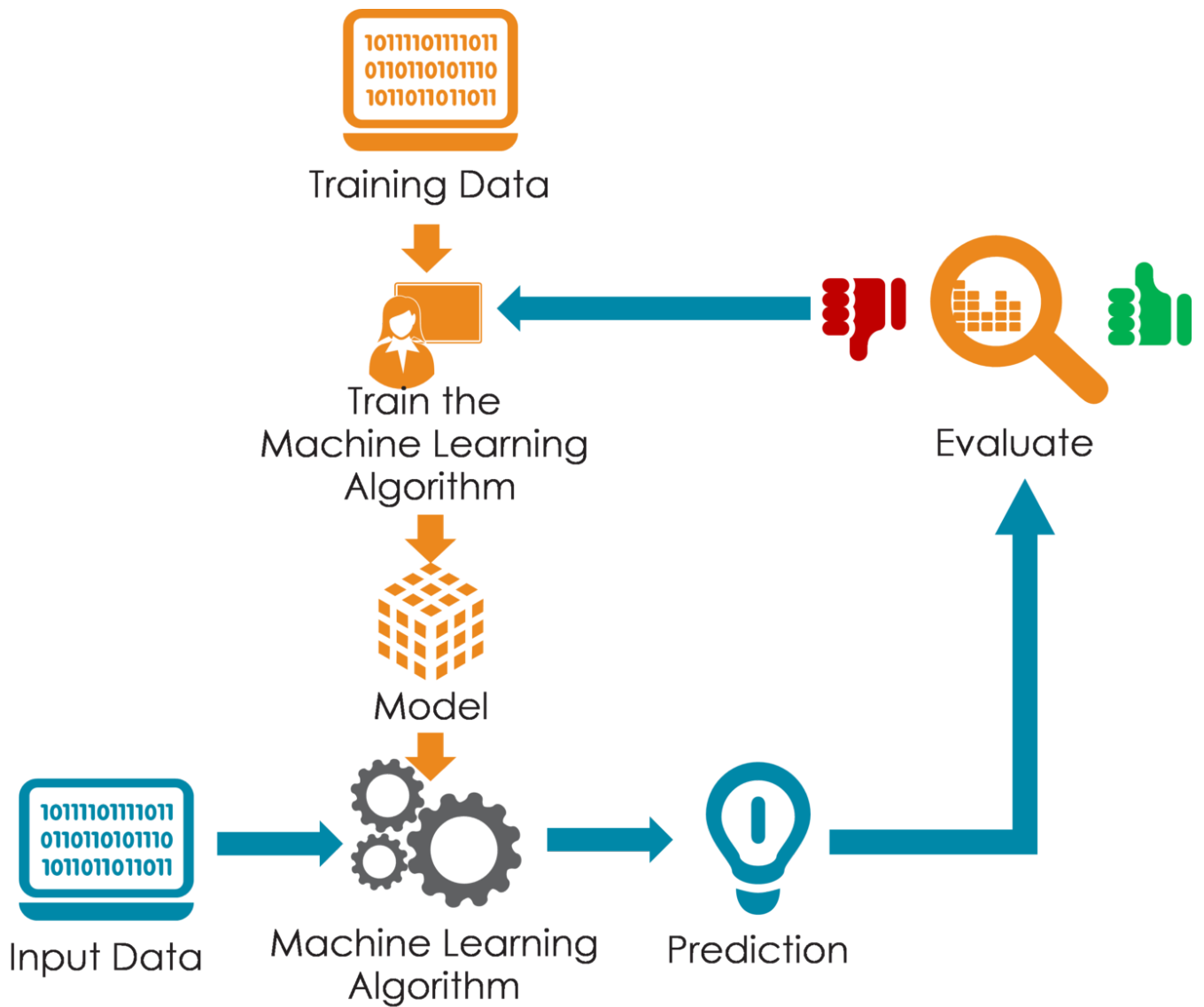


# Performance Evaluation in Machine Learning

## 1. Introduction

Just like chess players improve their technique by watching hundreds of games from top players, computers can be able to perform certain tasks by “looking” at data. These computers are sometimes called “**machines**” and this data observation step is also known as “**learning**”. Simple, isn’t it? So let’s define “**Machine Learning**” (ML) more formally, as stated by Tom Mitchell. A program is said to learn from experience  $E$  with respect to some task  $T$  and some performance measure  $P$ , if its performance on  $T$ , as measured by  $P$ , improves with experience  $E$ . Let’s get more **-h** and provide an example. Say that  $T$  is the process of playing chess,  $E$  is a set of chess matches on which the algorithm was trained and  $P$  is the probability for the program to win the next game of chess.

What it’s truly important however, is that machine learning algorithms, unlike traditional ones, are not required to be pre-programmed explicitly to solve a particular task. Just like in the previous example, they **learn** from the data fed into them and afterwards, are able to **predict** results for new scenarios. As you might expect, this isn’t just black magic. The majority of ML algorithms are based on some maths and statistics, but also on a fair amount of engineering. 😎



[Image Source](#)

🚫🚫🚫 Machine learning algorithms learn from data and they **don't** have to be programmed explicitly!

## ▼ 2. Supervised Learning vs Unsupervised Learning

Generally, in machine learning, there are two types of tasks: **supervised** and **unsupervised**.

### A. Supervised Learning

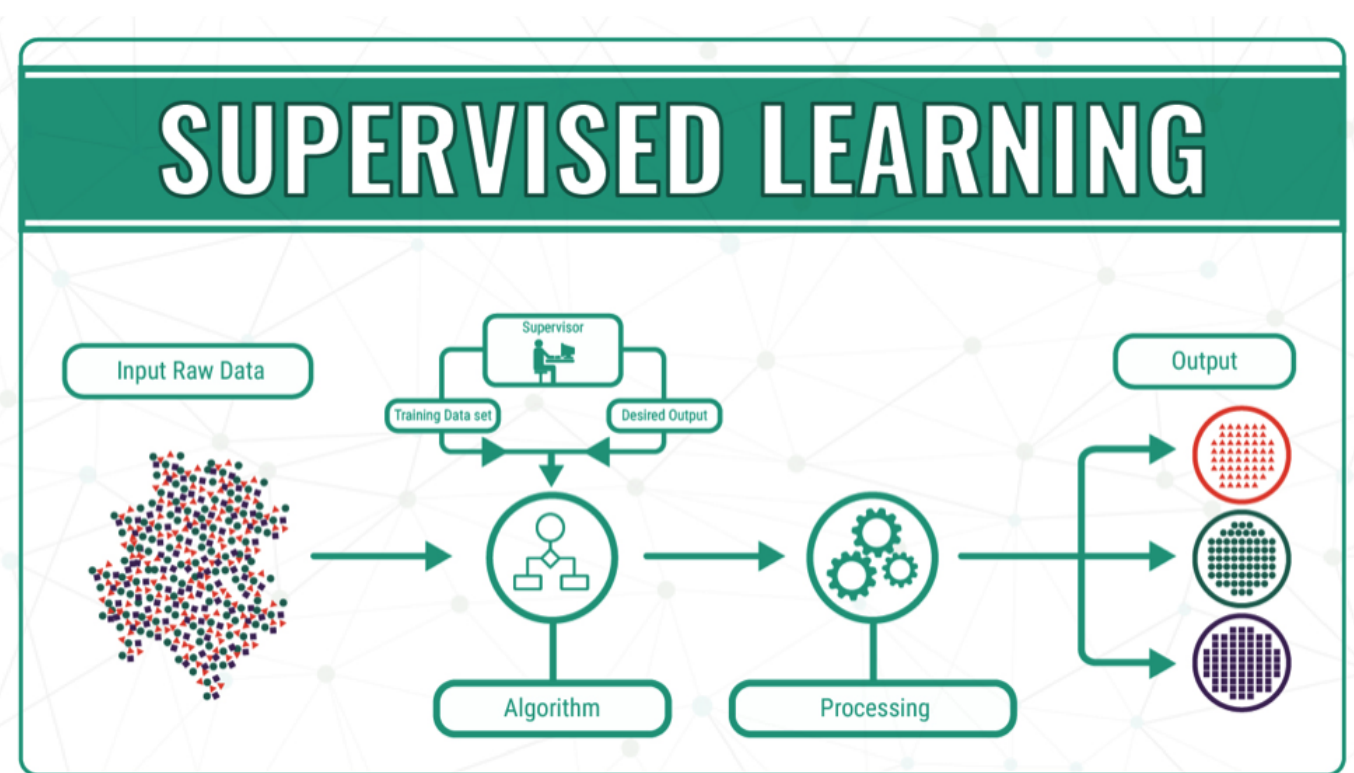
We refer to the former when we have prior knowledge of what the output of the system should look like. More formally, after going through lots of  $(X, y)$  pairs, the model should be able to determine that **mapping function**  $\hat{h}$  which approximates  $f(X) = y$  as accurately as possible.

In machine learning, the entire set of  $(X, y)$  pairs is often called **corpus** or **dataset** and an algorithm that learned from this corpus is sometimes called a **trained model** or simply a **model**.

For example, let's say that you want to build a smart car selling platform and you want to provide sellers with the ability to know what should be a "fair price" based on the **specs** and **age** of their cars. For that, you would need to provide your model with a set of tuples that might look like this:

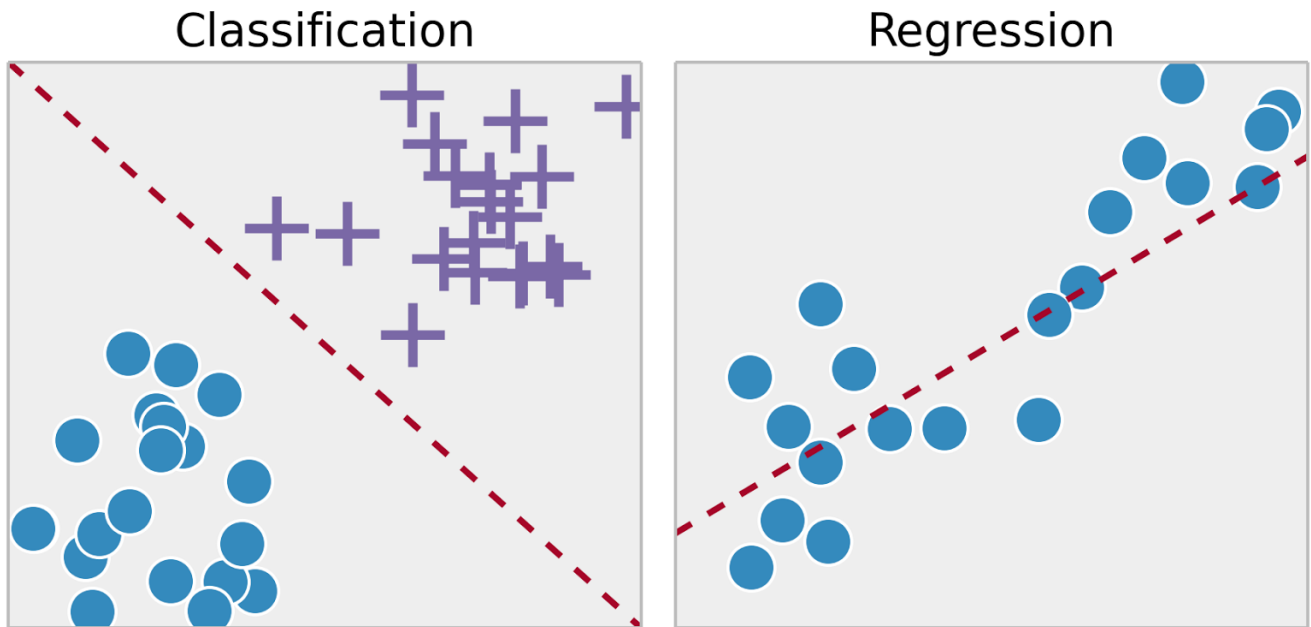
(bhp, fuel\_type, displacement, weight, torque, suspension, car\_age, car\_price)

In this case, **X** is formed from the tuple slices (bhp, fuel\_type, displacement, weight, torque, suspension, car\_age) and each slice component is called a **feature**. Likewise, **y** is represented by the car\_price and it's usually called **label** or **ground truth**. After learning from the provided corpus, a process that is commonly called **model fitting**, our model should be able to predict an approximation of the car price  $\hat{y}$  for new ages and specs **X**.



In the dataset of each **supervised** machine **learning** task, there will always be a **label y** that can be associated with a set of **features X**!

Because a **numerical** or **continuous** output is expected as a result, the aforementioned scenario is an example of a **regression task**. If  $y$  was a label that can take the values "cheap", "fair" or "expensive", then that would be called a **classification task** and  $y$  can also be referred to as a **class**.



[Image Source](#)

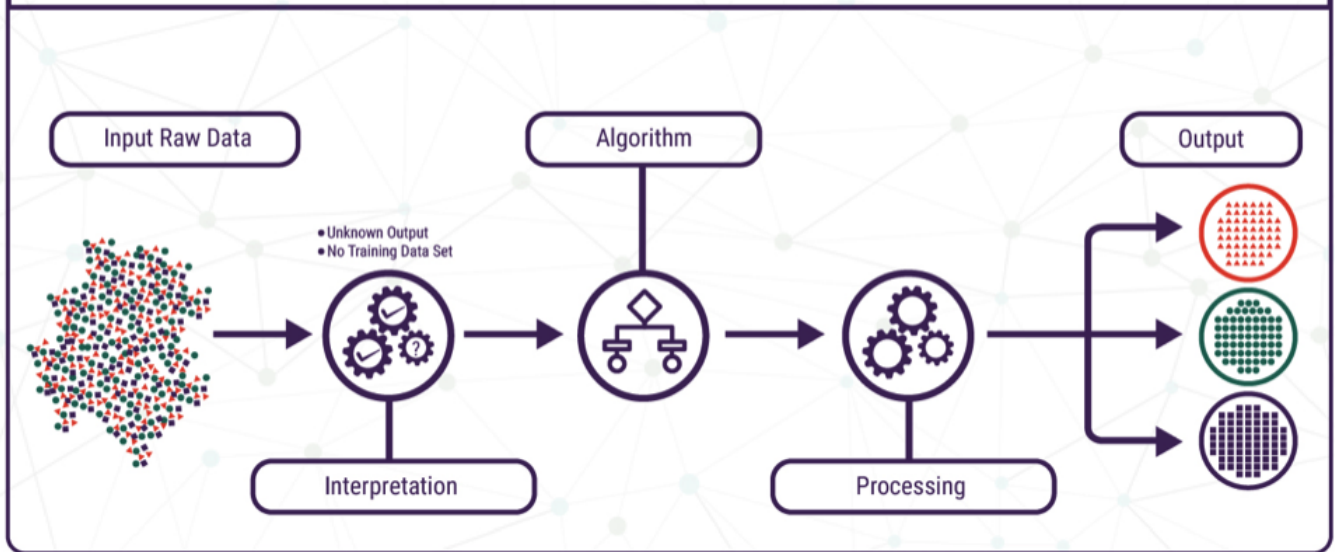


Two examples of **supervised learning** tasks are **regression** and **classification**!

## B. Unsupervised Learning

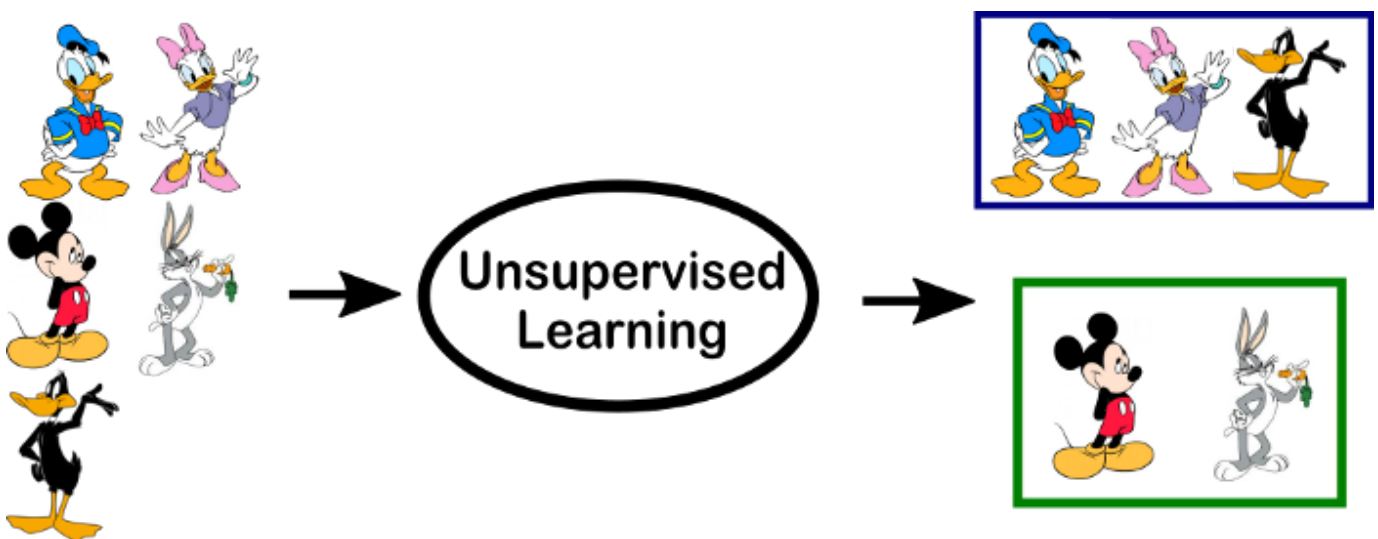
In the case of **unsupervised learning**, the training data is not labeled anymore, so the machine learning algorithm must take decisions without having a ground truth as reference. That means that your dataset is no longer made out of  $(X, y)$  pairs, but only of  $X$  entries. Consequently, the job of the model would be to provide meaningful **insights** about those  $X$  entries, by finding various **patterns** in the data.

# UNSUPERVISED LEARNING



[Image Source](#)

The most common tasks for this learning approach are **clustering algorithms**. In these type of problems, the model must learn how to group various data together, forming **components** or **clusters**. Say that having a table with information about all the Facebook users, you would like to recommend personalized ads for every single person. This sounds like a very difficult task, because there is a huge number of users on the platform. But what if we group users together based on their **traits** (or **features** as your Data Scientist within you might say)? Then, we can suggest the same ads bundle to a whole group of users, making the advertising process easier and cheaper.



[Image Source](#)

🚫🚫🚫 In the dataset of each **unsupervised** machine learning task, there will exist **only X** entries used to identify **patterns** and **insights** about data!

---

### ▼ 3. Performance Evaluation in Machine Learning

Generally, with machine learning algorithms, in the early phase of development, the focus is less on the computational resources (RAM, CPU, IO etc.) and more on the ability of the model to **generalize** well on the data. That means that our algorithm is trained well enough to provide accurate answers for data that has never been seen before (predict a car price for a new car posted on the platform, classify a new vehicle or assign a freshly registered Facebook user to a group). Hence, at first, we try to build a robust and accurate model and then we work towards making it function as computationally inexpensive as possible.

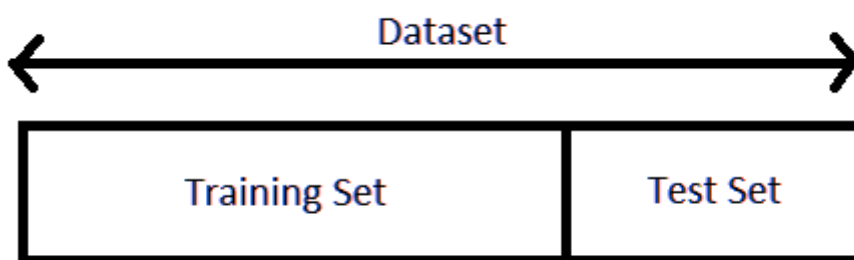
---

🚫🚫🚫 In machine learning, traditional evaluation metrics (RAM, CPU, IO) are not used as frequently!

---

#### A. Training & Test Sets

Evaluating machine learning algorithms is dependent on the type of problem we are trying to solve. But in most cases, we would want to split the data into a **training set** and a **test set**. As the name suggests, the former will be used to actually train the model and the later will be used to verify how well the model generalizes on unseen data. On the most common machine learning tasks, the split ratio ranges from **80-20** to **90-10**, depending on the size of the corpus. When a huge amount of data is operated (say millions or billions of entries), a smaller proportion of the dataset is used as a test set (even 1%) because the actual number of test entries are considered sufficiently numerous. For instance, 1% of 10M entries is 100k and that's generally regarded as **a lot of data** to test on.



🚨🚨🚨 In most problems, the data is split into a **training set** used in model training and a **test set** used in performance evaluation!

## B. Classification Problems

Let's begin our performance analysis journey by analyzing a **classification problem**. Let's suppose that you have a model that was trained to predict whether a given image corresponds to a cat or not. As you already know, this is a **supervised learning** task in which the model must learn to **predict** one of **two classes** - "cat" or "non-cat". In this case, the model is also called a **binary classifier**.



[!\[\]\(8d0f0e0fe25b320c33272c52aec1fbca\_img.jpg\) Image Source](#)

### Confusion Matrix

You fit the model with your training data, you ask it for predictions on the test set and now you have two results: the ground truth  $y$  that was part of your dataset and the predictions  $\hat{y}$  that have just been yielded by your machine learning algorithm. So how do you assess your model in this particular scenario? One way is to build a **confusion matrix**, like the one below:



# Confusion Matrix

	Actually Positive (1)	Actually Negative (0)
Predicted Positive (1)	True Positives (TPs)	False Positives (FPs)
Predicted Negative (0)	False Negatives (FNs)	True Negatives (TNs)

 [Image Source](#)

Now don't get confused (excuse the pun 🙄). It's actually quite simple to interpret it. There are **4 possible predictions** yielded by your binary classifier:

- **True Positives ( $y = \hat{y}$  and  $\hat{y} = \text{"cat"}$ )**: the model identified a cat in the image and in that image it is actually a cat.
- **False Positives ( $y \neq \hat{y}$  and  $\hat{y} = \text{"cat"}$ )**: the model identified a cat in the image, but there was no cat in the image.
- **False Negatives ( $y \neq \hat{y}$  and  $\hat{y} = \text{"non-cat"}$ )**: the model did not recognize a cat in the image, but there was a cat in the image.
- **True Negatives ( $y = \hat{y}$  and  $\hat{y} = \text{"non-cat"}$ )**: the model did not recognize a cat in the image and in the image there was no cat indeed.



The **first step** in analyzing the performance of a **classifier** is to build a **confusion matrix**!

These 4 scenarios can have different levels of importance, depending on the problem one wants to solve. For a smart antivirus, keeping the number of false negatives as low as possible is crucial, even if that means an increase in the false positives. Naturally, it's better to have annoying warnings than having your computer infected because the antivirus was unable to identify the threat. But if you don't necessarily have unusual requirements for your model and you just want to assess its generic performance, there are 3 metrics that can be inferred from this matrix:



## Accuracy

This is the most intuitive metric and represents the number of correctly predicted labels over the number of total predictions, the actual value of prediction (positive or negative) being irrelevant. Again, judging the model by this single metric can be misleading. Consider the case of credit card transaction frauds. There could be a lot of transactions that are perfectly valid, but only a handful of them are frauds. A model that trains on this data could learn to predict that all of the transactions are safe and the ones that are fraudulent are just outliers. Calculating the accuracy on a dataset of 1 million transactions would yield 99.9%. But our model is useless, because letting 1000 frauds unnoticed cannot be allowed.

$$\textit{accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

## Precision

The precision is the total number of correctly classified positive examples divided by the total number of predicted positive examples. If the precision is very high, the probability for our model of classifying non-cat images as cat images is quite low.

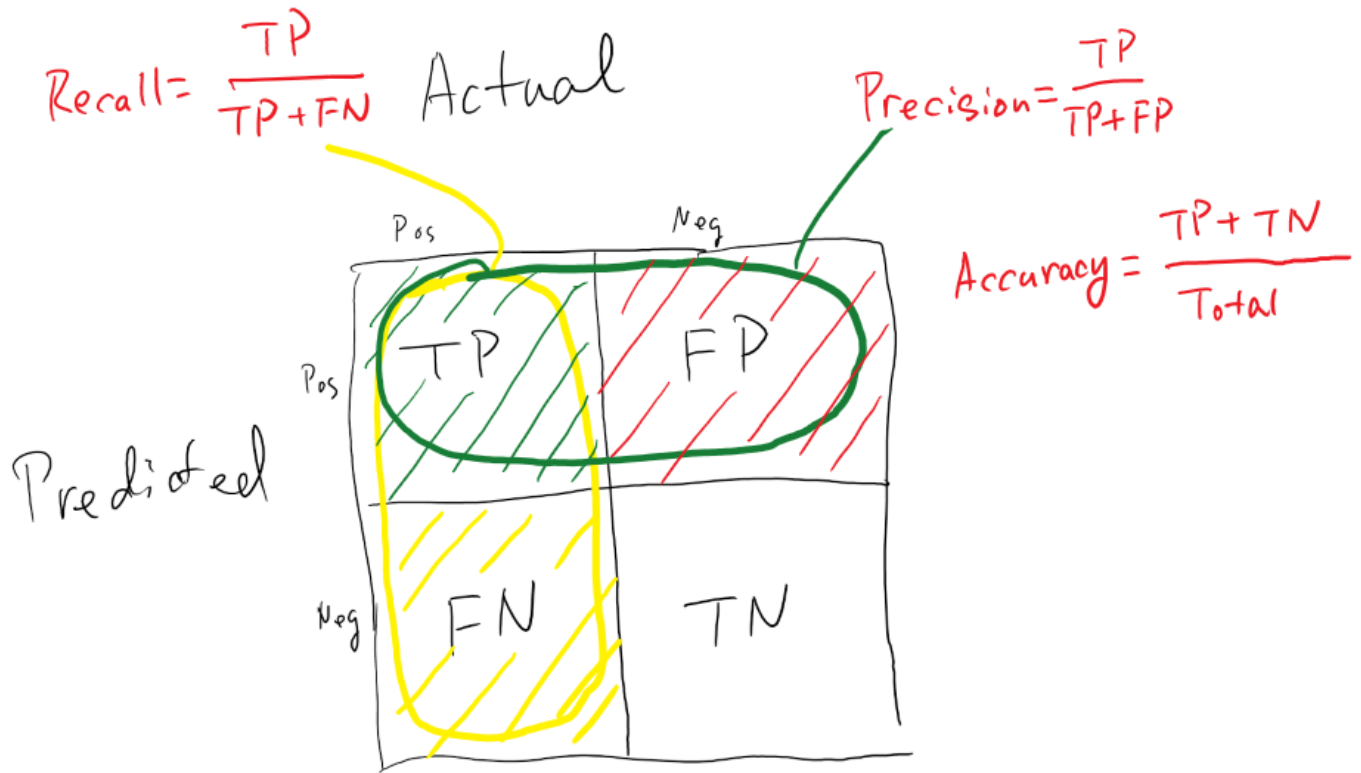
$$\textit{precision} = \frac{TP}{TP + FP}$$

## Recall

The recall is the total number of correctly classified positive examples divided by the total number of actual positive examples. If the recall is very high, the probability for our model of misclassifying cat images is quite low.

$$\textit{recall} = \frac{TP}{TP + FN}$$

When you have **high recall** and **low precision**, most of the cat images are correctly recognized, but there are a lot of false positives. In contrast, when you have **low recall** and **high precision**, we miss a lot of cat images, but those predicted as cat images have a high probability of being indeed cat images and not something else.



[Image Source](#)

## F1 Score

Ideally, you would want to have both **high recall** and **high precision**, but that is not always possible. So you can choose the trade-off you are most comfortable with or you can combine the 2 metrics into 1, by using the **F1 score**, which is actually the harmonic mean of the 2:

$$F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

This single metric is more generic, goes from **0** (worst) to **1** (best) and together with accuracy, can give you a solid intuition on the performance of your model.



From the **confusion matrix** you can extract the **accuracy**, **precision** and **recall**!

## Generic Confusion Matrix

Don't take the terms "positive" and "negative" written in the confusion matrix above literally! They are actually placeholders for the 2 classes ("cat" and "non-cat") the model is trying to predict. This means that you can use more generic classes like "cat" and "dog" or "human" and "animal", depending on the problem you want to solve. As a consequence, **each metric** (accuracy, precision, recall or F1 score) is computed **per class** and if you want to obtain a **generic score** for

the model, you must compute their **average**. For instance, if you have a recall of **0.6** for "cat" and **0.5** for "dog", the **average recall** of the model will be **0.55**.

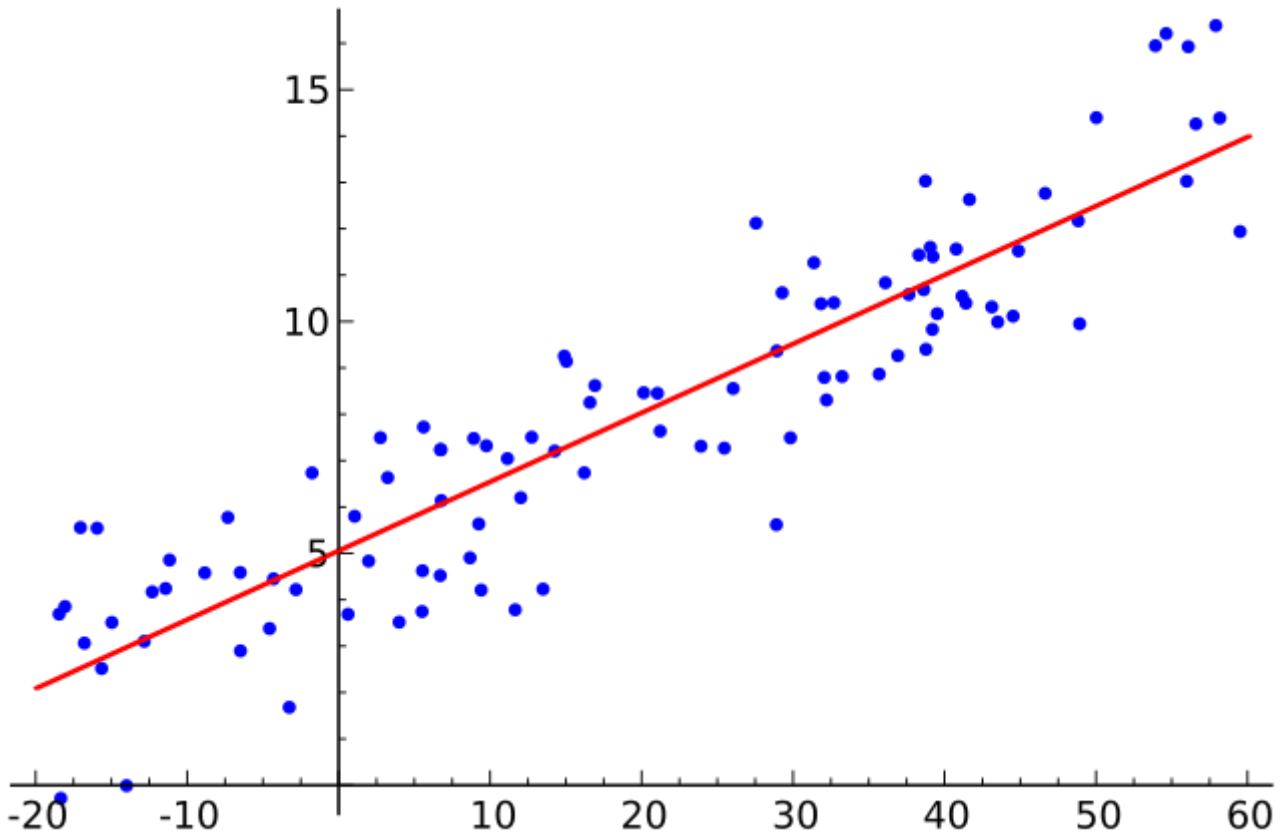
Consequently, the confusion matrix can be generalized to more than 2 classes (see the image below), having the same metrics and ways of computing them.

		Predicted		
		Greyhound	Mastiff	Samoyed
Actual	Greyhound	$P_{GG}$	$P_{MG}$	$P_{SG}$
	Mastiff	$P_{GM}$	$P_{MM}$	$P_{SM}$
	Samoyed	$P_{GS}$	$P_{MS}$	$P_{SS}$

 [Image Source](#)

### C. Regression Problems

As it was previously mentioned, **supervised machine learning** can also imply solving **regression tasks**, where a **numerical** or **continuous** value is used as a label. Let's take an example. Suppose that you want to predict the budget for an advertisement campaign, based on the revenue of the company. For this task, your model will learn from a training set of (X, y) pairs and will try to find the best  $\hat{h}$  that approximates the behaviour of the function  $f(X) = y$ . If we assume that the relationship between the two is **linear**, your model must learn how to **draw a line** that fits the points (X, y) the best. A visual representation can be seen in the figure below:



[Image Source](#)

### Root Mean Squared Error

But how do we know if that line is drawn correctly? Well, first we must define an **error** or **loss function** that can mathematically indicate us how far from the points the line has been drawn. In linear regression tasks, the most common approach is to use the **root mean squared error (RMSE)**, depicted below. You can sometimes get rid of the root and just compute the **MSE** to avoid extra computations.

$$RMSE = \sqrt{\frac{1}{n} \sum_{j=1}^n (y_j - \hat{y}_j)^2}$$

In this formula,  $y_j$  is the ground truth and  $\hat{y}_j$  is the result of the partially learned function  $\hat{h}(X_j)$ . Basically, the model must apply successive corrections to  $\hat{h}$ , such that the predicted  $\hat{y}_j$  values lead to a smaller RMSE. This process of minimizing the loss is also called **optimization** and is one of the foundational principles of machine learning. You can find more about it [here](#).

Similarly to the classification tasks, for regression problems, the value of the RMSE can be used as a **performance metric** for the model.

### R<sup>2</sup> Correlation

Now let's say that your model was properly trained and you have some predicted labels  $\hat{\mathbf{y}}$  and some ground truth values  $\mathbf{y}$ . In the case of classification problems, with these two pieces of information you could immediately compute the F1 score or the accuracy. But with regression, a 0 to 1 score cannot be simply derived. One metric that can be used however, is the  **$R^2$  correlation** and it is computed using the following formula:

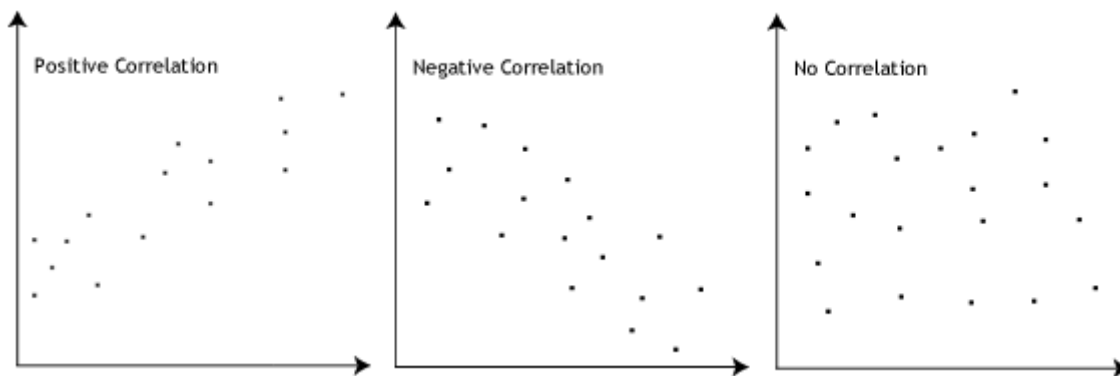
$$R^2 = 1 - \frac{\sum_{j=1}^n (y_j - \hat{y}_j)^2}{\sum_{j=1}^n (y_j - \bar{y})^2}$$

where  $\hat{y}_j$  is the predicted value,  $\bar{y}$  is the mean of the ground truth labels and  $y_j$  is the ground truth.

This score lies between  $-\infty$  and **1** and has the following interpretation:

- **Close to 1**: high positive linear correlation between X and y
- **Close to 0**: a linear correlation between X and y cannot be identified
- **Close to  $-\infty$** : high negative linear correlation between X and y

These 3 scenarios are visually represented in the figures below:



[!\[\]\(8bba887393ca45b761e5cb49e755e762\_img.jpg\) Image Source](#)



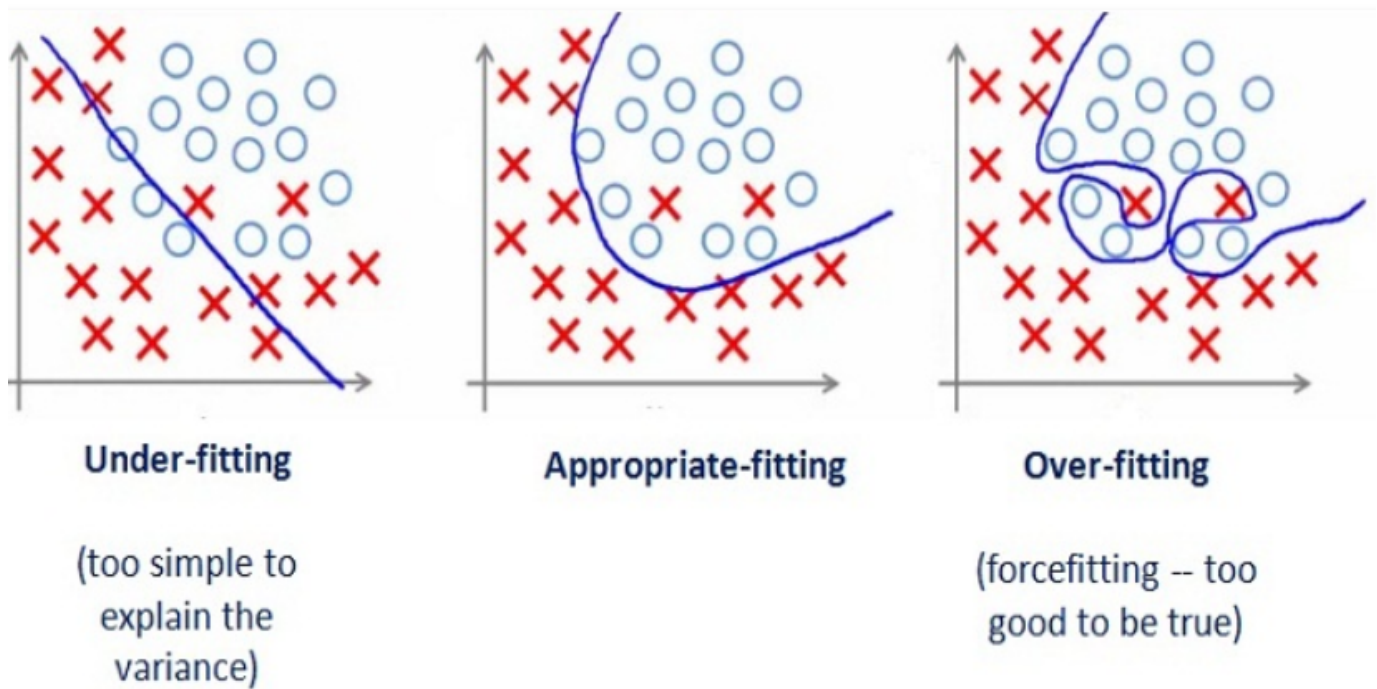
For **regression tasks** you can use **RMSE** and **R-squared score** as performance metrics!

## D. Underfitting vs Overfitting

Splitting the data into a **training set** and a **test set** is not only helping us to determine the **accuracy** or **error** of the model's predictions, but can also give us an insight about its **behaviour**

or **generalization capabilities**.

Let's take a trained binary classifier as an example and the accuracy as a performance metric. If it has a **small test score**, but a **high training score**, the model relied too much on the training data and is not able to generalize well on entries that has never seen before. In this case, we say that it has **overfit** the training data. In this particular case, the model is said to have a **high variance**. In contrast, if the model learned a function that is too generic, the problem of **underfitting** or **high bias** occurs. This time, the problem can be inferred from a **training score** that is **too small**. These situations were visually represented in the figures below:



 [Image Source](#)

The underlying causes of the aforementioned problems depend heavily on the model and how its **hyperparameters** were fine-tuned. Discussing them is outside the scope of this laboratory class, but one can learn more about those topics from these articles: [Underfitting & Overfitting](#) and [Hyperparameters](#).



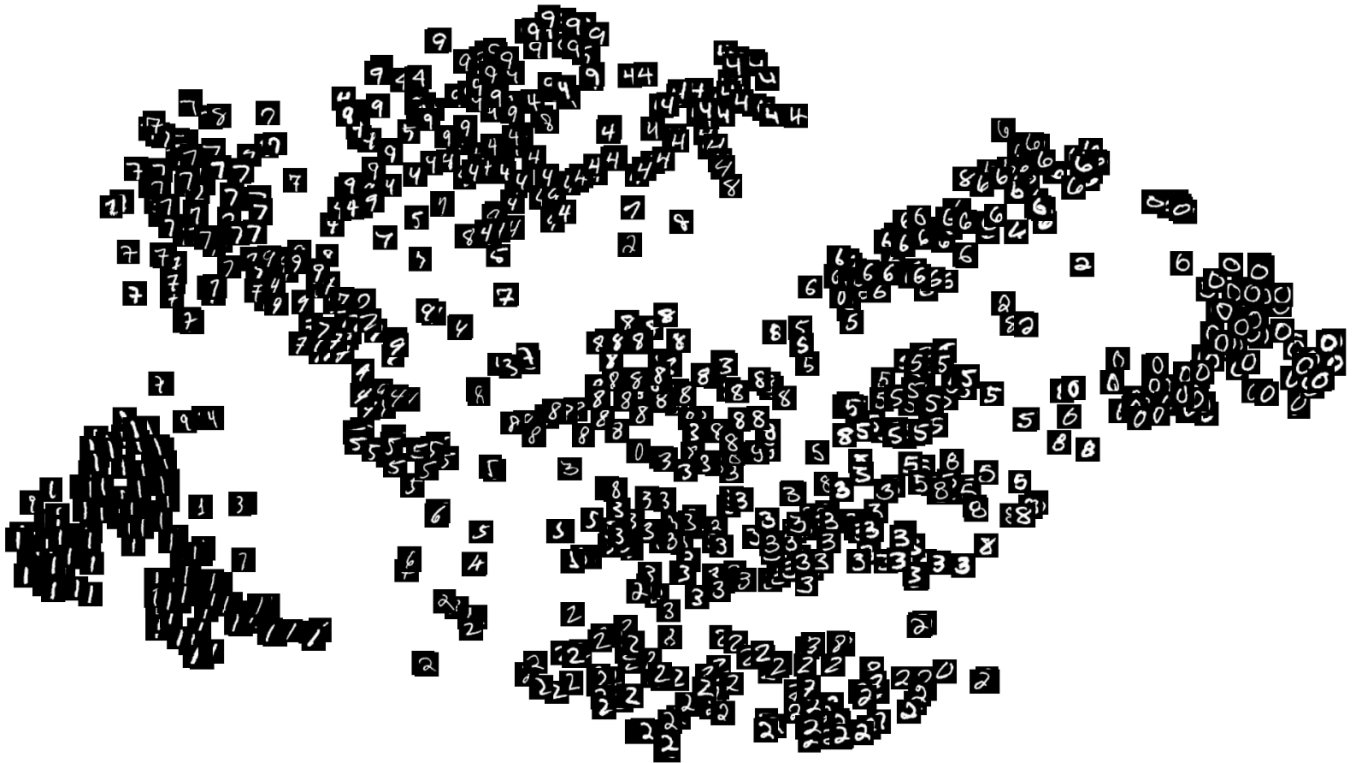
Besides looking at the usual **performance metrics**, one must also notice if the model **appropriately fits** the data!

## ▼ E. Clustering Algorithms

In the end, we should talk a little bit about **unsupervised learning**. As you already know, for such tasks, there is **no ground truth** - just a set of X values that might have some **underlying structure**

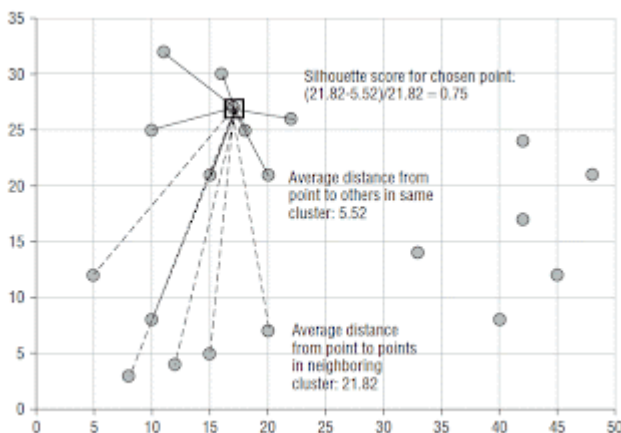
or **pattern** that can be learned. Evaluating a model without having something as reference might seem a pretty difficult task. And in some scenarios, you might be right!

Say that we wanted to group images containing handwritten digits into **clusters** and at the end of the learning process, our model grouped the data like this:



[Image Source](#)

At first glance, the clustering outcome looks good, but how can we express this “good-looking” result in a more formal manner? One solution is to measure how compact the clusters are, yet distant from one another, by computing a **silhouette score**. Because the formulas are too cumbersome to write in here, I will leave you a [link](#), where everything is explained clearly. However, one might understand the concept by looking at this simple example:



[Image Source](#)

## Exercise 0: Prerequisite



Import the necessary libraries and define some printing functions for later use.

```
# Import the necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

# Load various printing functions

# Pretty prints a data frame without display limits
def print_df(df):
    with pd.option_context('display.max_rows', None, 'display.max_columns', None):
        print(df)

# Pretty prints the results of the classifier performance evaluation
def print_classifier_results(accuracy, precision, recall, f1, mode):
    print()
    print('~~~~~ CLASSIFICATION RESULTS (' + mode.upper() + ') ~~~~~')
    print(' Accuracy: ' + str(accuracy))
    print(' Precision: ' + str(precision))
    print(' Recall: ' + str(recall))
    print(' F1 Score: ' + str(f1))

# Pretty prints the results of the regressor performance evaluation
def print_regressor_results(rmse, mae, r_squared, mode):
    print()
    print('~~~~~ REGRESSION RESULTS (' + mode.upper() + ') ~~~~~')
    print('Root Mean Squared Error (RMSE): ' + str(rmse))
    print(' Mean Absolute Error (MAE): ' + str(mae))
    print(' R-squared Score: ' + str(r_squared))

# Pretty prints the results of the multiple classifications
def print_fitting_results(accuracy_list):

    print()
    print('~~~~~ CLASSIFICATION RESULTS ~~~~~')

    for accuracy_pair in accuracy_list:
        print('Training Set Accuracy 1: ' + str(accuracy_pair[0]))
        print(' Test Set Accuracy 1: ' + str(accuracy_pair[1]))
        print()

# Pretty prints the results of the clustering algorithm
```

```
def print_clustering_results(silhouette_score):

    print()
    print('~~~~~ CLUSTERING RESULTS ~~~~~')
    print('Silhouette Score: ' + str(silhouette_score))
```

### Exercise 1: Classification (30p)

In this exercise, you will learn how to properly evaluate a **classifier**. We chose a **decision tree** for this example, but feel free to explore other alternatives. You can find out more about decision trees [here](#). For all the associated tasks, you will use the [diabetes.csv](#) dataset. The model must learn to determine whether the patient suffers from diabetes (**0** or **1**) by looking at a dataset with the following **features**:

- Number of pregnancies
- Glucose level
- Blood pressure
- Skin thickness
- Insulin level
- Body Mass Index (BMI)
- Diabetes pedigree function (likelihood of diabetes based on family history)
- Age

```
# Load the dataset from the local session folder
classification_dataset = pd.read_csv("https://raw.githubusercontent.com/vladastefai")

# Take a look at the first entries of the dataset
print_df(classification_dataset.head())
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	\
0	6	148	72	35	0	33.6	
1	1	85	66	29	0	26.6	
2	8	183	64	0	0	23.3	
3	1	89	66	23	94	28.1	
4	0	137	40	35	168	43.1	

	DiabetesPedigreeFunction	Age	Outcome
0	0.627	50	1
1	0.351	31	0
2	0.672	32	1
3	0.167	21	0
4	2.288	33	1

```
# Extract the features from the dataset
classification_X = classification_dataset.iloc[:, :-1]

# Extract the labels from the dataset
classification_y = classification_dataset.iloc[:, -1:]
```

```
# Split the data into a training set and a test set with a 80-20 ratio
classification_X_train, classification_X_test, classification_y_train, classificat:

# Build a classifier (less important: a decision tree will be used)
classifier = DecisionTreeClassifier(random_state=42)

# Fit data from the training set to the classifier
classifier.fit(classification_X_train, classification_y_train)

# Make predictions on the test set
classification_y_pred = classifier.predict(classification_X_test)
```

### Task A (15p)

Evaluate the classifier by **manually** computing the **accuracy**, **precision**, **recall** and **F1 score**. These metrics are derived from the **confusion matrix** but you don't have to build this matrix yourself. You can use [Scikit-learn](https://scikit-learn.org/) for that.

```
# Computes the accuracy of the model using the confusion matrix
def compute_accuracy(cm):
    accuracy = (cm[0][0] + cm[1][1]) / (cm[0][0] + cm[1][1] + cm[0][1] + cm[1][0])
    return accuracy

# Computes the precision of the model using the confusion matrix
def compute_precision(cm):
    precision = cm[0][0] / (cm[0][0] + cm[0][1])
    return precision

# Computes the recall of the model using the confusion matrix
def compute_recall(cm):
    recall = cm[0][0] / (cm[0][0] + cm[1][0])
    return recall

# Computes the F1 score of the model using the precision and recall
def compute_f1_score(precision, recall):
    f1 = 2 * (precision * recall) / (precision + recall)

    return f1
```

Before anything else, we must compute the **confusion matrix**. Luckily, the **metrics** package of the *Scikit-learn* library has just the right function for this task:

```
cm = confusion_matrix(y_test, y_pred)
cm[0][0], cm[1][1] = cm[1][1], cm[0][0]
```



Please note that in order to compute a binary confusion matrix that looks just like the ones used in the above diagrams, a small swap has to be performed.

For this exercise we simply identify the **TP, TN, FP, FN** terms on the **confusion matrix** and then compute the various metrics using their formulas. For instance, for the **precision**, we can write the following lines of code:

```
precision = cm[0][0] / (cm[0][0] + cm[0][1])
```



You should be able to compute the **accuracy, recall** and **F1 score** by yourself.

```
from sklearn.metrics import confusion_matrix

cm = confusion_matrix(classification_y_test, classification_y_pred)
cm[0][0], cm[1][1] = cm[1][1], cm[0][0]

task_a_accuracy = compute_accuracy(cm)
task_a_precision = compute_precision(cm)
task_a_recall = compute_recall(cm)
task_a_f1 = compute_f1_score(task_a_precision, task_a_recall)

print_classifier_results(task_a_accuracy, task_a_precision, task_a_recall, task_a_f1)
```

```
~~~~~ CLASSIFICATION RESULTS (DUMB) ~~~~~
Accuracy: 0.7467532467532467
Precision: 0.625
Recall: 0.7272727272727273
F1 Score: 0.6722689075630253
```

### Task B (15p)

Evaluate the classifier using the **metrics** package from the *Scikit-learn* library. Again, **accuracy, precision, recall** and **F1 score** are required.



**HINT:** There are 2 *Scikit-learn* functions that will help you with this computation: **accuracy\_score** and **precision\_recall\_fscore\_support**.

- Because we are computing these metrics on a binary classifier, think about what is the suitable value for the **average** parameter of the **precision\_recall\_fscore\_support** function.

```

from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_recall_fscore_support

task_b_accuracy = accuracy_score(classification_y_test, classification_y_pred)
task_b_precision, task_b_recall, task_b_f1, _ = precision_recall_fscore_support(classification_y_test, classification_y_pred)

print_classifier_results(task_b_accuracy, task_b_precision, task_b_recall, task_b_f1)

~~~~~ CLASSIFICATION RESULTS (SMART) ~~~~~
Accuracy: 0.7467532467532467
Precision: [0.83333333 0.625      ]
Recall: [0.75757576 0.72727273]
F1 Score: [0.79365079 0.67226891]

```

## Exercise 2: Linear Regression (40p)

In this exercise, you will learn how to properly evaluate a **regression model**. We chose a **simple linear regressor** for this example, but feel free to explore other alternatives. You can find out more about linear regressors [here](#). For all the associated tasks, you will use the [weather.csv](#) dataset. The model must learn to determine what is the **maximum temperature** for a certain day (**y**) based on the **minimum temperature (X)**.

```

# Load the dataset from the local session folder
regression_dataset = pd.read_csv("https://raw.githubusercontent.com/vladastefanescu/machine-learning-introduction/blob/main/Weather/weather.csv")

# Take a look at the first entries of the dataset
print_df(regression_dataset.head())

   MinTemp  MaxTemp
0  22.222222  25.555556
1  21.666667  28.888889
2  22.222222  26.111111
3  22.222222  26.666667
4  21.666667  26.666667

# Extract the features from the dataset
regression_X = regression_dataset.iloc[:, :-1]

# Extract the labels from the dataset
regression_y = regression_dataset.iloc[:, -1:]

# Split the data into a training set and a test set with a 80-20 ratio
regression_X_train, regression_X_test, regression_y_train, regression_y_test = train_test_split(regression_X, regression_y, test_size=0.2, random_state=42)

# Build a linear regressor
regressor = LinearRegression()

# Fit data from the training set to the regressor
regressor.fit(regression_X_train, regression_y_train)

```

```
regressor.fit(regression_X_train, regression_y_train)

# Make predictions on the test set
regression_y_pred = regressor.predict(regression_X_test)
```

### Task A (15p)

Evaluate the regressor by **manually** computing the **Root Mean Squared Error (RMSE)**, **Mean Absolute Error (MAE)** and **R<sup>2</sup> score**.

For instance, for the **RMSE** we simply apply the formula above:

```
rmse = list(np.sqrt(np.sum((y_test - y_pred) ** 2) / len(y_test)))[0]
```


---

▼  You should be able to compute **MAE** and **R<sup>2</sup> score** on your own!

---

```
# Computes the RMSE of the model
def compute_rmse(y_test, y_pred):
    return list(np.sqrt(np.sum((y_test - y_pred) ** 2) / len(y_test)))
```

---

 We know that **MAE** was not covered in the first section of this laboratory class and that is why you must use your powerful [Google](#) skills to solve this one. 😊

---

```
# Computes the MAE of the model
def compute_mae(y_test, y_pred):
    return float(np.mean(np.abs(y_test - y_pred)))

# Computes the R-squared score of the model
def compute_r2_score(y_test, y_pred):
    arg1 = (len(y_test) * sum(map(float, (y_test * y_pred).values))) - float(np.sum(y_test * y_pred))
    arg2_sqrt = np.sqrt(((len(y_test) * np.sum(np.square(y_test))) - np.square(np.sum(y_test))) *
                        ((len(y_test) * np.sum(np.square(y_pred))) - np.square(np.sum(y_pred))))
    return float(arg1 / arg2_sqrt)

task_a_rmse = compute_rmse(regression_y_test, regression_y_pred)
task_a_mae = compute_mae(regression_y_test, regression_y_pred)
task_a_r_squared = compute_r2_score(regression_y_test, regression_y_pred)

print_regressor_results(task_a_rmse, task_a_mae, task_a_r_squared, 'dumb')
```

```
~~~~~ REGRESSION RESULTS (DUMB) ~~~~~
Root Mean Squared Error (RMSE): [4.137065480286405]
```

Mean Absolute Error (MAE): 3.177483117041877  
 R-squared Score: 0.8787430104207181

### Task B (10p)

Evaluate the regressor using the **metrics** package from the *Scikit-learn* library. Again, **RMSE**, **MAE** and **R<sup>2</sup> score** are required.



**HINT:** You might not find a function that computes the actual **RMSE**, but the **MSE**.

```
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error

task_b_rmse = mean_squared_error(regression_y_test, regression_y_pred)
task_b_mae = mean_absolute_error(regression_y_test, regression_y_pred)
task_b_r_squared = compute_r2_score(regression_y_test, regression_y_pred)

print_regressor_results(task_b_rmse, task_b_mae, task_b_r_squared, 'smart')
```

```
~~~~~ REGRESSION RESULTS (SMART) ~~~~~
Root Mean Squared Error (RMSE): 17.115310788177382
Mean Absolute Error (MAE): 3.177483117041783
R-squared Score: 0.8787430104207181
```

### Task C (15p)

Train the model on variously sized chunks of the original dataset and notice how the **RMSE** changes. To better illustrate the behaviour, you should build a **plot** having the **data size** on the **X axis** and the **RMSE value** on the **Y axis**. Moreover, you should be able to **explain** the observed behaviour to the assistant.

Because this is rather a plotting task, we will do the hard work for you and compute the list of RMSE values for the various chunk sizes:

```
n = min(X.shape[0], max_chunk_size)
chunk_size = int((n - min_chunk_size) / chunks)

# Create 2 lists used in the plotting logic
size_list = []
rmse_list = []

# Train a model for each chunk
for i in range(0, chunks):

    # Compute the size of the current chunk
    size = min_chunk_size + (i + 1) * chunk_size
```



```

size_list.append(size)

# Select a chunk from the whole dataset
sample_X = X.sample(n=size, random_state=42)
sample_y = y.sample(n=size, random_state=42)

# Split the data into a training set and a test set with a 80-20 ratio
X_train, X_test, y_train, y_test = train_test_split(sample_X, sample_y, test_size=0.2)

# Build a linear regressor
regressor = LinearRegression()

# Fit data from the training set to the regressor
regressor.fit(X_train, y_train)

# Make predictions on the test set
y_pred = regressor.predict(X_test)

# Compute the rmse
rmse = compute_rmse(y_test, y_pred)
rmse_list.append(rmse)

```

Please read the code and try to understand it by following the comments. Building the plot is on you! 😊

```

# Plots the evolution of the RMSE when the dataset size varies
def plot_rmse_evolution(X, y, chunks, min_chunk_size, max_chunk_size):
    # TODO - TASK C
    # HINT: See the code above
    # Build the list of RMSE values
    sizes = []
    rmse_values = []
    n = min(X.shape[0], max_chunk_size)
    chunk_size = int((n - min_chunk_size) / chunks)

    for i in range(0, chunks):
        size = min_chunk_size + (i + 1) * chunk_size
        sizes.append(size)
        sample_X = X.sample(n=size, random_state=42)
        sample_y = y.sample(n=size, random_state=42)
        X_train, X_test, y_train, y_test = train_test_split(sample_X, sample_y, test_size=0.2)
        regressor = LinearRegression()
        regressor.fit(X_train, y_train)
        y_pred = regressor.predict(X_test)
        rmse = compute_rmse(y_test, y_pred)
        rmse_values.append(rmse)

    # TODO - TASK C

```

```
# Plot the RMSE evolution
plt.plot(sizes, rmse_values)
plt.show()
```

Play with the following parameters and observe how the plot changes. Make **at least 5** changes to the chunk-related parameters. That means that **5 plots should be generated**.

```
# TODO - TASK C
CHUNKS = 100
MIN_CHUNK_SIZE = 1000
MAX_CHUNK_SIZE = 1000000

plot_rmse_evolution(regression_X, regression_y, CHUNKS, MIN_CHUNK_SIZE, MAX_CHUNK_SIZE)

CHUNKS = 1000
MIN_CHUNK_SIZE = 100
MAX_CHUNK_SIZE = 10000

plot_rmse_evolution(regression_X, regression_y, CHUNKS, MIN_CHUNK_SIZE, MAX_CHUNK_SIZE)

CHUNKS = 600
MIN_CHUNK_SIZE = 2000
MAX_CHUNK_SIZE = 5000000

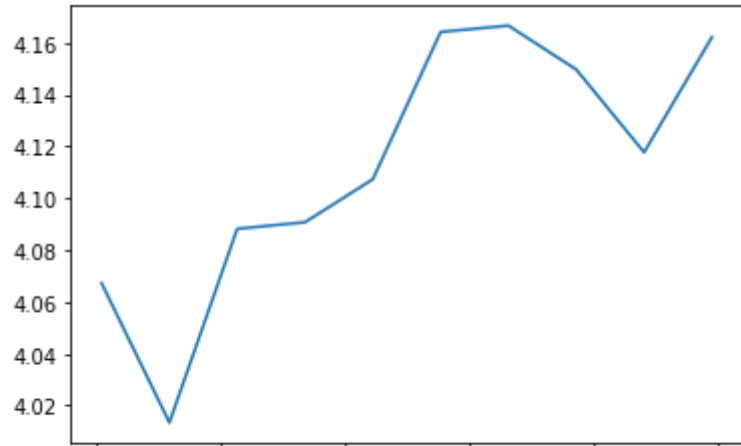
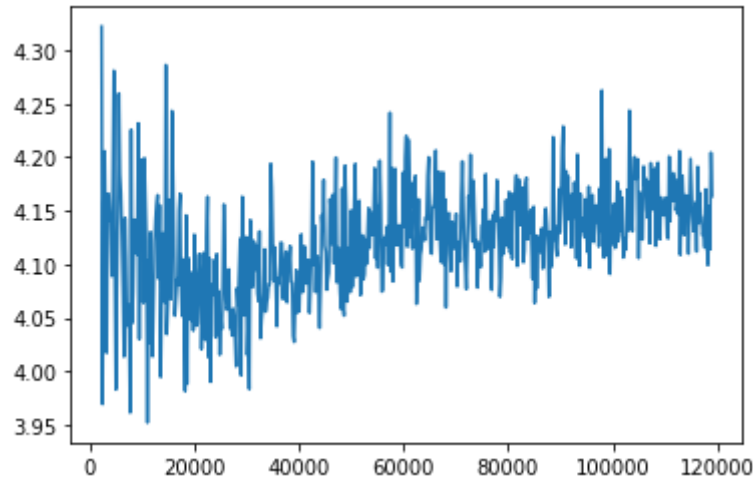
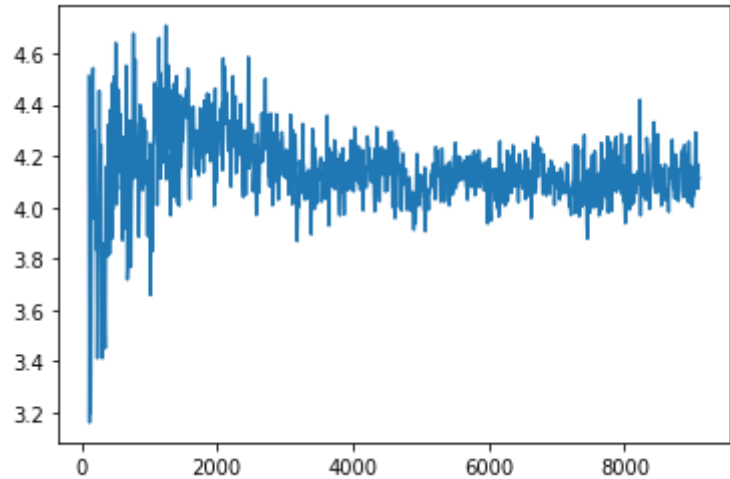
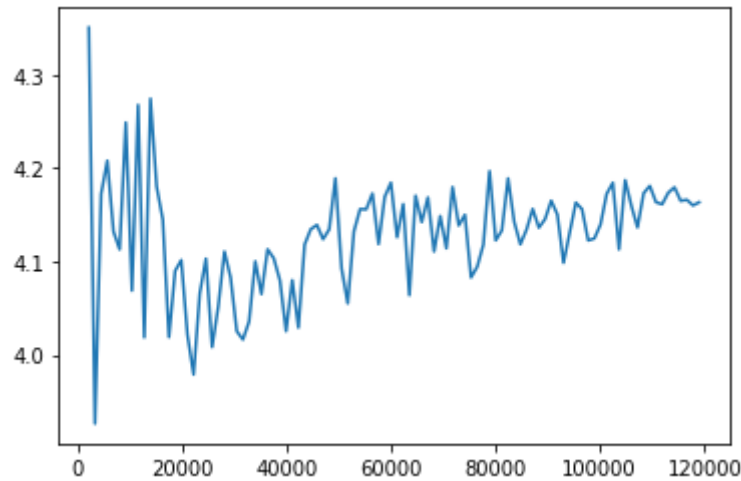
plot_rmse_evolution(regression_X, regression_y, CHUNKS, MIN_CHUNK_SIZE, MAX_CHUNK_SIZE)

CHUNKS = 10
MIN_CHUNK_SIZE = 10000
MAX_CHUNK_SIZE = 10000000

plot_rmse_evolution(regression_X, regression_y, CHUNKS, MIN_CHUNK_SIZE, MAX_CHUNK_SIZE)

CHUNKS = 500
MIN_CHUNK_SIZE = 15078
MAX_CHUNK_SIZE = 1000123

plot_rmse_evolution(regression_X, regression_y, CHUNKS, MIN_CHUNK_SIZE, MAX_CHUNK_SIZE)
```



**Exercise 3: Fitting Behaviour (15p)**

In this exercise, you will learn how to properly evaluate the **data fitting behaviour** of the **binary classifier** from **exercise 1**. For all the associated tasks, you will use the [diabetes.csv](#) dataset again. This time, **3 models** will be trained with different parameters and your job is to analyse their behaviour. For that, you will look at the **accuracy** of each model computed on both the **training set** and the **test set**.

```
# Load the dataset from the local session folder
fitting_dataset = pd.read_csv("https://raw.githubusercontent.com/vladastefanescu/machine-learning-introduction/master/diabetes.csv")

# Take a look at the first entries of the dataset
print_df(fitting_dataset.head())
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	\
0	6	148	72	35	0	33.6	
1	1	85	66	29	0	26.6	
2	8	183	64	0	0	23.3	
3	1	89	66	23	94	28.1	
4	0	137	40	35	168	43.1	

	DiabetesPedigreeFunction	Age	Outcome
0	0.627	50	1
1	0.351	31	0
2	0.672	32	1
3	0.167	21	0
4	2.288	33	1

```
# Extract the features from the dataset
fitting_X = fitting_dataset.iloc[:, :-1]

# Extract the labels from the dataset
fitting_y = fitting_dataset.iloc[:, -1:]

# Split the data into a training set and a test set with a 80-20 ratio
fitting_X_train, fitting_X_test, fitting_y_train, fitting_y_test = train_test_split(fitting_X, fitting_y, test_size=0.2, random_state=42)

# Build 3 different trained models
classifiers = [DecisionTreeClassifier(max_depth=1, random_state=42).fit(fitting_X_train, fitting_y_train),
                DecisionTreeClassifier(max_depth=5, random_state=42).fit(fitting_X_train, fitting_y_train),
                DecisionTreeClassifier(max_depth=32, random_state=42).fit(fitting_X_train, fitting_y_train)]
```

### Task A (10p)

For each model, make predictions on both the **training set** and **test set** and compute the corresponding **accuracy values**.

```
# Makes prediction on both the training set and test set
# HINT 1: You can reuse some code from the previous exercises
# HINT 2: The models are already trained
def make_predictions(clf, X_train, X_test, y_train, y_test):
    train_accuracy = accuracy_score(clf.predict(X_train), y_train)
```

```

test_accuracy = accuracy_score(clf.predict(X_test), y_test)

return train_accuracy, test_accuracy

# Make predictions on the training and test sets and evaluate the performance of each classifier
accuracy_list = []
for clf in classifiers:
    accuracy_list.append(make_predictions(clf, fitting_X_train, fitting_X_test, fitting_y_train, fitting_y_test))

# Print the performance evaluation results
print_fitting_results(accuracy_list)

```

```

~~~~~ CLASSIFICATION RESULTS ~~~~~
Training Set Accuracy 1: 0.7345276872964169
Test Set Accuracy 1: 0.7402597402597403

Training Set Accuracy 1: 0.8420195439739414
Test Set Accuracy 1: 0.7922077922077922

Training Set Accuracy 1: 1.0
Test Set Accuracy 1: 0.7467532467532467

```

### Task B (5p)

Comment the results by specifying which is the **best model** in terms of fitting and which are the models that **overfit** or **underfit** the dataset.

```
"""
```

```
TODO - TASK B
```

```
Your conclusion here...
```

```
"""
```

```
'\nTODO - TASK B\n\nYour conclusion here...\n'
```

### Exercise 4: Clustering (15p)

In this exercise, you will learn how to properly evaluate a **clustering model**. We chose a **K-means clustering algorithm** for this example, but feel free to explore other alternatives. You can find out more about K-means clustering algorithms [here](#). For all the associated tasks, you don't have to use any input dataset, because the clusters are generated in the skeleton. The model must learn how to group together **points in a 2D space**.

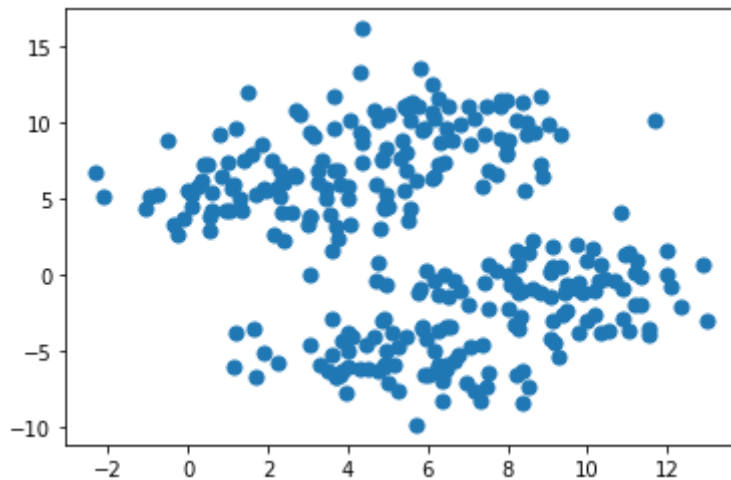
```

# NOTE: You can play around with these values
CLUSTERS = 4
SAMPLES = 300
CLUSTERS_STD = 2

```

```
# Generate a dataset
clustering_X, _ = make_blobs(n_samples=SAMPLES, centers=CLUSTERS, cluster_std=CLUS

# Plot the freshly generated blobs
plt.scatter(clustering_X[:, 0], clustering_X[:, 1], s=50)
plt.show()
```



```
# Create a clustering model (less important: a K-means clustering algorithm will be
clustering_model = KMeans(n_clusters=CLUSTERS, random_state=13)

# Fit the data
clustering_model.fit(clustering_X)

# Predict a cluster number for each point
clustering_y_pred = clustering_model.predict(clustering_X)
```

### Task A (5p)

Compute the **silhouette score** of the model by using a *Scikit-learn* function found in the **metrics** package.

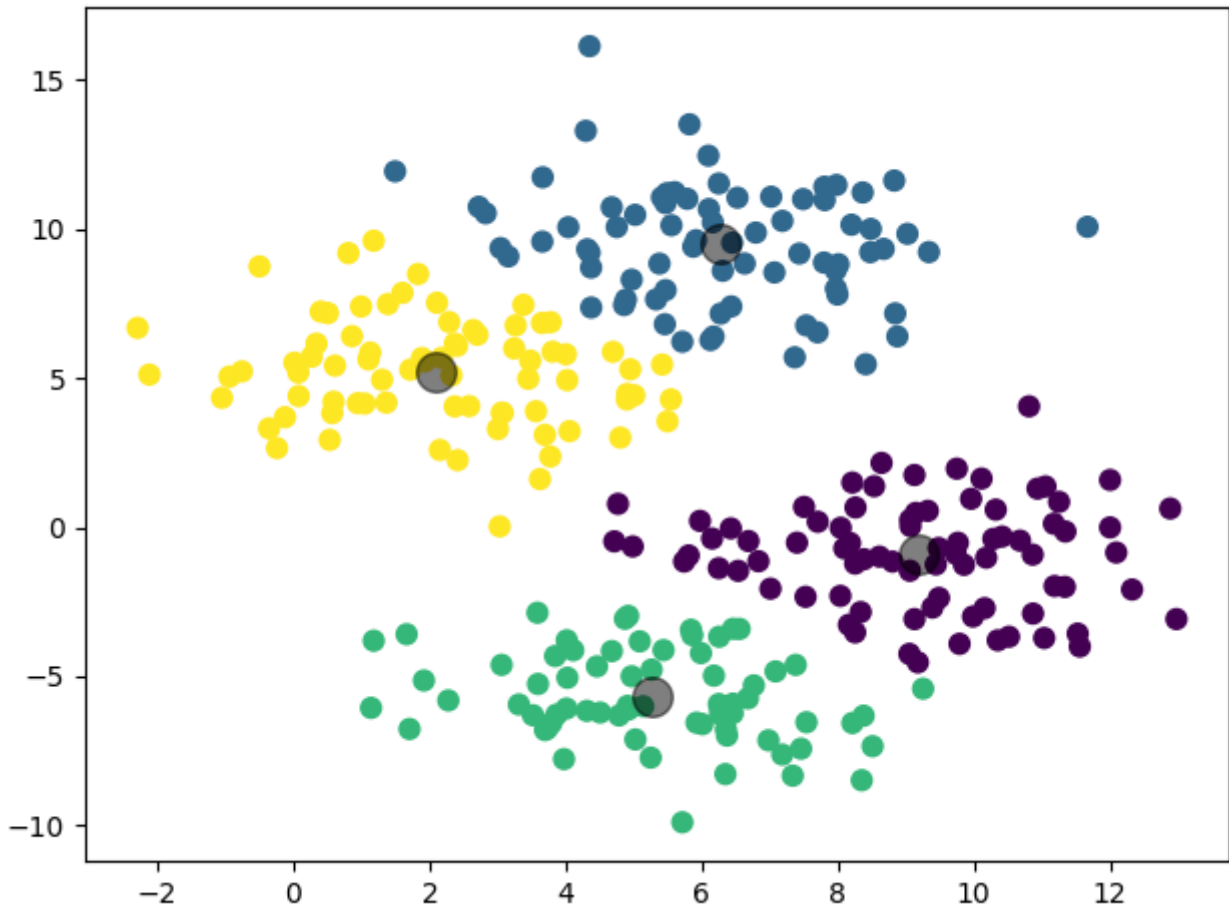
```
# Compute the silhouette score of the clusters and print it
from sklearn.metrics import silhouette_score

clustering_score = silhouette_score(clustering_X, clustering_y_pred)
print_clustering_results(clustering_score)
```

```
~~~~~ CLUSTERING RESULTS ~~~~~
Silhouette Score: 0.4923967156182758
```

### Task B (10p)

Fetch the **centres of the clusters** (the model should already have them ready for you 😊) and **plot** them together with a **colourful 2D representation** of the data groups. Your plot should look similar to the one below:



You can also play around with the **standard deviation** of the generated blobs and observe the different outcomes of the clustering algorithm:

```
CLUSTERS_STD = 2
```

You should be able to discuss these observations with the assistant.



**HINT:** The **plotting code** is very similar to the one found in the skeleton. You can also [Google](#) it out. 😊.

Look at the hint above and solve the tasks marked with **TODO - TASK B**. Make **at least 3** changes to the standard deviation. That means that **3 plots should be generated**.



```
CLUSTERS = 5
SAMPLES = 500
CLUSTERS_STD = 10

def plot_clusters(cluster_std_val):
    CLUSTERS_STD = cluster_std_val
    clustering_X, _ = make_blobs(n_samples=SAMPLES, centers=CLUSTERS,
                                cluster_std=CLUSTERS_STD, random_state=4)

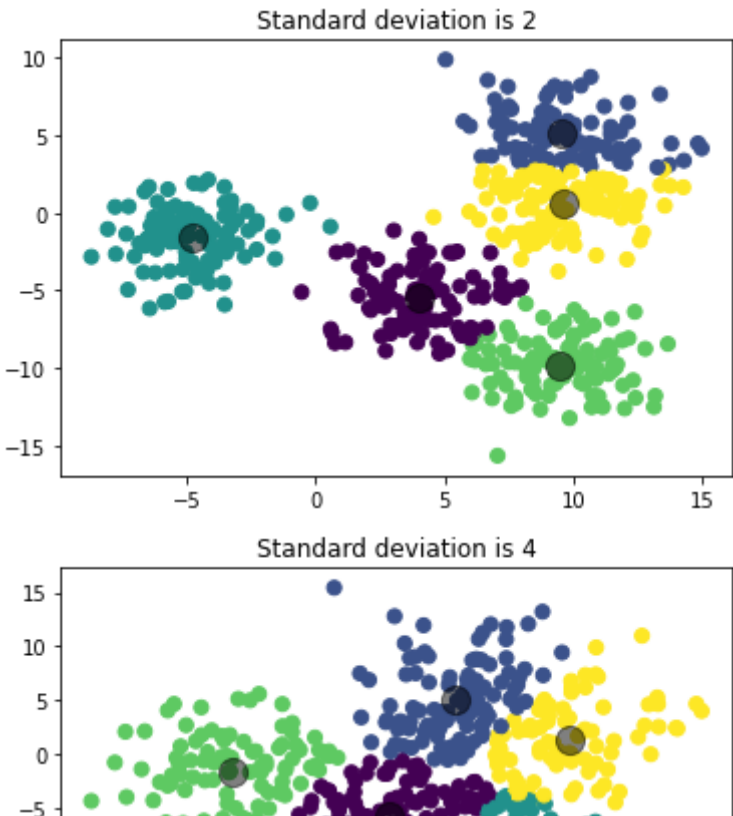
    # K-means clustering
    clustering_model = KMeans(n_clusters=CLUSTERS, random_state=4)

    # Fit the data
    clustering_model.fit(clustering_X)

    # Predict a cluster number for each point
    clustering_y_pred = clustering_model.predict(clustering_X)
    plt.scatter(clustering_X[:, 0], clustering_X[:, 1],
                c=clustering_y_pred, s=50)
    clustering_centers = clustering_model.cluster_centers_

    #Draw plot
    plt.scatter(clustering_centers[:, 0], clustering_centers[:, 1],
                c='black', s=200, alpha=0.5)
    plt.title("Standard deviation is %d" % CLUSTERS_STD)
    plt.show()

plot_clusters(2)
plot_clusters(4)
plot_clusters(6)
```



**Bonus: Feedback (10p)**

Please take a minute to fill in the [feedback form](#) for this lab.

