Universitat Politècnica de Catalunya
Facultat d'Informàtica de Barcelona

Heart Attack Prediction in Indonesia

Mineria de dades
2024-2025

Teacher: Manuel Gijon
Authors: Bednár Maroš, Escofet González Gina, Foroudian Kimia, Lafuente González Alex and Smyhelskyy Yaskevych Sergio

Table of contents

1.Introduction

Heart attacks have been a common death cause since the beginning of our species, and even though medicine has improved with the passage of time, heart disease remains claiming the lives of many people worldwide. With this project, we want to identify the contributing factors of heart attacks, so we can develop a model that classifies whether an individual is at risk of suffering a heart-attack. This might be useful as a prevention mechanism, and may be applied in preventive strategies for medical patients. This study leverages a dataset containing various health, lifestyle, and demographic features to achieve this goal.

The study will apply multiple machine learning strategies in order to classify the binary target variable “heart_attack”: 0 as for no heart attack, and 1 for heart attack. We will focus on uncovering patterns that improve the predictive precision, by exploring features like age, cholesterol levels, smoking status and the occurrence of heart attacks.

2.Description of the original data

Data Source and Data Matrix

This dataset was obtained from a Kaggle repository, where it is stated that it was inspired by real-world trends in Indonesia, and compiled via a combination of statistical research, health surveys, and medical studies. The reference of the source website is on section 7 references.
The data matrix is composed by 158,355 rows and 28 columns, of which 10 are numerical, 10 are categorical, and 8 are binary variables. Each row corresponds to a patient, and each column to an attribute regarding that patient.
The variables that form the data matrix are the following:

- **age**: Numerical (integer), representing the individual's age in years
- **gender**: Categorical, with values "Male" or "Female".
- **region**: Categorical, indicating "Urban" or "Rural" residence.
- **income_level**: Categorical, with levels "Low", "Middle", or "High".
- **hypertension**: Binary (0 or 1), indicating absence or presence of high blood pressure.
- **diabetes**: Binary (0 or 1), indicating absence or presence of diabetes.
- **cholesterol_level**: Numerical (integer), total cholesterol in mg/dL.
- **obesity**: Binary (0 or 1), indicating whether the individual is obese.
- **waist_circumference**: Numerical (integer), in centimeters.
- **family_history**: Binary (0 or 1), indicating family history of heart disease.
- **smoking_status**: Categorical, with values "Never", "Past", or "Current".
- **alcohol_consumption**: Categorical, with levels "None", "Moderate", or "High".
- **physical_activity**: Categorical, with levels "Low", "Moderate", or "High".
- **dietary_habits**: Categorical, with values "Healthy" or "Unhealthy".
- **air_pollution_exposure**: Categorical, with levels "Low", "Moderate", or "High".
- **stress_level**: Categorical, with levels "Low", "Moderate", or "High".
- **sleep_hours**: Numerical (float), hours of sleep per night.
- **blood_pressure_ystolic**: Numerical (integer), systolic BP in mmHg.
- **blood_pressure_diastolic**: Numerical (integer), diastolic BP in mmHg.
- **fasting_blood_sugar**: Numerical (integer), in mg/dL.
- **cholesterol_hdl**: Numerical (integer), HDL cholesterol in mg/dL.
- **cholesterol_ldl**: Numerical (integer), LDL cholesterol in mg/dL.
- **triglycerides**: Numerical (integer), in mg/dL.
- **EKG_results**: Categorical, with values "Normal" or "Abnormal".
- **previous_heart_disease**: Binary (0 or 1), indicating prior heart conditions.
- **medication_usage**: Binary (0 or 1), indicating use of heart-related medications.
- **participated_in_free_screening**: Binary (0 or 1), indicating participation in health screenings.
- **heart_attack**: Binary (0 or 1), the target variable.

Impact of missing values

Our dataset does not present any missing values for any of its variables. Therefore, there is no impact or potential bias to be induced, and for now there is no need for treatment/imputation.

3.Description of pre-processing of data

Pearson Correlations

For this part, we wanted to select the most relevant features, and decide how to deem which ones are not relevant and if we should discard those.

Firstly, since we did not find any specifically redundant feature, we generated a heatmap based on the correlation matrix of all the features to see their degree of multicollinearity.

	age	hypertension	diabetes	cholesterol_level	obesity	waist_circumference	family_history	sleep_hours	blood_pressure_systolic	blood_pressure_diastolic	fasting_blood_sugar	cholesterol_hdl	cholesterol_ldl	triglycerides	previous_heart_disease	medication_usage	participated_in_free_screening	heart_attack
age	1.000000	-0.000512	0.003616	-0.003747	-0.001556	0.001601	0.001046	0.000996	-0.001945	0.001823	0.001721	0.002060	0.000313	0.001071	-0.000475	-0.003072	-0.000620	0.105756
hypertension	-0.000512	1.000000	-0.001329	-0.001085	-0.002454	0.000724	0.000391	0.001773	-0.002323	-0.000049	0.001431	-0.000334	-0.000579	0.000362	-0.004754	0.003349	-0.002905	0.269261
diabetes	0.003616	-0.001329	1.000000	-0.001019	0.001720	-0.001773	-0.001133	0.001436	-0.004286	-0.002648	0.002989	0.000479	0.003871	-0.002776	0.000584	0.002031	-0.002566	0.194512
cholesterol_level	-0.003747	-0.001085	-0.001019	1.000000	-0.000630	-0.000419	-0.002900	0.001566	-0.000771	0.001419	-0.001864	0.003067	0.000929	0.000118	-0.004461	0.003451	-0.000869	0.092611
obesity	-0.001556	-0.002454	0.001720	-0.000630	1.000000	0.395426	0.001415	0.003177	0.001068	-0.000498	0.002241	0.001893	-0.001063	-0.001200	-0.001761	-0.004153	-0.003739	0.171720
waist_circumference	0.001601	0.000724	-0.001773	-0.000419	0.395426	1.000000	0.000109	0.002004	0.000340	0.002217	0.003899	-0.002747	-0.000781	0.000694	0.000844	0.000056	-0.002741	0.067883
family_history	0.001046	0.000391	-0.001133	-0.002900	0.001415	0.000109	1.000000	0.001150	-0.000897	0.002180	0.004453	-0.004747	0.002161	-0.000026	0.001014	-0.000015	0.002573	0.001374
sleep_hours	0.000996	0.001773	0.001436	0.001566	0.003177	0.002004	0.001150	1.000000	-0.000589	0.003410	-0.001069	-0.002888	-0.001024	-0.002672	0.002168	0.003941	-0.002101	0.000673
blood_pressure_systolic	-0.001945	-0.002323	-0.004286	-0.000771	0.001068	0.000340	-0.000897	-0.000589	1.000000	-0.002384	-0.001058	-0.005929	0.001367	-0.003280	0.003781	0.000549	0.001929	-0.001644
blood_pressure_diastolic	0.001823	-0.000049	-0.002648	0.001419	-0.000498	0.002217	0.002180	0.003410	-0.002384	1.000000	-0.000854	0.001408	-0.002969	-0.000498	-0.004296	0.002096	0.000923	-0.000301
fasting_blood_sugar	0.001721	0.001431	0.002989	-0.001864	0.002241	0.003899	0.004453	-0.001069	-0.001058	-0.000854	1.000000	-0.002455	0.002177	0.006049	0.001645	0.003893	-0.002183	0.069826
cholesterol_hdl	0.002060	-0.000334	0.000479	0.003067	0.001893	-0.002747	-0.004747	-0.002888	-0.005929	0.001408	-0.002455	1.000000	-0.001725	-0.000933	-0.000155	0.000224	0.000384	0.000648
cholesterol_ldl	0.000313	-0.000579	0.003871	0.000929	-0.001063	-0.000781	0.002161	-0.001024	0.001367	-0.002969	0.002177	-0.001725	1.000000	-0.002760	0.001334	-0.001587	0.003597	0.000632
triglycerides	0.001071	0.000362	-0.002776	0.000118	-0.001200	0.000694	-0.000026	-0.002672	-0.003280	-0.000498	0.006049	-0.000933	-0.002760	1.000000	0.001814	0.003515	0.000007	-0.000709
previous_heart_disease	-0.000475	-0.004754	0.000584	-0.004461	-0.001761	0.000844	0.001014	0.002168	0.003781	-0.004296	0.001645	-0.000155	0.001334	0.001814	1.000000	0.001303	0.001936	0.274775
medication_usage	-0.003072	0.003349	0.002031	0.003451	-0.004153	0.000056	-0.000015	0.003941	0.000549	0.002096	0.003893	0.000224	-0.001587	0.003515	0.001303	1.000000	-0.002091	0.004694
participated_in_free_screening	-0.000620	-0.002905	-0.002566	-0.000869	-0.003739	-0.002741	0.002573	-0.002101	0.001929	0.000923	-0.002183	0.000384	0.003597	0.000007	0.001936	-0.002091	1.000000	-0.003656
heart_attack	0.105756	0.269261	0.194512	0.092611	0.171720	0.067883	0.001374	0.000673	-0.001644	-0.000301	0.069826	0.000648	0.000632	-0.000709	0.274775	0.004694	-0.003656	1.000000

Top Absolute Correlations

obesity	waist_circunference	0.3954
previous_heart_disease	heart_attack	0.2748
hypertension	heart_attack	0.2693
diabetes	heart_attack	0.1945
obesity	heart_attack	0.1717

If two variables had too much correlation between them, they could affect linear models like SVM (with linear kernels), since it may inflate the variance, and affect the interpretability. Therefore, it might be argued that they might be removed, or rather be merged using PCA.

However, the most correlated pair have less than 0.5 correlation, which would rather be a low-moderate score. As a result, it would not be enough of an argument to discard one of them, or merge them. The other variables have even less correlation, and they are regarding the target variable, so it would not make sense to discard them/treat them.

We considered using MI scores for feature selection, but we did not want to lose tendencies or introduce bias by arbitrarily choosing a number of “k” best scored features. The results of some algorithms could be distorted, so we chose to keep all the features, and if the algorithm required it (like KNN), we would find the best features specifically for it.

Pearson correlation coefficients are also used in the data-quality phase to

- Detect obvious coding errors
- And obtain a first, univariate signal-to-noise estimate before multivariate modelling.

After running the code, we have found out:

- There are no extra correlations between variables.
- Target ‘heart_attack’ has the highest correlation with previous_heart_disease and hypertension. It means that if your body has these kinds of problems, then there is approximately 27% of connection between new heart attack
- Low correlation with heart attack is detected with many columns.

We have not found any outliers which needed to be treated or error values, so there was no need for the removal of any example.

Imputing/removing missing values?

Missing data

There weren’t any missing values in our dataset, therefore there was no need to perform data transformation over the dataset.

Encoding of categorical data

Before encoding the categorical variables, transforming them into binary features for each category, we checked if there were too many unique values for each category:

```
full_data.describe(include=object)
```

	gender	region	income_level	smoking_status	alcohol_consumption	physical_activity	dietary_habits	air_pollution_exposure	stress_level	EKG
count	158355	158355	158355	158355	158355	158355	158355	158355	158355	
unique	2	2	3	3	3	3	2	3	3	
top	Male	Urban	Middle	Never	None	Low	Unhealthy	Moderate	Moderate	
freq	82243	103038	71230	79183	94848	63417	95030	79144	79366	

Since there weren't too many (which could increase the computational cost), we decided to apply the encoding to all of them via *get_dummies*, while dropping one of the categories for each feature (which can be reconstructed implicitly –when all the other categories have a zero–)

Original:

smoking_status	alcohol_consumption	physical_activity	dietary_habits	air_pollution_exposure	stress_level
Never	None	High	Unhealthy	Moderate	Moderate
Past	None	Moderate	Healthy	High	High
Past	Moderate	Moderate	Healthy	Low	Low
Never	Moderate	Moderate	Unhealthy	Low	High
Current	Moderate	Moderate	Unhealthy	High	Moderate
Past	None	Low	Unhealthy	Low	Moderate
Never	Moderate	Moderate	Healthy	High	High

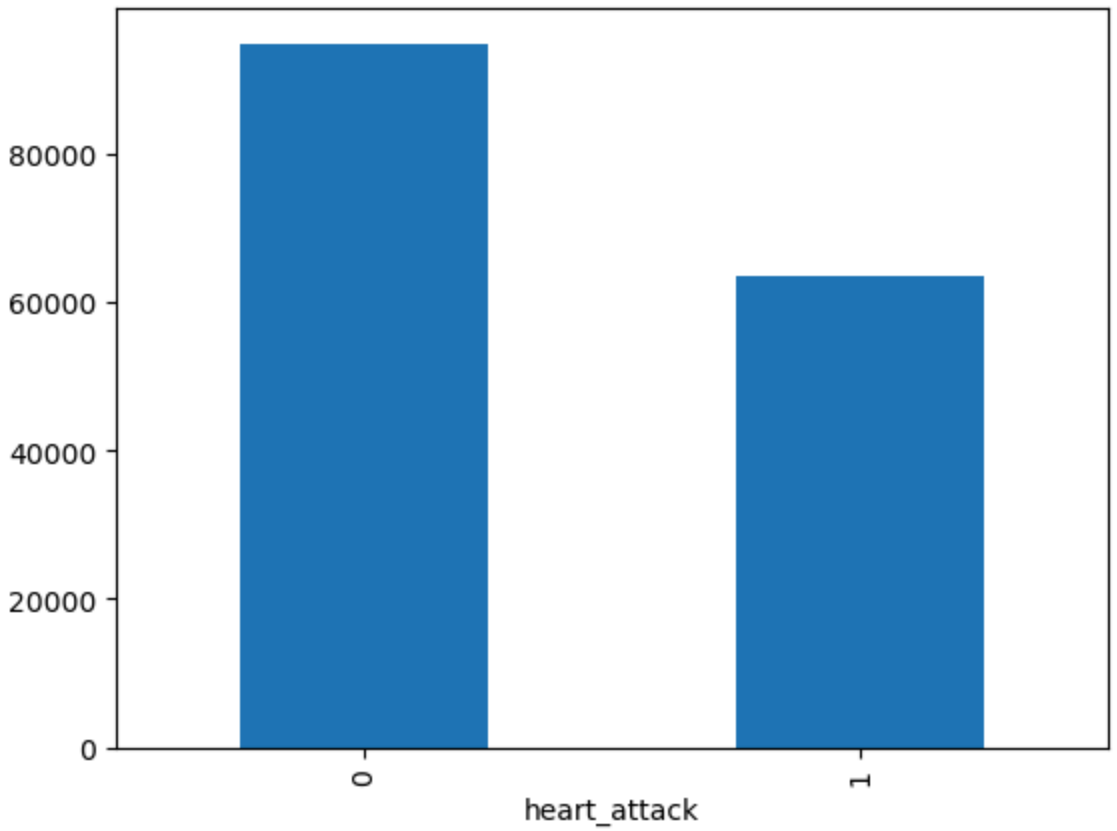
New dataset:

income_level_High	income_level_Low	income_level_Middle	smoking_status_Current	smoking_status_Never	smoking_status_Past	alcohol_consumption_High
0	0	1	1	0	0	0
0	1	0	0	1	0	0
0	0	1	0	1	0	0
1	0	0	0	1	0	0
0	0	1	1	0	0	0
0	0	1	0	1	0	0

Some examples were removed?

Data balancing

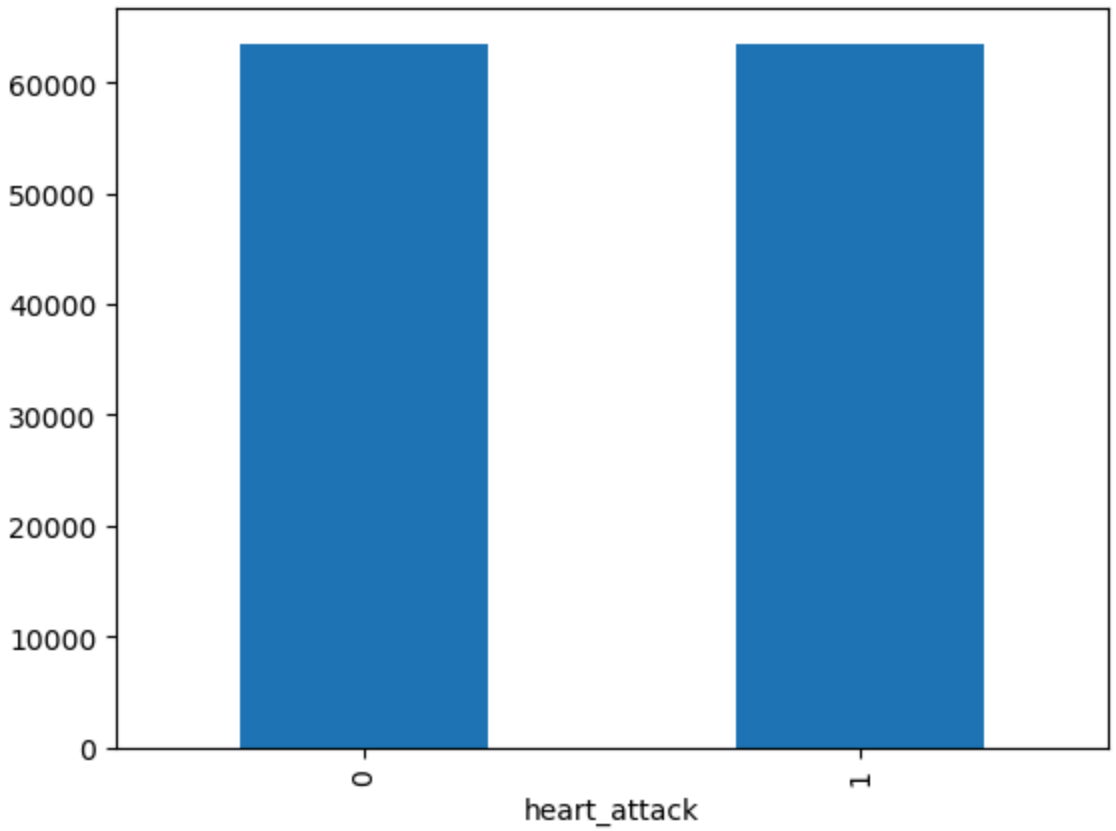
Our targets variables were unbalanced: approximately 60% of the rows were of class 0 on the target:



heart_attack
0 94854
1 63501

We decided to apply the undersampling method, since we wanted to improve the scores of the minority class, at the cost of losing information regarding the majority class. After all, our main priority is to ensure that true heart attacks are always detected, at the cost of more false positives. In addition, reducing the samples of the majority class improves the computational time. Specifically, we applied the Instance Hardness Method, since it retains the observations with the highest probability of being correctly classified for the class to be undersampled.

The result was the following:



heart_attack
0 63529
1 63501

After the undersampling, 31325 values from class 0 were removed from the dataset.

Normalization?

Value standardization

In order to give ensure that data is in a consistent and uniform format, so that the features scales do not skew the results (ensure same relevance to each feature) in algorithms that depend on distances, like K-NN.

age	hypertension	diabetes	cholesterol_level	obesity	waist_circumference	family_history
-0.12862391143930751	-0.6501202543695714	-0.495921730539036	-1.0309846769852582	1.733856953701036	0.3462521357657632	-0.6459200402624914
-0.6325209233787438	1.5381000565344054	-0.495921730539036	0.7964208997870171	-0.5767200101862721	0.40772938014021076	1.5481018356291234
-1.13641793531818	-0.6501202543695714	-0.495921730539036	-1.0059517238787887	1.733856953701036	2.3750012001225316	-0.6459200402624914
1.8029813009951983	1.5381000565344054	-0.495921730539036	1.9729696957910847	-0.5767200101862721	-1.1292017292209775	-0.6459200402624914
1.2151014537325227	-0.6501202543695714	-0.495921730539036	-0.1548313182588248	-0.5767200101862721	-0.8832927517231873	-0.6459200402624914
0.03934175920717125	-0.6501202543695714	-0.495921730539036	-0.355094943110581	1.733856953701036	0.40772938014021076	-0.6459200402624914

4.Evaluation criteria of data mining models

Description of the procedure followed in order to obtain a representative validation dataset and description of the method that will be used to evaluate the different data mining models. This includes a description of the parameters used in the evaluation (Cross-validation? K-fold cross-validation? How many folders? Why that number?) and discussion of the metric used for evaluation (accuracy, f1, etc). Which is the splitting procedure of data set into train and validation data set? Description of the procedure that you have followed to obtain a representative validation data set

Which is the splitting procedure of data set into train and validation data set? Description of the procedure that you have followed to obtain a representative validation data set

---

Representative validation dataset

Once we have completed all the preprocessing steps, we have obtained a balanced and standardized version of the initial data, which will be the one used for training the models for each algorithm covered in this work. We decided to keep all the dataset's samples, in order to preserve their statistics and tendencies of the variables. We know that this will have an important impact on the computational load, but the samples we gathered did not represent the original distributions.

Consequently, we have decided to divide the half of the dataset for training, and the other half for testing. This way, we minimize the computational capacity in training and increase the statistical trustiness in data evaluation, while assuming more cost in inference.

Cross-validation? K-fold cross-validation? How many folders? Why that number?

Parameters used in the evaluation

Following the previous argument regarding the notably large dataset, we have decided to principally use cross validation for all the algorithms. In the least demanding algorithms, we have also tried using 5-fold cross validation for most of the algorithms, since it's a commonly used value, which balances bias and variance, but still demands more computation the more folds there are.

discussion of the metric used for evaluation (accuracy, f1, etc)

Metric used for evaluation

The main metric used for evaluation will be the recall of the models after being trained. The reason behind this choice is that we want to ensure that the true positives, which are the people that actually suffer from a heart attack, are predicted and diagnosed successfully. Due to the consequences of missing a fatal diagnosis, we can assume a reduction on accuracy, or some false positives (as false alarms are far less tragic than deaths).

5.Execution of different machine learning methods

For each method, you should describe how you adjusted parameters for the algorithm and how did you perform an evaluation of the method. You have to report also the performance of the model on the validation dataset. In addition, for each method you should describe and discus some particular issues:

• Naïve Bayes: Think about hypothesis of independence of variables. Do you have enough number of elements to obtain reliable probabilities? Keep that information for the discussion section.

• K-NN: Description of procedure followed for choosing the best k-parameter. Show a graph with k's evolution. Have you adjusted other parameters as distance measure? Have you considered removal of irrelevant features if accuracy is poor compared with other approaches? (remember that k-nn is sensible to irrelevant features when computing distance to closest examples).

• Decision Trees: Discussion of choice of parameters used. Try to interpret the obtained DT using some examples of the validation set. Show some of the most relevant rules. Discuss how + and – examples are mixed in leaves in order to estimate the reliability of the tree.

• Support Vector Machines: Discuss choice of kernel and parameters used. Did you run any method to speed the building of the model? Report number of supports of the selected machine and try to interpret why the kernel selected and parameters selected for the final run give you the best results for your dataset. Try also to inspect main supports of your machine.

• Meta-learning algorithms: Performance Majority Voting, Bagging, RandomForest and Adaboost. Explain parameters selected for each algorithm

---

5.1.Naïve Bayes

Think about hypothesis of independence of variables. Do you have enough number of elements to obtain reliable probabilities? Keep that information for the discussion section.

"Gaussian Naïve Bayes is a type of Naïve Bayes method working on continuous attributes and the data features that follows Gaussian distribution throughout the dataset." Says Geeks for Geeks [https://www.geeksforgeeks.org/gaussian-naive-bayes/].

To start with the algorithm, we need to modify dataset values to Gaussian format. Therefore, we have applied steps in preprocessing to create dataset like this. Below is an example of these values:

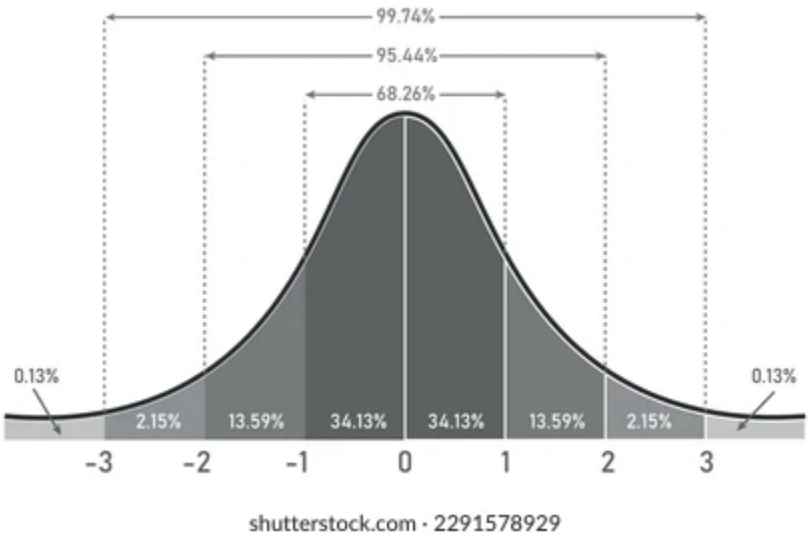
Gaussian Naive Bayes

```
import ...

url = "./data_balanced_normalized.csv"
df = pd.read_csv(url)
df.head()
✓ [4] 264ms
```

5 rows × 5 rows × 34 cols — Edit in Data Wrangler — AI Quick Charts

	age	hypertension	diabetes	cholesterol_level	obesity	waist_circumference	family
0	-0.191687	-0.611612	-0.473831	0.819943	1.807022	-0.733750	
1	-0.445728	-0.611612	-0.473831	-0.325946	-0.553397	1.350383	
2	0.231714	-0.611612	-0.473831	0.896336	-0.553397	-1.408028	
3	-1.885292	-0.611612	-0.473831	-0.147697	-0.553397	-1.162836	
4	-0.530408	-0.611612	-0.473831	-1.650085	-0.553397	-0.978942	

Now, values look like this. We have many average values and some outside values.



Features and predictors

In the next step, we have created two arrays. First array X contains features and predictors. We have dropped our target column so it does not affect results. It contains all other columns except heart_attack information. Array Y contains values of the result of heart attacks. No other columns.

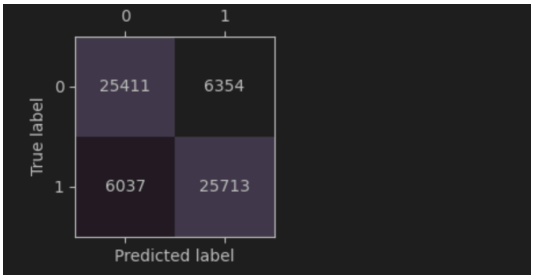
```python
from sklearn.model_selection import train_test_split

X = df.drop(columns=["heart_attack"], axis = 1).values
y = df["heart_attack"].values.astype(np.int8)

print(X.shape)
print(y.shape)
print(f"Dataset: {X.shape}  |  positives {(y==1).sum()}")

# Train-test split on the sampled data (BEFORE preprocessing)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.5, stratify=y, random_state=1
)

✓ [160] 31ms
```

```
(127030, 33)
(127030,)
Dataset: (127030, 33)  |  positives 63501
```

After we create arrays, we display information about those arrays. We can see that dataset X contains 127 030 rows and 33 columns, and array Y contains 63 501 positive heart attacks.

After the check of the dataset, we split it into Train and Test sets. They are approximately the same size.

Initial Cross Validation




```
0.8098582677165355
              precision    recall  f1-score   support

           0       0.81      0.80      0.80     31765
           1       0.80      0.81      0.81     31750

    accuracy                           0.80     63515
   macro avg       0.80      0.80      0.80     63515
weighted avg       0.80      0.80      0.80     63515
```

Recall: 0.81.
Threshold: 0.5

As we can see from the confusion matrix and the classification report, most of the metrics are good, with values around 0.80. Our main metric Recall is also good. We want to maximize the recall as much as possible. Current recall value is 0.81 which is fairly good and can detect 4 out of 5 heart attacks.

In the next part, we will find the best threshold for GNB.

Adjusting threshold

```python
kf = StratifiedKFold(
    n_splits = 20,
    shuffle = True,
    random_state = 42
)

# Now we compute the threshold by iterating the data we have
for train_index, test_index in kf.split(X_train, y_train):
    # Use X_train instead of X
    X_train2, X_test2 = X_train[train_index], X_train[test_index]
    y_train2, y_test2 = y_train[train_index], y_train[test_index]

    clf.fit(X_train2, y_train2)
    probs = clf.predict_proba(X_test2)
    ProbClass1 = probs[:,1]  # This should work now as GaussianNB always returns probabilities for both classes

    # Sort probabilities and generate pairs (threshold, f1-for-that-threshold)
    res = np.array([[th,f1_score(y_test2,filterp(th,ProbClass1),pos_label=1)] for th in np.sort(ProbClass1)])

    # Uncomment the following lines if you want to plot at each iteration how f1-score
    # evolves increasing the threshold
    plt.plot(res[:,0],res[:,1])
    plt.show()

    # Find the threshold that has maximum value of f1-score
    maxF = np.max(res[:,1])
    pl = np.argmax(res[:,1])
    optimal_th = res[pl,0]

    # Store the optimal threshold found for the current iteration
    lth.append(optimal_th)

# Compute the average threshold for all 10 iterations
threshold = np.mean(lth)
print(f"Selected threshold in 10-fold cross validation: {np.round(threshold, 6)}")
```

```
[10] 1m 30s
```

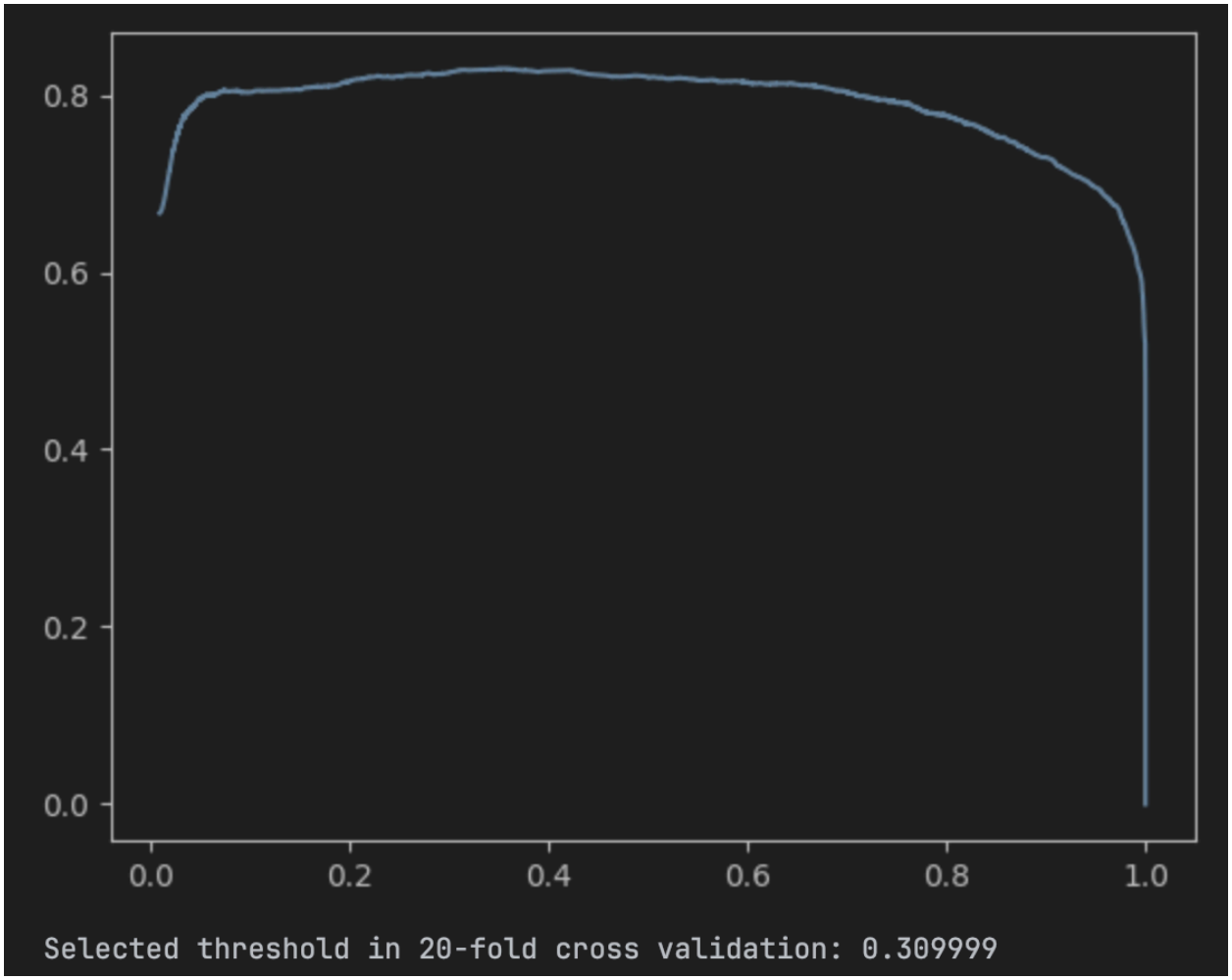
In **Threshold tuning methodology,** we have replaced the default 0.50 decision rule of the Gaussian Naive Bayes classifier with an optimized probability threshold. We split the dataset into 50% training data and 50% test data. Then we applied the following:

Applied steps:

1. **Stratified K-fold CV (20x)** on the training partition preserves class balance in every fold.
2. For each fold we:
   - trained NB on 50 % of the rows

- predicted class-1 probabilities on the 50 % hold-out
- evaluated the **F1-score** for every unique probability produced by the model
- stored the threshold that maximized F1 in that fold.

3. The final threshold is the **mean** of the 20 optimal thresholds: **0.31**

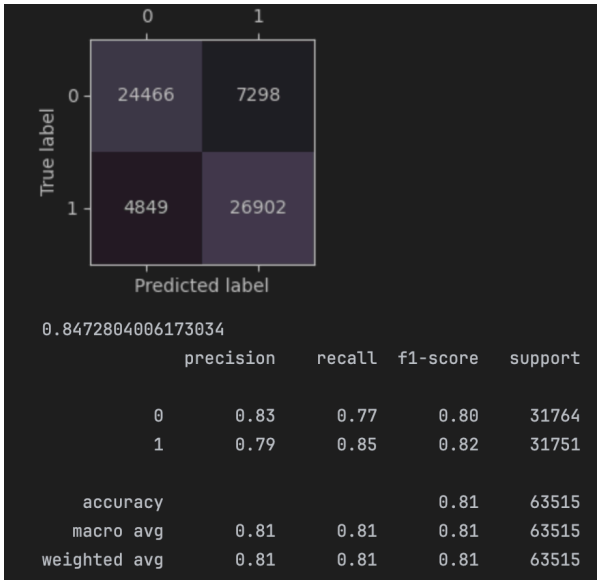


- X-axis (0 - 1) – the probability threshold that decides whether a patient is labelled heart-attack
- Y-axis – the resulting F1-score (harmonic mean of precision & recall) that the model achieves on the validation fold when that threshold is used

Optimal threshold: 0.31
Our optimal point is between 0.05–0.6. That's where our **threshold 0.31** was selected.

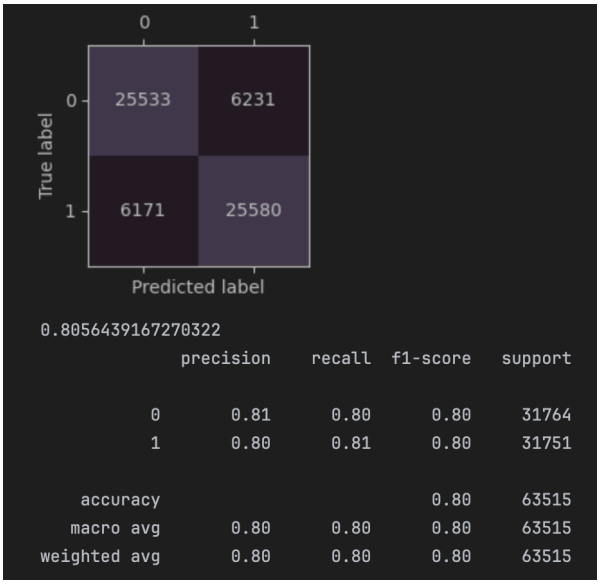
The threshold–F1 curve in the bow chart peaks near this value for every fold, confirming the stability of the optimum.

Final Validation New vs Old



0.8472804006173034

	precision	recall	f1-score	support
0	0.83	0.77	0.80	31764
1	0.79	0.85	0.82	31751
accuracy			0.81	63515
macro avg	0.81	0.81	0.81	63515
weighted avg	0.81	0.81	0.81	63515



0.8056439167270322

	precision	recall	f1-score	support
0	0.81	0.80	0.80	31764
1	0.80	0.81	0.80	31751
accuracy			0.80	63515
macro avg	0.80	0.80	0.80	63515
weighted avg	0.80	0.80	0.80	63515

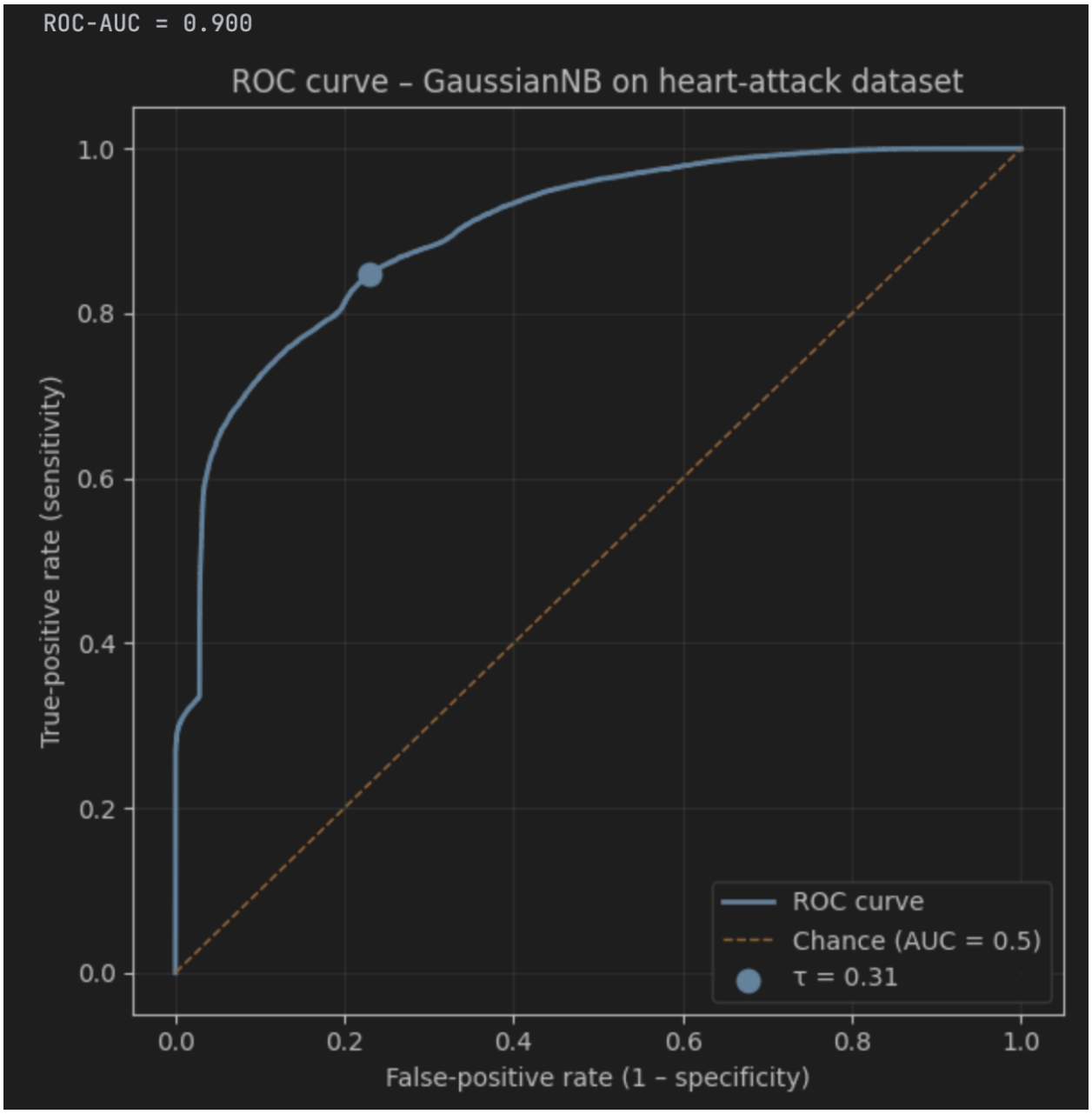
First image displays new **threshold 0.31** and **second** image **threshold 0.50.**

Using this tuned cut-off improved recall by 4 percents on the test set while reducing precision by 1 percent, raising the overall **recall from 0.81 to 0.85 %**.

Varying the probability cut-off shows an F1-score maximum of = 0.81 at a threshold around 0.31. Thresholds below 0.05 over-predict positives and lose precision, while thresholds above 0.65 under-predict and miss true cases, with performance dropping sharply beyond 0.8.

- With the tuned 0.31 threshold, the model behaves like an aggressive screener. It catches 4 out of 5 real heart-attack cases, but triggers a fair number of false alerts.
- With the default 0.50 threshold, the model is conservative. Fewer false alarms and better overall accuracy, but it misses almost 4 in 10 genuine cases.

ROC-AUC Curve



- **Blue Curve** – plots the True-Positive Rate (sensitivity / recall) on the y-axis against the False-Positive Rate (1 – specificity) on the x-axis as we sweep the decision threshold from 0 to 1.
- **Diagonal** – the "no-skill" reference; a model that guesses at random would trace this 45° line and have an AUC of 0.50.
- **Shaded area under the curve (AUC = 0.9)** – summarizes the model's ranking power over all thresholds 1.0 is perfect separation, 0.5 is chance. An AUC of 0.9 indicates strong discriminative ability.
- **Marker ● at t ≈ 0.31** – the probability threshold selected during 20-fold cross-validated F-score tuning. At this operating point, the model achieves:
  - **Recall / TPR ≈ 0.85** – it detects 85 % of actual heart-attack cases

The sharp rise on the left and the gentle roll-off near the top show that the classifier can maintain high sensitivity while holding the false-alarm rate below 30 %. Shifting the threshold left would further increase recall at the cost of more false positives, shifting right would do the opposite.

Conclusion

Applying the cross-validated **threshold** of **0.31** raised recall for the positive class from **0.81 to 0.85** and its F1-score from **0.80 to 0.82**. This comes at the cost of overall accuracy and a drop in class-0 recall.

The tuned model is therefore recommended for high-recall scenarios such as clinical pre-screening, while the default 0.50 threshold remains preferable when minimizing false alarms is more important.

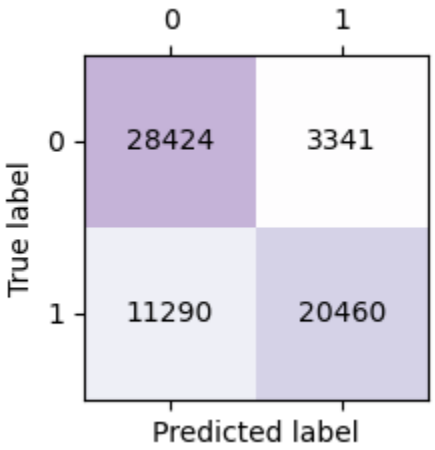
5.2.K-NN

K-NN: Description of procedure followed for choosing the best k-parameter. Show a graph with k's evolution. Have you adjusted other parameters as distance measure? Have you considered removal of irrelevant features if accuracy is poor compared with other approaches? (remember that k-nn is sensible to irrelevant features when computing distance to closest examples).

This method requires a normalized and clean datasets in order to train the models correctly. For that, we will begin training the models with the dataset we obtained after applying the preprocessing steps (which include standardization and balancing), and we will study how to improve the results by applying feature selection, and finding the best parameters. For this model, we will use 5-fold cross validation, since it presents a reasonable tradeoff between variance and bias, and to keep consistency with the majority of the models, which will also use that value.

Initial cross validation

The first thing we did after loading the data was applying 5-fold cross validation to the dataset, to see the initial performance obtained of applying K-NN to them before hyperparameter tuning.



- Recall for class 1: 0.64

	precision	recall	f1-score	support
0	0.72	0.89	0.80	31765
1	0.86	0.64	0.74	31750
accuracy			0.77	63515
macro avg	0.79	0.77	0.77	63515
weighted avg	0.79	0.77	0.77	63515

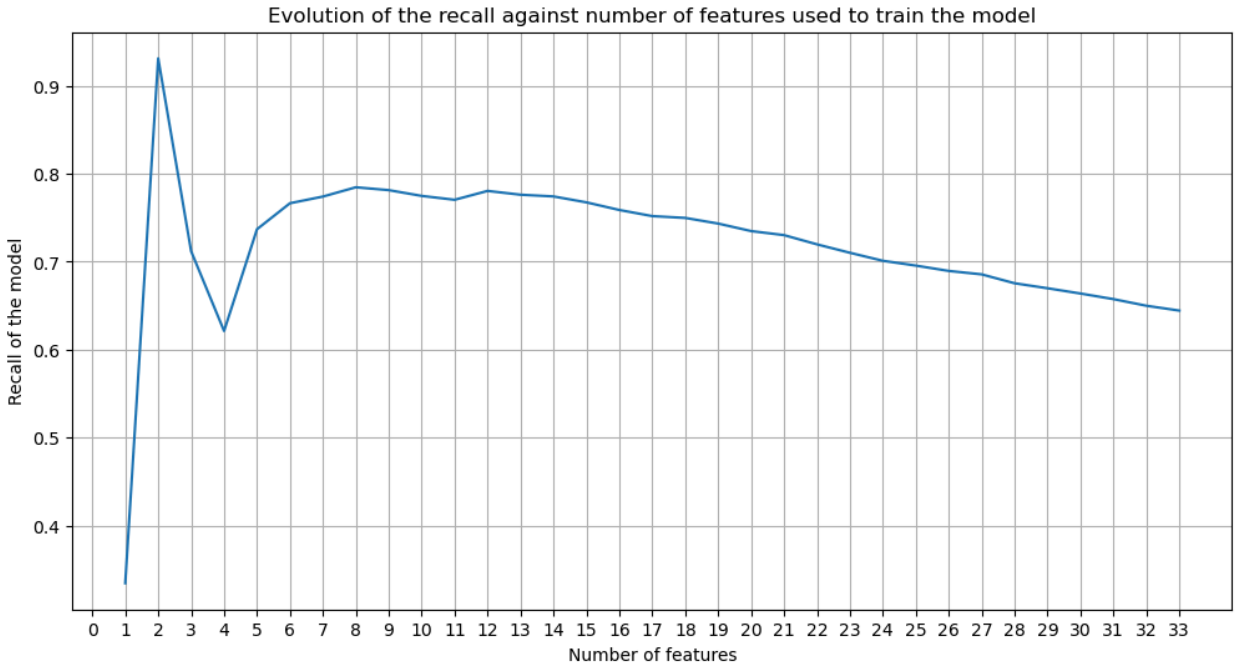
As we can see from the confusion matrix and the classification report, most of the metrics are fairly good, with their values surpassing 0.70, except for the recall for class which stays at 0.64. This is not what we want, since it is the main metric we want to maximize, and currently the model detects most of the people which will certainly not have a heart attack (recall of class 0 = 0.89), at the expense of having a 0.36 true positive rate, which might cause many misdiagnoses. Therefore, it is clear that we will have to tune the hyperparameters until we improve the model for our medical purpose.

Hyperparameter tuning

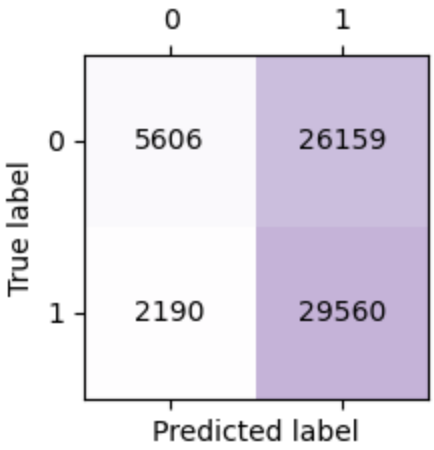
Have you considered removal of irrelevant features if accuracy is poor compared with other approaches?

As we previously mentioned in the preprocessing section, some models like K-NN are affected by irrelevant features, so we decided to apply feature selection as the first step in our process of hyperparameter tuning.

We applied 5-fold cross validation with different values for k as the number of columns with the best MI scores. The results were the following:



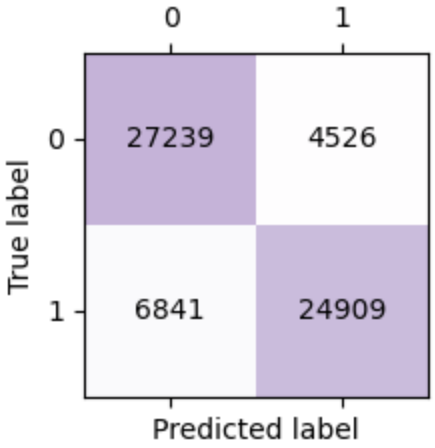
This plot shows that the best recall for class 1 is obtained when selecting the k best features = 2.



	precision	recall	f1-score	support
0	0.72	0.18	0.28	31765
1	0.53	0.93	0.68	31750
accuracy			0.55	63515
macro avg	0.62	0.55	0.48	63515
weighted avg	0.62	0.55	0.48	63515

We can observe that the recall for class 1 is of 0.93 which is really. However, the other metrics are affected very negatively by the removal of the other features, leading us to a 0.82 false positive rate (0.18 recall for class 0). We consider that it is too much, as we want to maximize the detection of people who have a predisposition to suffering a heart attack, but we would not want our model to diagnose almost everyone just to be safe (too radical, the model would not be reliable at all). That might lower the trustiness of our model and waste resources from the people who have an actually true tendency.

Consequently, we decided to choose the second-highest peak of recall, which is obtained when selecting the eight best features. We obtained the following confusion matrix:



	precision	recall	f1-score	support
0	0.80	0.86	0.83	31765
1	0.85	0.78	0.81	31750
accuracy			0.82	63515
macro avg	0.82	0.82	0.82	63515
weighted avg	0.82	0.82	0.82	63515

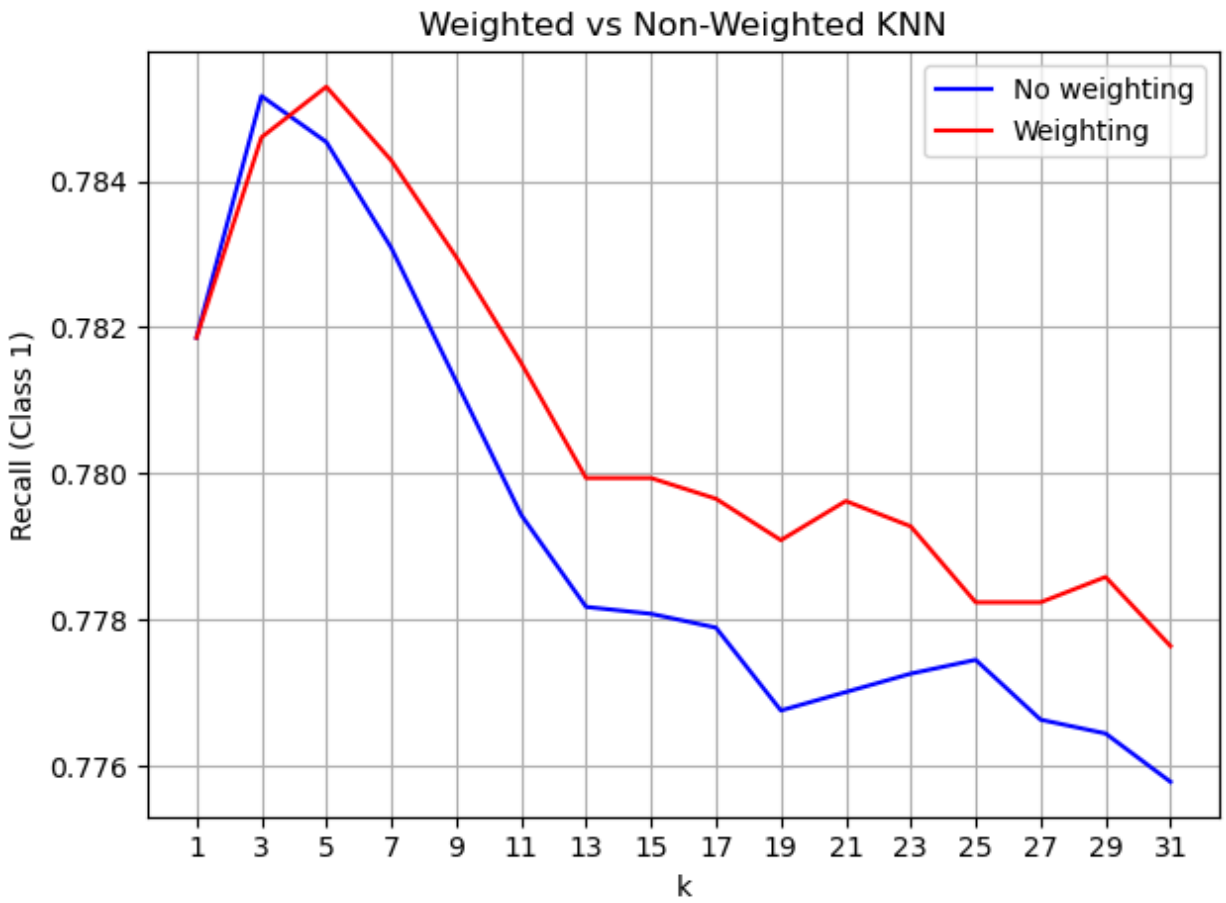
It is true that we now get a lower recall for class 1 (which is not bad, almost reaching 0.8), but the other metrics are much better, with all of them surpassing the 0.8 value. Furthermore, the recall for class 1 has a 14% improvement compared to the initial one.

Choosing the best k-parameter

Finding the best parameters

Now that we have finished with feature selection, we are going to look for the best parameters for the K-NN algorithm.

For that, we will apply cross validation within a range of different values for the neighbors, to see how much recall is obtained both when applying bigger weighting to the closest distances, and when applying uniform weighting.



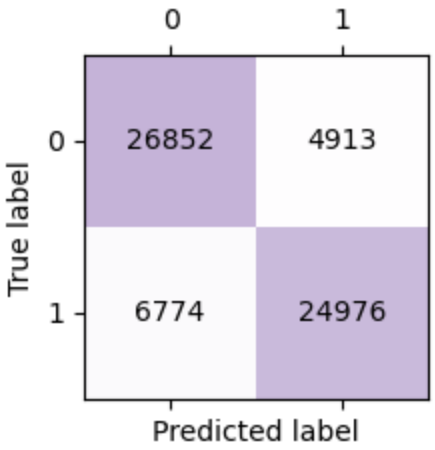
We can see similar slopes for both unweighted and weighted K-NN, with the latter returning a slightly better recall, having its peak at k = 5 neighbors.

In order to corroborate the best k value, and find the other best parameters for K-NN we applied GridSearchCV for the previous range of neighbors, weighted and unweighted distances and a “p” range of [1, 2] (1 as Manhattan distances, and 2 as Euclidean distances). We did not amplify the range for “p”, due to the already high cost on computational time, which would increase even more.

The results were the following:

- N neighbors: 5
- p: 1 (Manhattan distances)
- Recall (Class 1): 0.79

We plotted its confusion matrix and classification report to visualize the achieved scores:

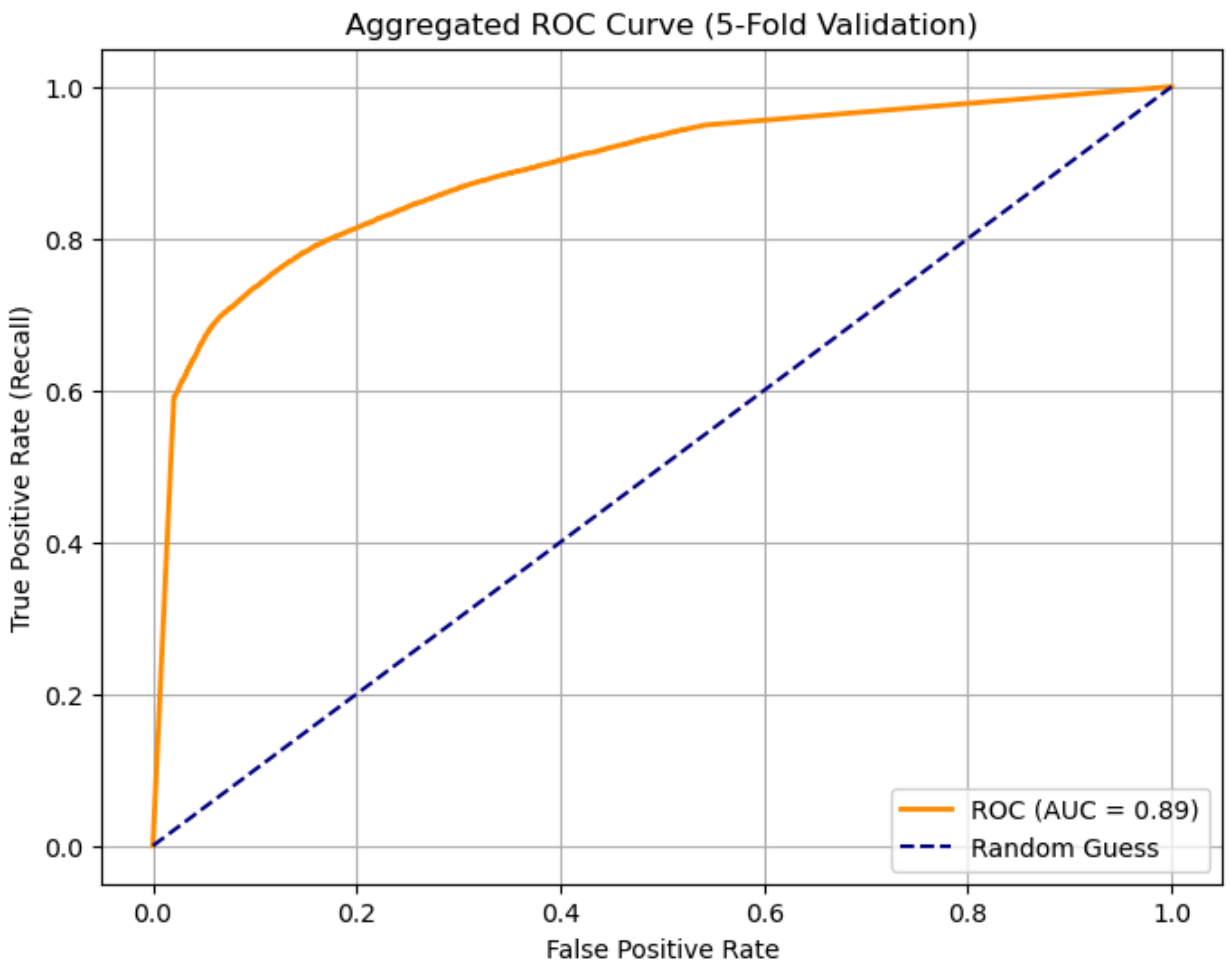


	precision	recall	f1-score	support
0	0.80	0.85	0.82	31765
1	0.84	0.79	0.81	31750
accuracy			0.82	63515
macro avg	0.82	0.82	0.82	63515
weighted avg	0.82	0.82	0.82	63515

The results are almost the same as the ones we previously obtained, since the previous model were also trained with 5 n neighbors (the default value), and the slight improvements are due to the better “p” distance selected.

Finding the best threshold

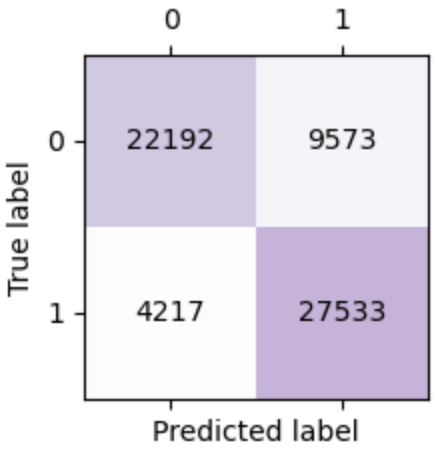
Now that we have found the best parameters for the algorithm, we want to see if we can improve the true positive rate while maintaining fairly good metrics. For that, we plotted the ROC curve:



Since the mean Area Under the curve is of the 5 folds is of 0.89, which is higher than the diagonal "chance line" of 0.5, we can confirm that our model is better than random guessing.

From looking at the curve, we can see the trade-off between true and false positive rate values. As we want to maximize the recall (true positive rate), we will lower the threshold used by the classifier. This way, patients with a smaller predicted probability of suffering a heart attack will be classified as true cases (at the expense of more people with no heart disease being misclassified).

Specifically, we considered that adjusting the threshold to 0.3 raised the class 1 recall to 0.87, while having little impact on the accuracy and precision, and an acceptable FPR. It is displayed both in the ROC curve and the following classification report, that when applying this threshold we get a TPR of 0.87 (87% of heart attacks detected) and an FPR of 0.3 (30% of patients flagged correctly), which are quite good results.

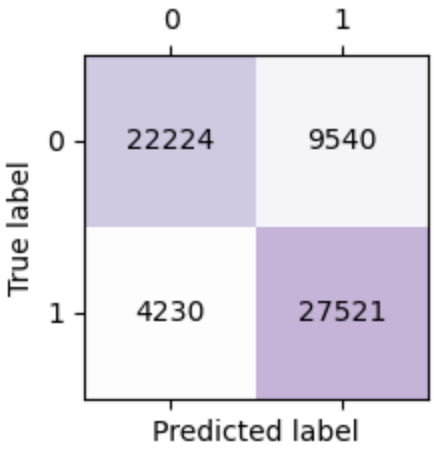


	precision	recall	f1-score	support
0	0.84	0.70	0.76	31765
1	0.74	0.87	0.80	31750
accuracy			0.78	63515
macro avg	0.79	0.78	0.78	63515
weighted avg	0.79	0.78	0.78	63515

Having 30% of false positive rate is an acceptable overhead for the greater good of avoiding deaths, and we fear that lowering the threshold more and therefore amplifying the FPR, our model would not be seen as reliable by the public. Even worse, resources, time and professionals could be wasted on people who do not actually need medical attention (we also think patients do not specially like being misdiagnosed with such a severe condition).

Final validation on test set

Now that we had optimized all the parameters for the algorithm, and the threshold used by the classifier, we moved on to train the final model on the complete training set (with the best features we previously selected). After that, we proceeded to apply the prediction on the test set, and we got the following results:



	precision	recall	f1-score	support
0	0.84	0.70	0.76	31764
1	0.74	0.87	0.80	31751
accuracy			0.78	63515
macro avg	0.79	0.78	0.78	63515
weighted avg	0.79	0.78	0.78	63515

The results are almost identical to the ones obtained through cross validation after hyperparameter and threshold tuning. This indicates that the overfitting is minimal or almost not existent.

5.3.Decision Trees

Discussion of choice of parameters used. Try to interpret the obtained DT using some examples of the validation set. Show some of the most relevant rules. Discuss how + and – examples are mixed in leaves in order to estimate the reliability of the tree.

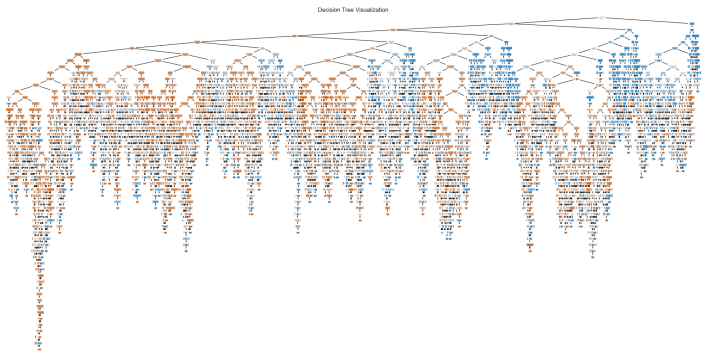
The third machine learning method applied to our dataset is *Decision Trees* located in *decisionTrees.ipynb*. We are going to work with the preprocessed data in *data_balanced_normalized.csv* file.

Firstly, for this method, we are going to make use of ***Cross-Validation*** to separate the data into train/test split. In our case, we wanted to use the standard of 70/30 or 80/20 train/test split, but as we have **127 030 samples** in our data frame, training 70% of that would lead to a very high computational cost. In fact, at the beginning of the project we set 70/30, and we had to wait more than 1 hour to run 1 cell of our notebook. So, for the sake of learning and efficiency, we decided to sacrifice a bit of the training dataset and change to **50/50 of train/test split**. For the same reason, we are going to set **cv = 5** for balancing computational costs.

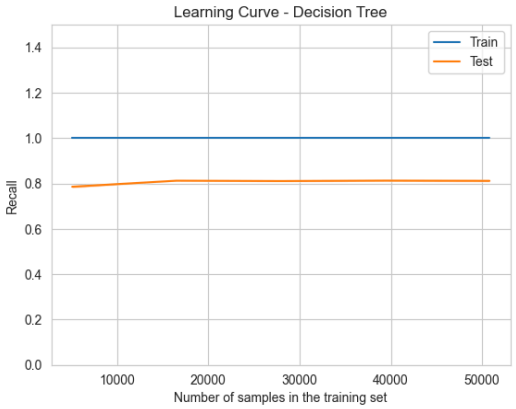
For training the decision tree, we have to set some **hyperparameters**. In order to find the **best parameters** for our decision tree to perform better, we used **GridSearchCV** from the scikit-learn (Python) package. GridSearchCV tests all possible combinations of hyperparameters, it uses K-fold cross-validation to evaluate each combination's performance and returns the hyperparameters that give the highest score (Recall in our study because we want to focus on a high TRP). So, in GridSeachCV we tried the following parameters:

Parameter	Options	Description	Best
criterion	'Gini' and 'entropy'	Measure how pure the nodes in the tree are	'entropy'
min_impurity_decrease	(0, 0.5, 21)	Control the size of the tree	0.0 → tree will split even with small improvements → very deep
min_sample_split	(2, 20, 2)	Minimum number of samples needed to split an internal node	2 → tree with a lot of splits

Using those optimal parameters, we made our initial Decision Tree.



This is the plotted tree executed with the best parameters found by GridSearchCV. We are focused on finding the best recall possible because we want to predict a heart attack diagnosis, so we want the best performance possible and a high recall. Seeing the plotted tree, we can clearly see that the interpretation of our decision tree will be very difficult, there are a lot of leaf nodes and the tree is very deep and wide. We cannot conclude why a prediction is made due to the complexity of the tree. Even so, we are going to validate our decision tree with the learning curve. We used LearningCurveDisplay, learning_curve from the sklearn package.



Viewing this learning curve, we clearly understand that our model is **overfitting** because in train data stays constant at ~1.0 and that could mean that our model has predicted or memorized the data behavior rather than generalizing patterns. The model's performance on unseen data is significantly worse than its performance on the training data.

In conclusion, our decision tree is very **complex** that surely it creates a lot of nodes that at the end what they are doing is adapting to our dataset, memorizing every sample of train set. But there is a solution to study our decision tree and prevent overfitting, that is using ensemble methods. We decided to focus on two ensemble methods: **Random Forest** and **Ada Boost.** The justification for only selecting these two classifiers is their different approaches to building an ensemble and addressing overfitting, which gives us a good look at how ensembles can handle our problem.

We are going to use a decorator. Using the package sklearn.ensemble we are going to execute the following methods: RandomForestClassifier, AdaBoostClassifier. In every method, they are going to

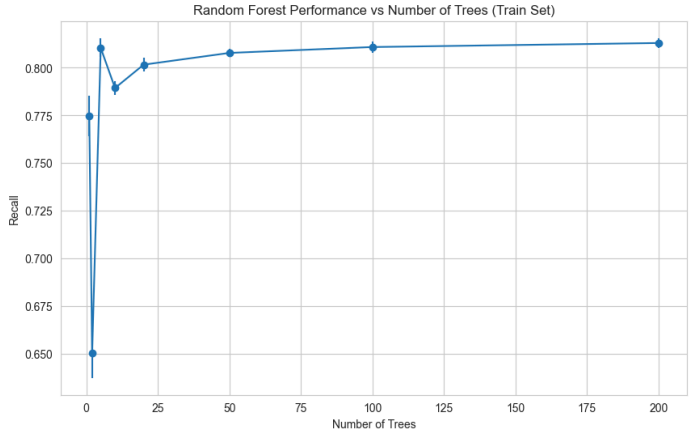
work with a reduced sample of our dataset and calculate then the mean recall.
In essence, Random Forest builds many diverse, independent trees and averages their result, while Ada Boost trains a sequence of trees, each one specializing in the mistakes of its predecessors. This fundamental difference in how they learn and combine models leads to distinct strengths and weaknesses in their performance characteristics.

The results of each method are:

- **Random Forest**

The logic in Random Forest is building multiple decision trees (a forest) and merging their outputs for a final prediction (Recall for us). So firstly we tried hyperparameter tunning with **cross_val_score** with multiple estimators to understand the **computational cost** (which is a very important aspect for model selection), and we used the decorator provided**.** And secondly, we tried hyperparameter tunning with **gridSearchCV** to prevent **overfitting** and be able to generalize.

The following figure shows the performance of recall in relationship with the number of estimators. We used the function *execute_random_forest_with_different_estimators_and_cv()* to show the effect of increasing the number of trees.



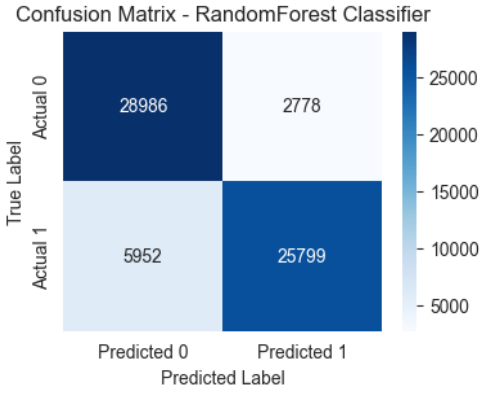

```
Recall: 0.7748 ± 0.0105 [n° estimators (trees): 1]
Time: 0.49028730392456055 [n° estimators (trees): 1]
Recall: 0.6503 ± 0.0130 [n° estimators (trees): 2]
Time: 0.7382822036743164 [n° estimators (trees): 2]
Recall: 0.8105 ± 0.0049 [n° estimators (trees): 5]
Time: 0.9300532341003418 [n° estimators (trees): 5]
Recall: 0.7894 ± 0.0036 [n° estimators (trees): 10]
Time: 1.6292669773101807 [n° estimators (trees): 10]
Recall: 0.8016 ± 0.0036 [n° estimators (trees): 20]
Time: 2.8151142597198486 [n° estimators (trees): 20]
Recall: 0.8078 ± 0.0021 [n° estimators (trees): 50]
Time: 7.201610803604126 [n° estimators (trees): 50]
Recall: 0.8109 ± 0.0030 [n° estimators (trees): 100]
Time: 14.195285558700562 [n° estimators (trees): 100]
Recall: 0.8130 ± 0.0026 [n° estimators (trees): 200]
Time: 28.726196765899658 [n° estimators (trees): 200]
```

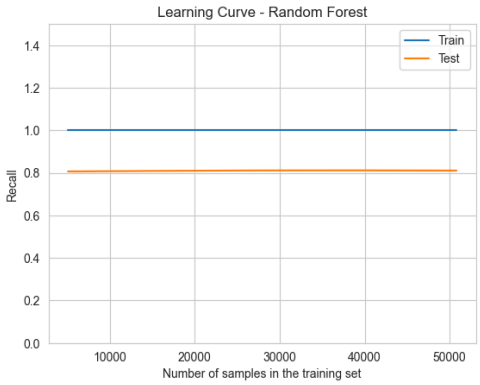
As the number of estimators increases, the standard deviation (SD) decreases and the mean recall improves. This indicates that having multiple trees makes our model more generalizable. With more trees, the model can predict more true positives (recall). Also, the decrease in the SD indicates that having more trees leads to better performance and more reliability. For instance, a single tree is very sensitive to training data, so we have higher variance. Additionally, we can consider a trade-off between computational cost and performance. We see that the optimal performance number of estimators here are **50 estimators,** having the lowest SD and a high mean recall. We also observe that the mean does not increase much between 50, 100, and 200 estimators, although the computation time does.

Random Forest has other hyperparameters that also impact overfitting and performance. That is why we are going to run GridSearchCV with the following parameters.

Parameter	Option	Description	Best
n_estimators	[30, 50, 80]	Range around sweet point, 50	80
max_features	'sqrt', 'log2', 0.5, 0.7	The number of features looking for the best split	‘sqrt’
max_depth	None, 10, 20	Maximum depth of individual trees.	None

```
Time RandomForest GridSearchCV: 472.01 seconds
Best Params = {'max_depth': None, 'max_features': 'sqrt', 'n_estimators': 80}  -  Recall = 0.8108661417322836
RandomForest Accuracy: 0.8626
RandomForest Recall: 0.8125
RandomForest Precision: 0.9028
RandomForest F1-score: 0.8553
RandomForest Classification Report:
              precision    recall  f1-score   support

           0       0.83      0.91      0.87     31764
           1       0.90      0.81      0.86     31751

    accuracy                           0.86     63515
   macro avg       0.87      0.86      0.86     63515
weighted avg       0.87      0.86      0.86     63515
```





We confirm that the sweet spot was in a 50-100 range, so **80** was the best parameter, where it performs very well and does not have much computational overhead. Having **‘sqrt’** as max_feature mean that the individual trees are diverse, so there is less correlation between them. And max_depth of **None** means that we will have better performance if we let the trees grow deeper with no limitations. That could help with learning of complex patterns, with the ensemble preventing the overall model from overfitting. We also plotted the learning curve of random forest classifier. At first sight it does not show a very big improvement in comparison with the original learning curve, maybe it is due to our data noise or the quality of our features, but anyway it seems to exist a threshold at ~0.81 that with our dataset we cannot pass. For that reason we are going to compare other metrics, like accuracy, recall, f1-score, ROC curve.

Metric	Decision Tree	Random Forest	Interpretation
Recall	0.8151	0.8125	Minimum variance→RF detects the (~) same percentage of true positives than DT
Accuracy	0.8122	0.8626	RF is a better predicting cases than DT

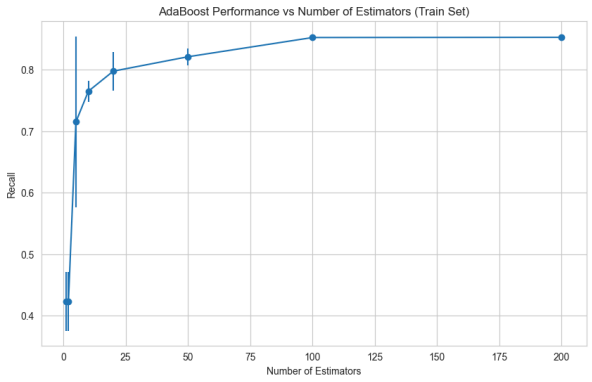
Precision	0.8103	0.9028	RF detects less false positives than DT
F1-score	0.8127	0.8553	RF is more balanced (recall-precision) than DT
ROC AUC	0.8122	0.9363	RF is able to distinguish better 1 or 0 than DT

In conclusion, Random Forest is a more robust and reliable model than our Decision Tree.

- **Ada Boost**

For Ada Boost, we are going to follow the same structure as in Random Forest. First, try different estimators and time them in order to find a sweet point that balanced computational cost and performance. And then, try estimators and other parameters that could affect the performance.

The following figure shows the performance of recall in relationship with the number of estimators. We used the function *execute_ada_boost_classifier_for_different_classifiers()* to show the effect of increasing the number of trees.



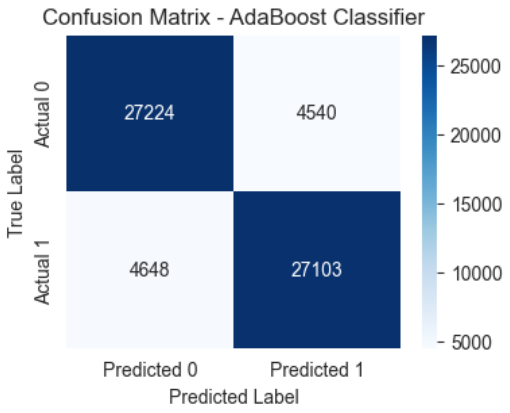

```
Recall: 0.4233 ± 0.0479 [n° estimators (trees): 1]
Time: 5.374006032943726 [n° estimators (trees): 1]
Recall: 0.4233 ± 0.0479 [n° estimators (trees): 2]
Time: 2.994502067565918 [n° estimators (trees): 2]
Recall: 0.7150 ± 0.1391 [n° estimators (trees): 5]
Time: 1.2203376293182373 [n° estimators (trees): 5]
Recall: 0.7649 ± 0.0169 [n° estimators (trees): 10]
Time: 1.6051175594329834 [n° estimators (trees): 10]
Recall: 0.7975 ± 0.0315 [n° estimators (trees): 20]
Time: 3.448737859725952 [n° estimators (trees): 20]
Recall: 0.8209 ± 0.0139 [n° estimators (trees): 50]
Time: 7.388431787490845 [n° estimators (trees): 50]
Recall: 0.8522 ± 0.0029 [n° estimators (trees): 100]
Time: 14.028777122497559 [n° estimators (trees): 100]
Recall: 0.8525 ± 0.0028 [n° estimators (trees): 200]
Time: 28.93548345565796 [n° estimators (trees): 200]
```

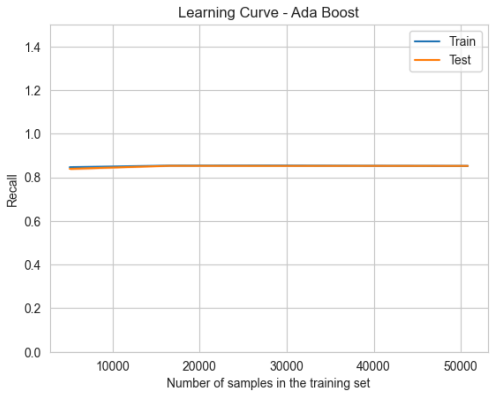
In Ada Boost classifiers we can see that while the number of estimators increase, mean recall increases and SD decreases. With multiple trees, it becomes a more stable model and generalizable because of the increment of recall and diminution of variance. We can observe that there is a sweet point around **100 estimators**. We achieve a high recall of 0.8522 and a low SD of 0.0029. Between 100 and 200 the computational time is two times higher and the difference in recall is minimum, so balancing time and performance we are going with 100 estimators. In order to find more hyperparameters, we are going to use gridSearch.

Parameter	Option	Description	Best
n_estimators	80, 90, 100, 110, 120	Range around sweet point 100	110
learning_rate	0.01, 0.1, 0.5, 1.0	How much each one contributes	1.0

```
Time AdaBoost GridSearchCV: 230.54 seconds
Best Params = {'learning_rate': 1.0, 'n_estimators': 110}  -  Recall = 0.8523779527559056
AdaBoost Accuracy: 0.8553
AdaBoost Recall: 0.8536
AdaBoost Precision: 0.8565
AdaBoost F1-score: 0.8551
AdaBoost Classification Report:
              precision    recall  f1-score   support

           0       0.85      0.86      0.86     31764
           1       0.86      0.85      0.86     31751

    accuracy                           0.86     63515
   macro avg       0.86      0.86      0.86     63515
weighted avg       0.86      0.86      0.86     63515
```

```
AdaBoost ROC AUC: 0.9380
```





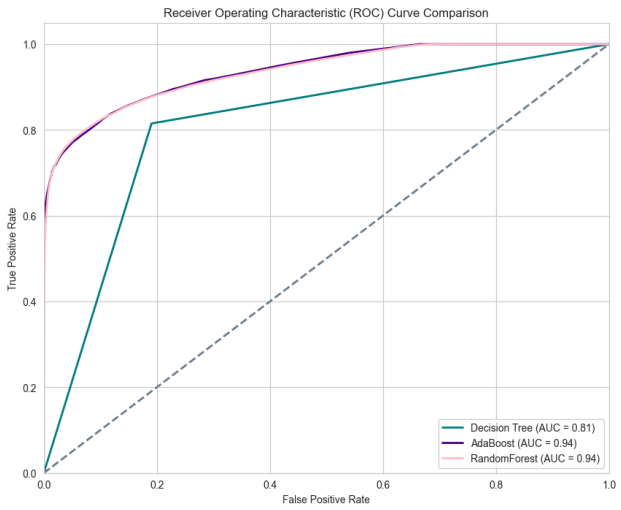
The best parameter found by gridSearch are **110 estimators** (around the first approach 100), which gives us a high recall, and **1.0** which means that each tree contributes at full weight. We also plotted the learning curve of Ada Boost. At first sight we see a clear difference with the other learning curves, that is, both training and testing are constant at the same value ~0.8. That means that **Ada Boost is generalizing very well**, so it is preventing overfitting. In comparison with Random Forest and the original Decision Tree, where the gap between train and test was notorious and that could mean data noise or poorer generalization in test, Ada Boost is performing as good in train as in test.

Metric	Decision Tree	Ada Boost	Interpretation
Recall	0.8151	0.8536	AB detects more true positives than DT
Accuracy	0.8122	0.8553	AB is a better predicting cases than DT
Precision	0.8103	0.8565	AB detects less false positives than DT
F1-score	0.8127	0.8551	AB is more balanced (recall-precision) than DT
ROC AUC	0.8122	0.9380	AB is able to distinguish better 1 or 0 than DT

In conclusion, Ada Boost is also a more robust and reliable model than our Decision Tree. And Ada Boost generalizes better than Random Forest

Comparison

For our decision Tree, Random Forest and Ada Boost, we decided to compare them by plotting the ROC curve with the results of three models.



Model	Accuracy	Recall	Precision	F1-score	ROC AUC
Decision Tree	0.8122	0.8151	0.8103	0.8127	0.8122
AdaBoost	0.8551	0.8546	0.8554	0.8550	0.9382
RandomForest	0.8650	0.8154	0.9050	0.8579	0.9376

Our original decision tree has the lower performance across all metrics. Random Forest performs well, it has the highest precision, and it has a good performance overall. Random Forest has a better performance than our single decision tree. Ada Boost has the best performance over all. It gives the best Recall (what we focus on), the best ROC curve and very good f1-score.

Ada Boost's logic is to build trees sequentially, with each new tree focusing on the mistakes made by the previous ones. This allows it to better handle edge cases, which explains the higher recall. Additionally, as a boosting method, it is capable of adapting while still maintaining good generalization

5.4. Support Vector Machines

Discuss choice of kernel and parameters used. Did you run any method to speed the building of the model? Report number of supports of the selected machine and try to interpret why the kernel selected and parameters selected for the final run give you the best results for your dataset. Try also to inspect main supports of your machine.

We employ SVM to predict heart attacks based on the preprocessed dataset described in sections 2 and 3. The goal of SVM is to find the optimal hyperplane that separates the two classes (0: no heart attack, 1: heart attack) with the largest possible margin.

For non-linearly separable data, SVM uses kernel functions that transform the data to a higher dimensional space where linear separation is feasible. Of course, in this model we have used the unbalanced dataset

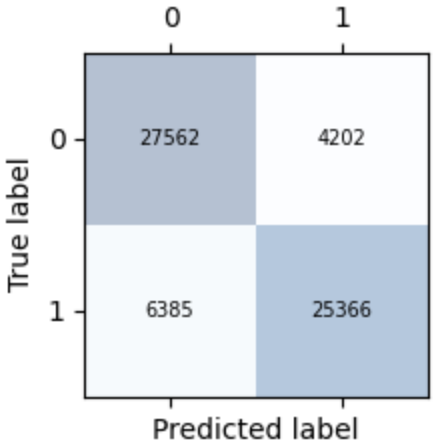
To optimize the performance of this model SVM, we executed 3 types of kernels on our dataset (to select the most appropriate one with better results): Linear kernel, Polynomial kernel and RBF kernel.

Overall methods to accelerate this model execution
It is worth noting that, in trying to run these kernels, we had endless runs in a long time (in the magnitude of hours). Because we have a 'large' dataset (150,000 rows) and SVM Kernels have a considerable quadratic computation time (algorithm has a cubic cost relative to the number of rows!) , we decided to take a stratified sample of 30,000 (enough to maintain trends) with a reduced fold cross validation down to 3.

Also to parallelize the execution, at GridSearchCV exists an parameter called n_jobs which allows you to specify how many cores can be used in the execution of the SVC kernel. We had also considered using the cuml library, in order to use the GPU's potential to run these kernels, but due to incompatibility errors we decided not to risk it. Maybe if we had powerful tools we could have gotten these results but we had to stick to using Google Colab or Jupyter.

Linear Kernel

Since our highest correlations (e.g., obesity vs. waist_circumference = 0.3954) are moderate and do not indicate severe multicollinearity (section 3), this kernel is a viable option.



Accuracy on test set: 0.833314964968905

Class Name	precision	recall	f1-score	support
0	0.81	0.87	0.84	31764
1	0.86	0.80	0.83	31751

macro avg	**0.83**	**0.83 63515**	**None**
weighted avg	0.83	0.83	0.83 63515

Confusion matrix on test set:
[[27562 4202]
[6385 25366]]
Accuracy on test set: 0.833314964968905
Accuracy : 0.833 (95% CI 0.830 – 0.836)
Precision (1): 0.858 (95% CI 0.854 – 0.862)
Recall (1): 0.799 (95% CI 0.794 – 0.803)
Normal-approximation acceptable for accuracy?: True

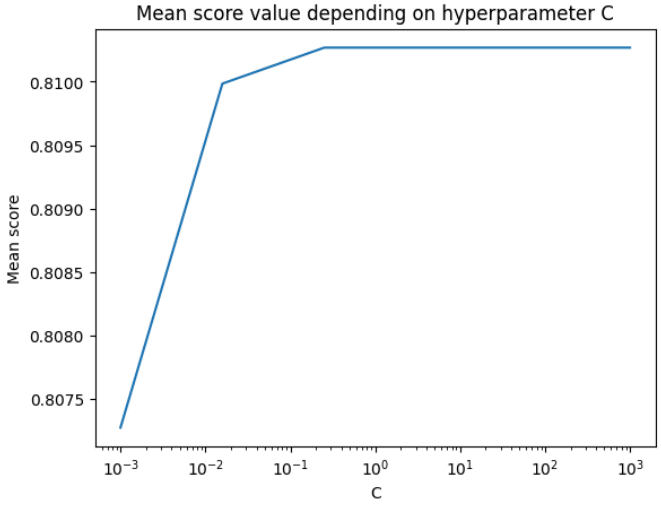
We see that the confusion matrix tells us that the linear SVM has a solid performance (83.2% accuracy (f1-score), 85.6% precision and 79.9% recall for class 1), but there is room for improvement, especially in the reduction of false negatives (20.1%).

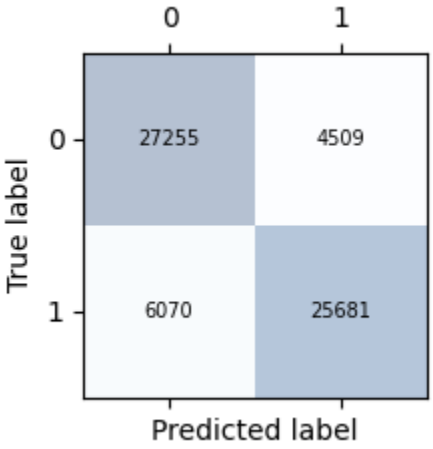
But, we can do better, with the regularising parameter C having to be adjusted and we have used GridSearch (systematically explore combinations of parameters) to find the optimal value of C (find the best possible parameters), as has been done for k in KNN.

We tried to do it in a similar way to the notebooks provided in class, but due to the very long times achieved, we decided to reduce it to complete this method. To optimize the computation time of this kernel, for the GridSearch we have specified the kernel with LinearSVC (instead of using the conventional SVC(kernel = 'linear')) and limited the maximum number of iterations.

We have also avoided cross validation for the C value that is actually obtained from the grid search. The only drawback of this is that LinearSVC has no support number attribute, which we have alternated with the coefficient shape, obtained from (1,33).

We obtained that the best value for this is with **C= 0.2511**, but even when adjusting for this C we can't really see an improvement, it even stays the same.





Best C found: 0.25118864315095796
Accuracy on test set: 0.8334409194678423

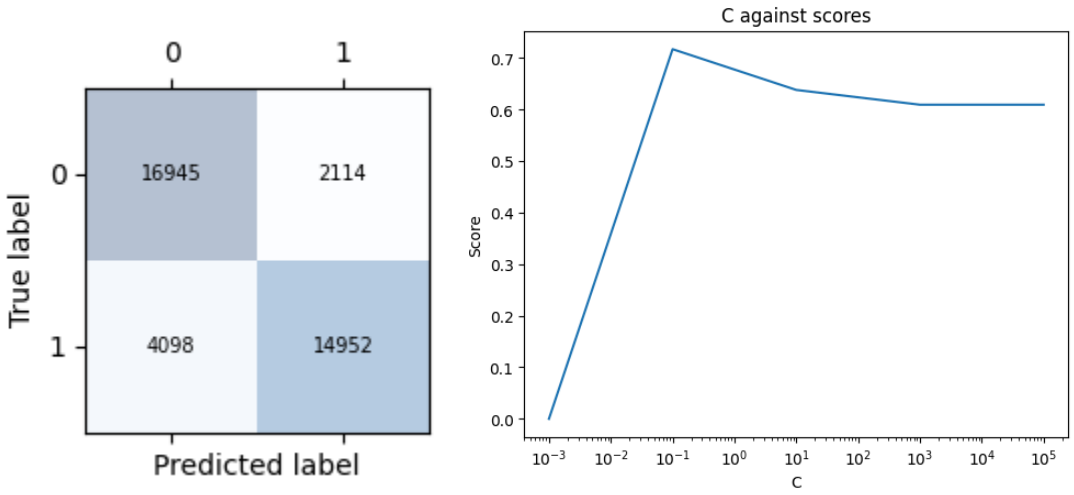
Class Name	precision	recall	f1-score	support
0	0.82	0.86	0.84	31764
1	0.85	0.81	0.83	31751

macro avg	0.83	0.83 63515	None
weighted avg	0.83	0.83	0.83 63515

We have noticed that, in addition to using the balanced dataset, we get a good accuracy in a small computational time O(n*d) [n: number of samples, d: dimensionality of data], as it only does scalar product between two feature vectors (does not perform non-linear transformations).

Polynomial Kernel

With the polynomial kernel, we only performed with a quadratic polynomial kernel (degree = 2), as we experienced that with a higher degree (degree = >3) we reached very long computation times . As can be seen in the following figure, we obtain similar results to the previous linear model, with practically the same accuracy in the confusion matrix.



Accuracy on test set: 0.8369938859586974
Best combination of parameters found: {'C': np.float64(0.1)}
Number of supports: 9990 (9990 of them have slacks)
Prop. of supports: 0.666

Class Name	precision	recall	f1-score	support
0	0.81	0.89	0.85	19059
1	0.88	0.78	0.83	19050

macro avg	**0.84**	**0.84 38109**	**None**
weighted avg	0.84	0.84	0.8438109

There is simply an imbalance in recall in both classes, which is completely normal. However it is apparent that the model is capturing the balance of the dataset well, maintaining accuracy, with high sensitivity.

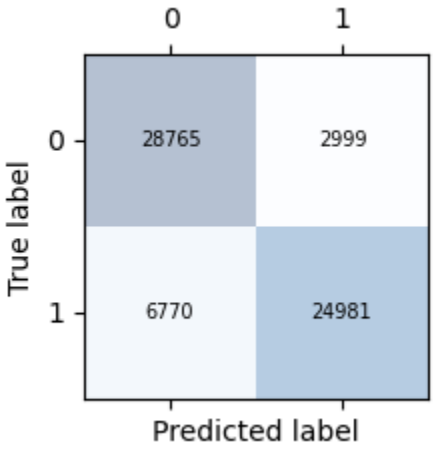
In the same way, we had to optimize the run to find the best value of C for the polynomial kernel, but in this case we have obtained that the value was 0.1. However, with this value the same recall is maintained.

RBF Kernel

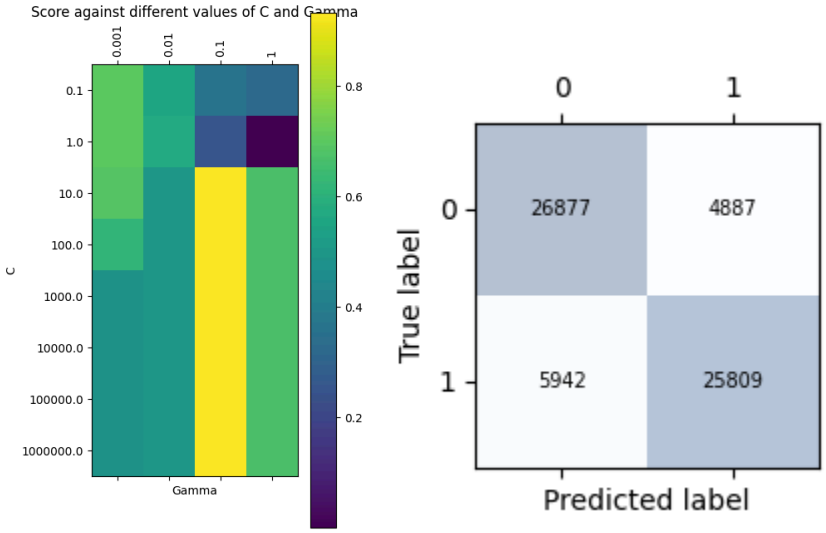
Lastly, we repeat the process but with RBF Kernel, the results obtained are almost identical compared to previous models. A first run gives us the following results, where we can see that the recall goes up 91% for the class 0 and the confusion matrix is also slightly better but not by much.

Class Name	precision	recall	f1-score	support
0	0.81	0.91	0.85	31764
1	0.89	0.79	0.84	31751

macro avg	**0.85**	**0.85 0.85**	**None**
weighted avg	0.85	0.85	0.856351



We can conclude that this is a good result and that it improves in contrast to the polynomial kernel previously mentioned. To perform the grid search we have used the following values Cs = np.logspace(-3, 4, num=5, base=10.0). As the image below shows, for a C value of 10 and gamma 0.01 we get accuracy of 83%. The conclusion we can take from all this is that, because we have performed these algorithms on balanced datasets, given that SVM is a very good classifier due to its hyperplane-setting performance, we have obtained good results.



Best combination of parameters found: {'C': 10.0, 'gamma': 0.1}
Accuracy on test set: 0.8295048413760528

5.5. Meta-learning algorithms

Performance Majority Voting, Bagging, RandomForest and Adaboost. Explain parameters selected for each algorithm

---

After experimenting with many single models, it's important to combine predictions of learners to build a stronger one. This is known as meta-learning or ensemble learning. The idea is to use the same algorithms but apply them with different data splits or parameters. We use meta-learning for 3 main reasons. First, they achieve higher accuracy in practice. They all have different straights, so by using them, we will be able to incorporate domain knowledge into different learners. Finally, it will be able to reduce overfitting.

The meta-learning method is in the file *meta_learning.ipynb*. The data used for the analysis came from the preprocessed data in the file *data_balanced_normalized.csv* file.

For this method, cross-validation was used to separate the data into train slits and test splits. Just like the other methods, it was decided that it was best to slip the data 50/50 for the split train/test, since we have 127 030 samples in our data frame. Training 70% of the data had a very high computational cost. We are also going to set cv (cross-validation) to 5. This means that the dataset is split into 5 equal parts, using a different fold for validation in each round.

We will use multiple meta-learning methods, such as Majority Voting, Bagging, RandomForest and Adaboost. The goal of the project is to be able to predict a heart attack, so the recall value is the best parameter to study. We want to be able to identify the true positive cases. The accuracy is the proportion of all correct predictions (both true positives and true negatives) out of all predictions made. It tells us how often the model is correct overall. However, a recall is the proportion of actual positive cases (people who really had a heart attack) that were correctly identified. The recall will tell us how good the model is at catching actual heart attack cases, which is the most important information in our case.

It is also important to note that, we choose to use LinearSVC, instead of using SVC with a parameter kernel = ‘linear’. SVC did not run even after 70 minutes in our case and for the sake of the project and our learning experience, it was decided that LinearSVC was a better choice. It was suggested to use SVC with max iteration. However, by using this method, the recall value changed completely and gave a very low recall value. SVC uses a one-vs-one strategy for multi-class classification. This means it trains a separate classifier for every pair of classes. Per example, if you had 4 classes, it would train 6 classifiers (4x3 / 2). This can be good, but gets computationally expensive with more classes. LinearSVC uses a one-vs-rest approach. This means it trains one classifier per class, where each model tries to separate one class from all others. For example, with 4 classes, it trains 4 classifiers. This is simpler and generally faster (See references). Since our problem is binary (heart attack vs. no heart attack), the difference between one-vs-one and one-vs-rest strategies doesn’t matter, since both SVC and LinearSVC reduce to training a single classifier. So due to the time component, we chose to use LinearSVC.

Majority Voting

Two types of voting were considered: equal voting and weighted voting. For our tests, we used hard voting because soft voting isn't possible with LinearSVM, because it does not support probability estimates. In hard voting, each model gives one vote, and the final prediction is the class that gets the most votes. However, soft voting uses the average of class probabilities predicted by each model. Both methods have their advantages, but we chose hard voting to include the SVM model. We also tried to calibrate the SVM classifier's predicted probabilities so that we could use soft voting, but the results weren't as good as using hard voting, so this idea was put to the side.

For weighted voting, we needed to find the best weights to assign to each classifier (GaussianNB, KNN, Decision Tree, and SVM) to maximize the recall value. We used GridSearchCV to test different weight combinations, starting from [1, 1, 1, 1] up to [2, 2, 2, 2]. The combination [1, 1, 1, 2] gave the best recall, which means giving more weight to SVM improved performance. After this result, we tested a few other combinations to see if we could find a better combination of weights. We considered the following combination, because we knew from previous tests that KNN gave a lower recall. We tested the following combination: [2, 1, 2, 2], [2, 1, 3, 2] and [2, 2, 3, 2]. We found that [2, 1, 2, 2] gave a better recall than the others. This likely happened because KNN wasn't contributing as much to improving recall, so reducing its weight helped improve the overall performance. In other words, our testing showed that giving more weight to the stronger models (like SVM, GaussianNB and Decision Tree) while still considering KNN leads to better results during cross-validation. The best parameters for the KNN method were n_neighbors = 5, weight = 'distance'.

In this initial test, it is possible to see that Knn has a slightly lower recall value.

```
Recall: 0.802 [Naive Bayes]
Recall: 0.787 [KNN (n_neighbors=5, weights=distance, p=1)]
Recall: 0.790 [Decision Tree]
Recall: 0.801 [Linear SVM]
```

After performing a majority voting, we can observe that the recall is 0.751. After performing a weighted vote, we observe that the recall has a value of 0.818. These results show that weighted voting gives a better recall value. This is probably due to the fact that the Knn method has a lower recall value, so by reducing its weight for the weighted voting, the recall value improves.

	Majority hard voting	Weighted voting
Recall Value	0.751	0.817

Looking at the results below, we can see the same patterns we noticed earlier. The KNN method has the lowest recall when it comes to predicting heart attacks, with a score of 0.79. Decision Trees stand out overall, with the best recall scores: 0.81 for no heart attacks and 0.82 for heart attacks. The recall value for SVM and GNB are the same, 0.81, but SVM has a recall value of 0.86 for predicting no heart attacks.

GaussianNB

	precision	recall	f1-score	support
No Heart Attack	0.81	0.80	0.80	31764
Heart Attack	0.80	0.81	0.80	31751
accuracy			0.80	63515
macro avg	0.80	0.80	0.80	63515
weighted avg	0.80	0.80	0.80	63515

KNN

	precision	recall	f1-score	support
No Heart Attack	0.80	0.84	0.82	31764
Heart Attack	0.83	0.79	0.81	31751
accuracy			0.82	63515
macro avg	0.82	0.82	0.82	63515
weighted avg	0.82	0.82	0.82	63515

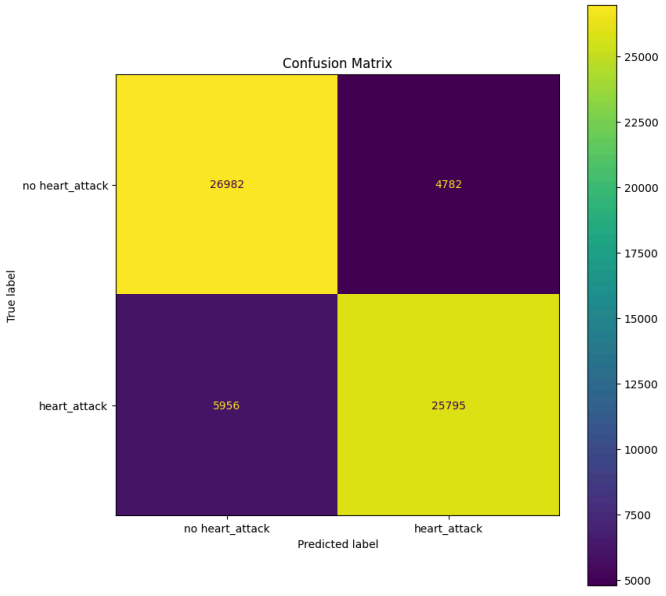
SVM

	precision	recall	f1-score	support
No Heart Attack	0.82	0.86	0.84	31764
Heart Attack	0.85	0.81	0.83	31751
accuracy			0.83	63515
macro avg	0.83	0.83	0.83	63515
weighted avg	0.83	0.83	0.83	63515

DT

	precision	recall	f1-score	support
No Heart Attack	0.81	0.81	0.81	31764
Heart Attack	0.81	0.82	0.81	31751
accuracy			0.81	63515
macro avg	0.81	0.81	0.81	63515
weighted avg	0.81	0.81	0.81	63515

In the confusion matrix below, we can see that combining all the models we saw gives us a recall of 0.81 for heart attacks and 0.85 for no heart attack. By combining the methods, it leads to a better recall than what we got with just KNN, but compared to the other individual models, the improvement isn't significant. The recall for predicting no heart attack is better here, compared to all of the singular models except SVM. However, the recall value for heart attacks stays the same as the singular models. For Decision Trees, its own recall for heart attacks is actually slightly better (0.82 vs 0.81), but the combined model does better at predicting no heart attack. So overall, the ensemble is better at predicting when there's no heart attack, and about the same for predicting when there is one, with Decision Trees still doing slightly better on its own. Finally, the confidence interval was also calculated and it gave: [0.804 – 0.813], which is quite small,indicating that the model's recall performance is stable and reliable.



Classification Report

	precision	recall	f1-score	support
no heart_attack	0.82	0.85	0.83	31764
heart_attack	0.84	0.81	0.83	31751
accuracy			0.83	63515
macro avg	0.83	0.83	0.83	63515
weighted avg	0.83	0.83	0.83	63515

Bagging

Bagging helps reduce overfitting by combining predictions from the same model trained on different random subsets of the data. Since decision trees are very sensitive to the training data (high variance), bagging is especially effective for them, making the final predictions more stable and accurate. In our case, since we are already analyzing every method, we will perform bagging with every method, not just decision trees. For the first step of bagging, we need to find the best estimators to train our values. For the Decision Tree, we chose n_estimators = 50 because it gave the best recall score of 0.803. This value was ideal as the performance started to stabilize around this point, and the convergence time was lower compared to using 100 or 200 estimators, which provided similar results but took longer to run. Based on the results of GaussianNB (GNB), we chose n_estimators = 20 for GaussianNB, since it gave a recall of 0.810 which was equivalent to the recall value of 50 and 100, but 20 had a much faster training time. For KNN, we also went with n_estimators = 20 which gave a recall of 0.787. Although using 100 gave the same recall value, the training time was extremely long, so an estimator of 20 was a better balance between performance and efficiency. For LinearSVC, it was decided to choose n_estimators = 50. This value was equal to the value of 100 estimators , but it had a less high convergence time, so for us, it was the best option.

```
Recall DT: 0.808 [nº estimators: 1]
Recall DT: 0.731 [nº estimators: 2]
Recall DT: 0.811 [nº estimators: 5]
Recall DT: 0.791 [nº estimators: 10]
Recall DT: 0.799 [nº estimators: 20]
Recall DT: 0.803 [nº estimators: 50]
Recall DT: 0.804 [nº estimators: 100]
Recall DT: 0.805 [nº estimators: 200]
```

```
Recall KNN: 0.784 [1]
Recall KNN: 0.786 [2]
Recall KNN: 0.786 [5]
Recall KNN: 0.786 [10]
Recall KNN: 0.787 [20]
Recall KNN: 0.786 [50]
Recall KNN: 0.787 [100]
```

```
Recall GNB: 0.811 [1]
Recall GNB: 0.811 [2]
Recall GNB: 0.810 [5]
Recall GNB: 0.809 [10]
Recall GNB: 0.810 [20]
Recall GNB: 0.810 [50]
Recall GNB: 0.810 [100]
```

```
Recall LinearSVC: 0.808 [1]
Recall LinearSVC: 0.803 [2]
Recall LinearSVC: 0.809 [5]
Recall LinearSVC: 0.808 [10]
Recall LinearSVC: 0.809 [20]
Recall LinearSVC: 0.810 [50]
Recall LinearSVC: 0.810 [100]
```

Using the optimal number of estimators, we built a Bagging model for each of the previous classifiers to see its effect on the recall. The results show that GaussianNB, Decision Tree, and SVM all achieved the same recall score of 0.81 for predicting heart attacks. Once again, KNN performed worse for this class, with a recall of only 0.63. However, when predicting no heart attacks, Bagging significantly improved the recall scores for several models: KNN, Decision Tree, and SVM reached 0.88, 0.92, and 0.86, respectively.

GaussianNB

	precision	recall	f1-score	support
no heart_attack	0.80	0.80	0.80	31764
heart_attack	0.80	0.81	0.80	31751
accuracy			0.80	63515
macro avg	0.80	0.80	0.80	63515
weighted avg	0.80	0.80	0.80	63515

KNN

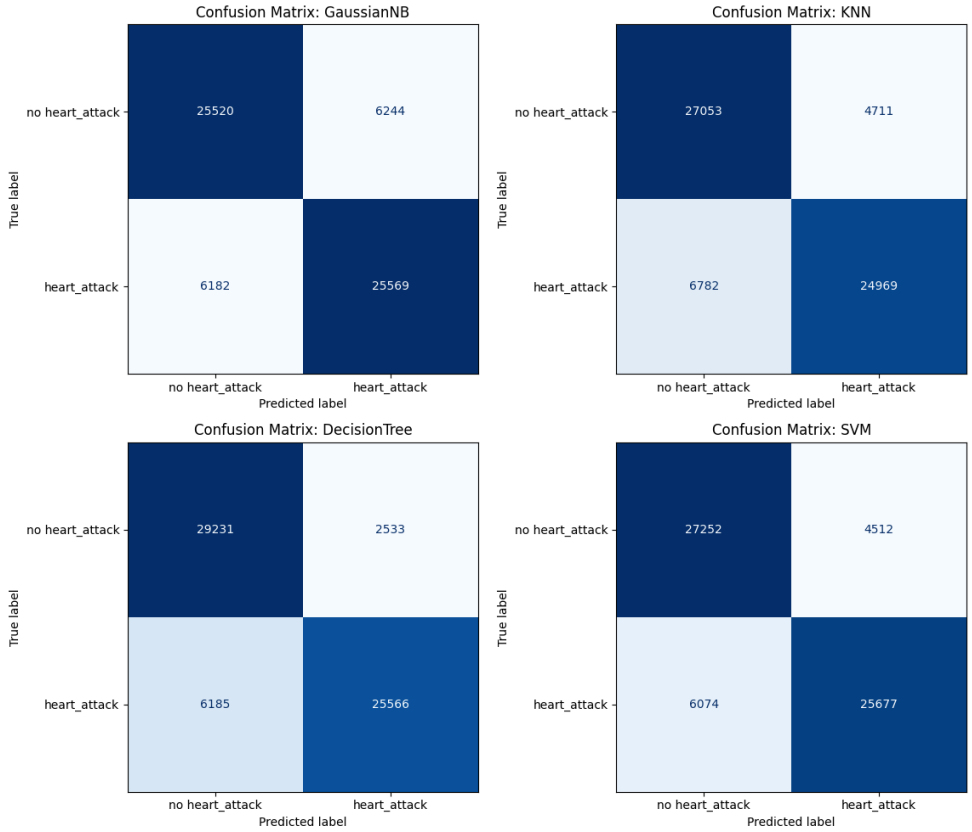
	precision	recall	f1-score	support
no heart_attack	0.80	0.85	0.82	31764
heart_attack	0.84	0.79	0.81	31751
accuracy			0.82	63515
macro avg	0.82	0.82	0.82	63515
weighted avg	0.82	0.82	0.82	63515

DecisionTree

	precision	recall	f1-score	support
no heart_attack	0.83	0.92	0.87	31764
heart_attack	0.91	0.81	0.85	31751
accuracy			0.86	63515
macro avg	0.87	0.86	0.86	63515
weighted avg	0.87	0.86	0.86	63515

SVM

	precision	recall	f1-score	support
no heart_attack	0.82	0.86	0.84	31764
heart_attack	0.85	0.81	0.83	31751
accuracy			0.83	63515
macro avg	0.83	0.83	0.83	63515
weighted avg	0.83	0.83	0.83	63515



Confidence intervals:

```
GNB:
Accuracy     : 0.804  (95% CI 0.801 - 0.807)
Precision : 0.804  (95% CI 0.799 - 0.808)
Recall   : 0.805  (95% CI 0.801 - 0.810)

KNN:
Accuracy     : 0.819  (95% CI 0.816 - 0.822)
Precision : 0.841  (95% CI 0.837 - 0.845)
Recall   : 0.786  (95% CI 0.782 - 0.791)

DT:
Accuracy     : 0.863  (95% CI 0.860 - 0.865)
Precision : 0.910  (95% CI 0.906 - 0.913)
Recall   : 0.805  (95% CI 0.801 - 0.810)

SVM:
Accuracy     : 0.833  (95% CI 0.830 - 0.836)
Precision : 0.851  (95% CI 0.846 - 0.855)
Recall   : 0.809  (95% CI 0.804 - 0.813)
```

Random Forest

Random Forest combines the output of multiple decision trees to reach a single result. This approach reduces overfitting by averaging uncorrelated trees and provides flexibility for both classification and regression tasks. To evaluate the performance of the Random Forest model, we calculated the recall for different values of n_estimators. Starting from n_estimators = 50, the recall becomes high and more stable. Therefore, we chose n_estimators = 200 as a value for training the Random Forest model.

Random Forest with different estimators:

```
Recall: 0.775 [nº estimators (trees): 1]
Recall: 0.650 [nº estimators (trees): 2]
Recall: 0.810 [nº estimators (trees): 5]
Recall: 0.789 [nº estimators (trees): 10]
Recall: 0.802 [nº estimators (trees): 20]
Recall: 0.808 [nº estimators (trees): 50]
Recall: 0.811 [nº estimators (trees): 100]
Recall: 0.813 [nº estimators (trees): 200]
```

We will also perform ExtraTrees, which is an ensemble machine model that combines multiple decision trees, similar to Random Forest but with additional randomization. For the ExtraTrees model, we also chose to use 200 estimators. This number provides a good and stable recall while being more computationally efficient than using 200 estimators, which offers similar performance with increased training time.

Extra Trees with different estimators:

```
Recall: 0.754 [nº estimators (trees): 1]
Recall: 0.617 [nº estimators (trees): 2]
Recall: 0.786 [nº estimators (trees): 5]
Recall: 0.759 [nº estimators (trees): 10]
Recall: 0.776 [nº estimators (trees): 20]
Recall: 0.783 [nº estimators (trees): 50]
Recall: 0.787 [nº estimators (trees): 100]
Recall: 0.789 [nº estimators (trees): 200]

123.36028957366943 seconds
```

To see which model performs better, we compared RandomForest and ExtraTrees side by side. We used the same training setup for both, with 200 estimators. In the result below, it is possible to observe that RandomForest has a higher recall for the heart_attack class (0.82), compared to ExtraTrees (0.79). The results can show that RandomForest is slightly better at catching real heart attack cases. RandomForest also has a better recall value for no heart attacks, which means it is also a bit better at correctly identifying people who don't have a heart attack. Both models have similar F1-scores, which is a good balance of precision and recall. The overall accuracy is also slightly higher for RandomForest.

RandomForest

	precision	recall	f1-score	support
no heart_attack	0.83	0.91	0.87	31764
heart_attack	0.90	0.82	0.86	31751
accuracy			0.86	63515
macro avg	0.87	0.86	0.86	63515
weighted avg	0.87	0.86	0.86	63515

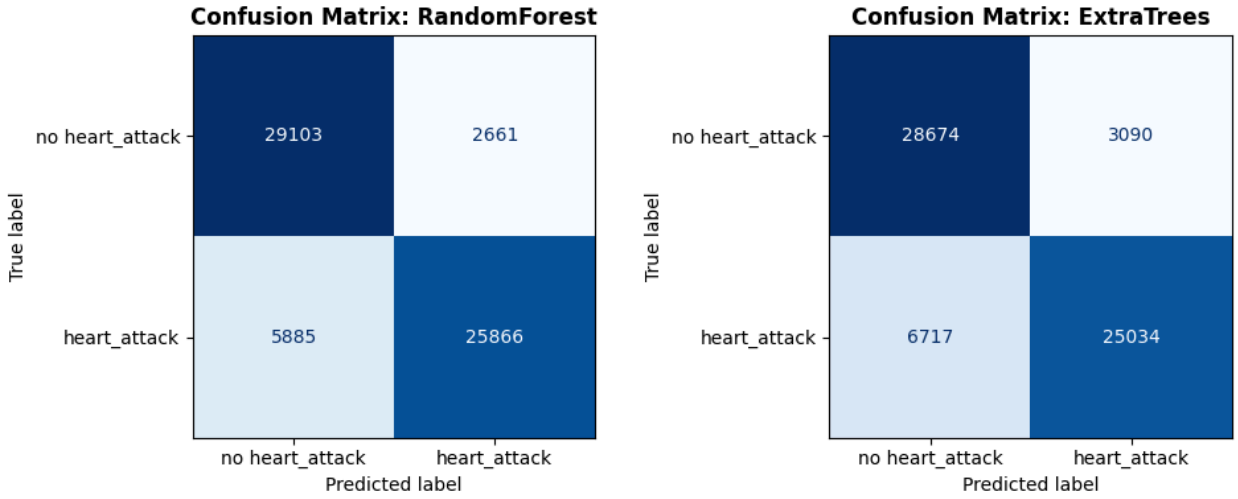
ExtraTrees

	precision	recall	f1-score	support
no heart_attack	0.81	0.90	0.85	31764
heart_attack	0.89	0.79	0.84	31751
accuracy			0.85	63515
macro avg	0.85	0.85	0.85	63515
weighted avg	0.85	0.85	0.85	63515

After these results, we decided to perform the McNemar test that also compares the predictions of two models. We will use this test to check if there's a significant difference between the performances of RandomForest and ExtraTrees. In the test, it is possible to see that the null hypothesis was rejected. The p-value was much lower than the standard threshold (p=0.05), so it is possible to reject the null hypothesis. By doing so, we are implying it is possible that both models perform differently. The contingency table shows how often RandomForest and ExtraTrees agree or disagree on predictions. The McNemar's Test checks if the two models are significantly different. Here it is possible to observe that ExtraTrees correctly predicted 2997 cases where RandomForest was wrong, while RandomForest only correctly predicted 1688 cases that ExtraTrees missed.

```
 Contingency Table:

                     ExtraTrees Correct  ExtraTrees Wrong
RandomForest Correct               51946              1688
RandomForest Wrong                  2997              6884

 McNemar Test Results:
Statistics value: 365.17908217716115
p-value: 2.098339405361464e-81
Reject the null hypothesis
```

Here are the confusion matrices for RandomForest and ExtraTrees to help visualize the results explained earlier from a different perspective. It is also possible to see here that RandomForest slightly predicts true heart attacks better (25866 vs 25034).



Here are the confidence intervals for the two methods.

```
modelRandomForest:
Accuracy     : 0.865  (95% CI 0.863 - 0.868)
Precision : 0.907  (95% CI 0.903 - 0.910)
Recall    : 0.815  (95% CI 0.810 - 0.819)

modelExtraTrees:
Accuracy     : 0.846  (95% CI 0.843 - 0.848)
Precision : 0.890  (95% CI 0.886 - 0.894)
Recall    : 0.788  (95% CI 0.784 - 0.793)
```

AdaBoost

AdaBoost is a technique used as an ensemble method to adjust the weights of training samples and combine multiple weak classifiers into a single strong classifier. AdaBoost combines a lot of weak learners to make classifications. We are going to use AdaBoost with DT, GBN and LinearSVC. We didn't choose KNN because it's not ideal for boosting, which relies on fast and simple models as base learners. To use adaBoost, the first step was to determine the optimal number of estimators for our model. Using 5-fold cross-validation (cv = 5), we evaluated several options. For LinearSVC, we found that each estimator gave the same result of recall, so we decided to use 100. The recall value for GaussianNB was the same as soon as we reached 10, and we found that from 10 estimators, the recall value was the same. Then we decided to also calculate the recall value of different estimators for GaussianNB and we found that from 10 estimators, the recall value was the same, so we decided to also use 100 estimators. For the Decision Tree method, from 20 estimators, the result starts to stabilize and the best values for the higher recall appears to be 100 too.

```
Recall adaBoost LinearSVC: 0.793 [1]
Recall adaBoost LinearSVC: 0.793 [2]
Recall adaBoost LinearSVC: 0.793 [5]
Recall adaBoost LinearSVC: 0.793 [10]
Recall adaBoost LinearSVC: 0.793 [20]
Recall adaBoost LinearSVC: 0.793 [50]
Recall adaBoost LinearSVC: 0.793 [100]
Recall adaBoost LinearSVC: 0.793 [200]
```

```
Recall GNB: 0.810 [1]
Recall GNB: 0.810 [2]
Recall GNB: 0.848 [5]
Recall GNB: 0.854 [10]
Recall GNB: 0.854 [20]
Recall GNB: 0.854 [50]
Recall GNB: 0.854 [100]
Recall GNB: 0.854 [200]
```

```
Recall DT: 0.765 [1]
Recall DT: 0.765 [2]
Recall DT: 0.813 [5]
Recall DT: 0.818 [10]
Recall DT: 0.825 [20]
Recall DT: 0.825 [50]
Recall DT: 0.826 [100]
Recall DT: 0.823 [200]
```

In the following results, we can see that for GaussianNB, AdaBoost increased the recall value from 0.81 to 0.85, so there was a noticeable improvement. For the Decision Tree, AdaBoost didn't seem to change the recall value. As for SVC, the recall for predicting heart attacks slightly decreased (0.79 vs 0.81), which suggests that AdaBoost may not work well with SVC.

AdaBoost GNB

	precision	recall	f1-score	support
no heart_attack	0.85	0.80	0.82	31764
heart_attack	0.81	0.85	0.83	31751
accuracy			0.83	63515
macro avg	0.83	0.83	0.83	63515
weighted avg	0.83	0.83	0.83	63515

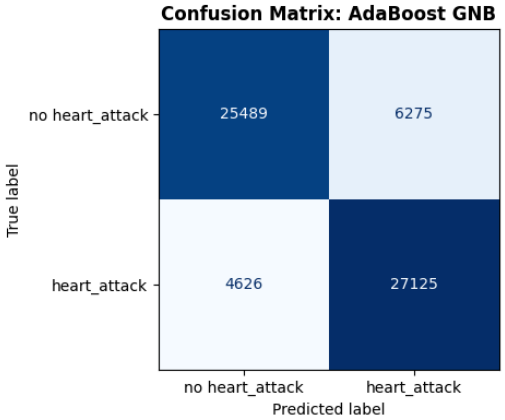
AdaBoost DecisionTree

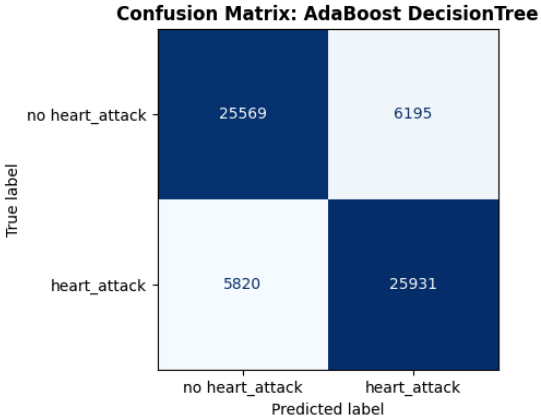
	precision	recall	f1-score	support
no heart_attack	0.81	0.80	0.81	31764
heart_attack	0.81	0.82	0.81	31751
accuracy			0.81	63515
macro avg	0.81	0.81	0.81	63515
weighted avg	0.81	0.81	0.81	63515

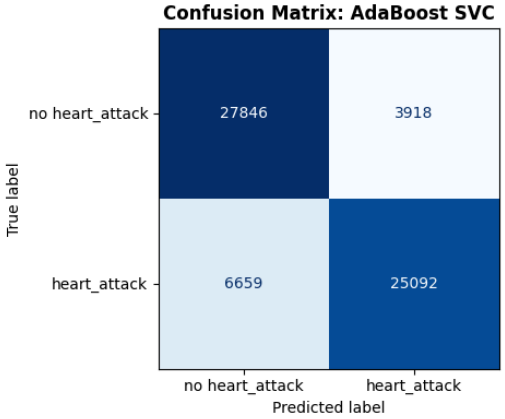
AdaBoost SVC

	precision	recall	f1-score	support
no heart_attack	0.81	0.88	0.84	31764
heart_attack	0.86	0.79	0.83	31751
accuracy			0.83	63515
macro avg	0.84	0.83	0.83	63515
weighted avg	0.84	0.83	0.83	63515

Here, we visualize the data using three confusion matrices. It is evident that the number of correctly identified heart attack cases (true positives) is higher when we use AdaBoost with GaussianNB.







6.Comparison and conclusions

Best methods on test set validation:

Classifier	Recall for class 1	Intervals of confidence
Naive Bayes	0.85	[0.801 - 0.810]
K-NN	0.87	[0.863 - 0.870]
Decision Trees	0.815	[0.812 - 0.818]
Support Vector Machines	0.83	[0.830 – 0.836]
Majority voting weighted	0.815	[0.811 - 0.819]
Bagging GNB	0.805	[0.801 - 0.810]
Bagging DT	0.805	[0.801 - 0.810]
Bagging KNN	0.786	[0.782 - 0.791]
Bagging SVM	0.809	[0.804 - 0.813]
Random Forest	0.815	[0.810 - 0.819]
Extra Trees	0.788	[0.784 - 0.793]
Ada Boost Linear SVC	0.790	[0.786 - 0.795]
Ada Boost GNB	0.854	[0.850 - 0.858]
Ada Boost DT	0.854	[0.851 - 0.856]

Despite that overall quite good performance, we can conclude that the best classifiers regarding the true positive rate are all the following: K-NN is the first with the best result with recall when testing on the validation data set, followed by both Naive-Bayes (and Naive Bayes with Ada Boost), and Ada Boost with DT.

Regarding the performance of K-NN, it seems to be reinforced due to the fact that optimal hyperparameter tuning with feature selection was applied, and the dataset was properly normalized. That way, the distances could be calculated with minimal noise and probability of overfitting, causing good enough predictions, with the recall being easily optimizable by lowering the threshold for classifying

probabilities, which had a higher interval of confidence than the other methods. In addition, the other metrics had scores above 0.7, and the computational time was low.

This good performances in Naive Bayes (even better with Ada Boost as it has better confidence intervals) fits with the low correlation between the variables that we saw when applying Pearson correlations during the preprocessing, reinforcing the independent assumption of the attributes in which Naive Bayes is based. It also presented a low computational time, and the other metrics apart from recall had a value of 0.8, or more.

Regarding the Ada Boost ensemble method with Decision Trees as the base algorithm, its functionality as a boosting technique involves creating multiple shallow decision trees that are capable of modeling non-linear relationships between features and the target variable. By sequentially combining these trees, Ada Boost builds a strong classifier that captures complex patterns in the data while controlling overfitting. In each iteration, the algorithm focuses on the errors made by the previous trees, allowing subsequent trees to correct them. Generally, Ada Boost assigns higher weights to misclassified instances to emphasize difficult cases; however, in our case, GridSearch determined that all trees contributed equally. Ada Boost with decision trees has proven to be the best method that generalizes data without overfitting. For a heart attack diagnosis, it is a key aspect of the capacity of generalization.

To conclude our report, conducting this study proved that there are multiple machine learning algorithms which are productive for classifying whether a patient will suffer from a heart attack, which all of them presenting good scores for recall within a high and narrow confidence intervals. Each one applied its methods and parameters, and they were tuned in the training stage for having the maximum performance regarding the true positive rate: correctly diagnosing the affected patients, while assuming a certain augmented degree of higher false positive rate (false alarms). In the testing stage of the models, they all presented good results, which were similar to the ones applied in the previous stage, which showed that the models performed consistently across different subsets of data (CV). This suggests that the training, validation and test set had similar distributions, with small differences in CV folds. That indicates that 5-folds were enough to represent the data distribution, considering our large dataset and its complexity.

All of this led us to think that the dataset had a large enough sample of patients, which gathered relevant information about them, and keeping all the samples allowed us to maximize the information available for training and validating our algorithm. Moreover, we deduced that the dataset was correctly preprocessed and adapted for each algorithm, preserving the main and important tendencies of the initial data, while balancing it and keeping it consistent.

That said, there was an exception with SVM. The large amount of samples, and the complexity of the data made the computational time too large (almost unreachable), so extra optimizations had to be applied, such as trying dimensionality reduction via PCA, sampling a small dataset of the whole data, applying CV of 2 folds and extra “optimizations” to reduce the time. This potentially affected the distributions between the variables, adding variance to the results, and also not allowing us to use the best kernels or finding the actual best parameters for it.

7.References

Kaggle. Heart Attack Prediction in Indonesia [Data set]. Retrieved May 19, 2025, from https://www.kaggle.com/datasets/ankushpanday2/heart-attack-prediction-in-indonesia/data

Scikit Learn. Support Vector Machine. Retrieved May 21, 2024, from https://scikit-learn.org/stable/modules/svm.html