Project submission

Team member:

Full Name: Amir Levy

Mail: amirlevy2@mail.tau.ac.il

Competition: **ICR - Identifying Age-Related Conditions**

Challenge

This competition focuses on predicting whether an individual has one or more specific medical conditions. The task involves building a model that classifies individuals into two groups: those with at least one of these conditions (Class 1) and those without any (Class 0), based on health-related data.

Diagnosing these conditions traditionally requires an extensive data-gathering process from patients, which can be intrusive. By applying predictive models, it's possible to streamline this process by identifying and encoding only the most relevant health characteristics while safeguarding patient privacy.

This effort aims to assist researchers in discovering patterns that link specific health measurements to the likelihood of certain conditions.

Data

- 617 partially observations. Each observation has 56 anonymized health characteristics (AB-GL). All are numeric except for EJ, which is categorical (string of A or B).
- Class: Binary target. 1 for positive observation, 0 otherwise.
- Example:

Id	AB	AF	AH	..	..	EJ	..	..	Class
000ff2bfdfe9	0.209377	3109.03329	85.200147	22.394407	8.138688	A	..	..	1

Submission

For each observation, probability to be in class 1 (and 0 respectively):

Id	1	0
000ff2bfdfe9	0.75	0.25

Ideal outcomes

Evaluation is LogLoss function:

$$\text{Log Loss} = \frac{-\frac{1}{N_0}\sum_{i=1}^{N_0} y_{0i}\log p_{0i} - \frac{1}{N_1}\sum_{i=1}^{N_1} y_{1i}\log p_{1i}}{2}$$

My goal is to get the lowest score possible (0 max). Leadboard score (Private score):

TOP	Score (<)
100	0.38190
250	0.40344
500	0.42292

The leaderboard is pretty competitive. My initial goal is to achieve a score below 0.5 using my own models and ideas from 0.

After that, I plan to use some existing solution and make some improvements to reach the top 500.

Ultimately, my target is to break into the top 250.

Expected compute and storage requirements

The dataset is quite small, with only 617 observations, so I expect it won't require many resources to work with.

Initial ideas

I plan to start with a few simple algorithms that I find effective: **SVM** (with and without logistic regression), **K-Means**, and **KNN** (K-Nearest Neighbors). I'll also incorporate **PCA**, as the data has 56 dimensions. GPT has also suggested trying **Random Forests**.

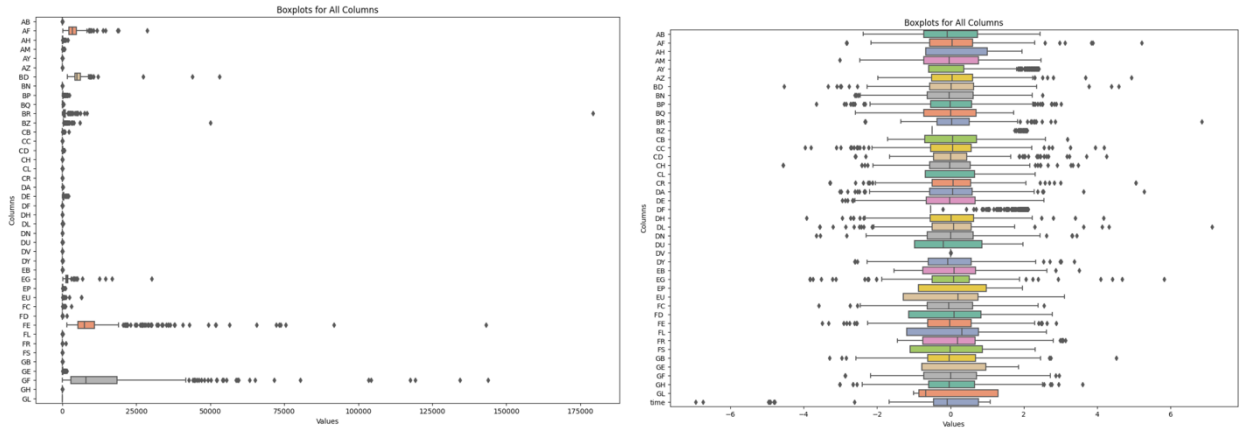
For preprocessing, I'll focus on **excluding biased columns** and assigning unique values for the 'A' and 'B' strings in the EJ column. I'll also look for smart ways to handle any missing values.

Initial Attempts

Data preprocessing

The data is **relatively straightforward**, so I anticipated some difficulty in removing data or applying weights to different columns. I decided to remove columns with **high or low variance**. High variance might indicate that a column can classify samples effectively—provided there's a strong correlation between high/low values in this column and a specific class. If such a connection exists, we prefer to retain these columns. Otherwise, it's better to remove them to avoid introducing noise. This preprocessing generally improved results across nearly every algorithm and method I used.

BoxPlots before / after transforming (using top 2 leadboard bad cols + trnasformer):



Typically, removing columns with high variance yielded the best outcomes. Most often, removing 10% was optimal, though occasionally removing up to 90% was more effective.

Additionally, I examined two more aspects: first, handling empty cells. I experimented with several methods, all of which produced similar results, and ultimately chose to **fill empty cells with the average value of each column**. Second, I converted the values in the 'EJ' column to numeric form. To ensure compatibility with the algorithms, I **replaced 'A' with 0 and 'B' with 1**, which was the most straightforward and effective approach among the methods I considered.

Attempts

1. KNN: In this approach, I use **several K-Nearest Neighbors (KNN) models with different values of k** to get solid predictions. For each odd k in a certain range, I train a separate KNN model to predict the probability that a sample belongs to class 1. Then, I take the **average of these probabilities across all the models**, which helps smooth out any inconsistencies that might come from using just one k value.

To turn these probabilities into final class predictions, for training purpose, I apply a threshold, classifying each sample as either class 0 or class 1 based on whether its averaged probability is above or below 0.5. Then, after finding the best 'k', I create an output that shows probabilities for both classes (0 and 1) for each sample, along with the sample IDs. This gives a more complete picture of how confident the model is in each prediction, both as probabilities and as actual classes.
The best values I found for 'k' were 6 and 56. I chose **56** because it felt safer, and, as I mentioned before, I **removed the top 10% of columns with the highest variance**. This resulted in a **private score of 0.783**, which I felt was quite decent for my first attempt.

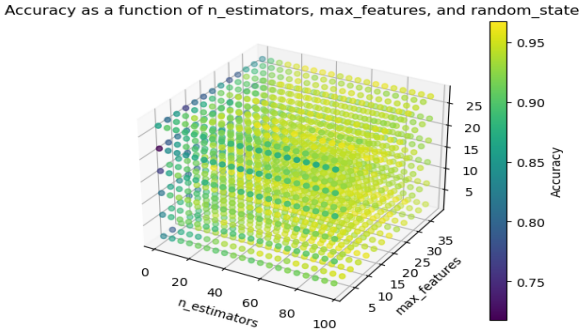
2. K-means: I tried to replicate the idea behind KNN, but it performed **much worse**, so I don't think it's worth much analysis. I experimented with different numbers of means and finally predicted the probability of each new data point being classified as 1 or 0 by **averaging based on the distance from each mean**: (sum of distances from class 1 means) / (sum of total distances from all means) = probability of classifying as class 1. The results were **quite poor**, with a **private score of over 4.1**. I like this algorithm because it's fairly simple and usually effective, so I thought it was worth a try.

3. Neural Networks: Probably one of the most well-known and widely used machine learning methods today, so I knew I had to give it a try. Unfortunately, it led to overfitting and did not produce good results. I used a **3-layer network** and achieved a **private score of 0.741** and a **public score of 0.47**, which can indicate for an **overfitting** (again, after removing the top 10% of columns with the highest variance).

4. Random forest: After applying some basic algorithms, I moved on to more classification-focused methods. One of the most classic and successful of these is the decision tree algorithm. However, I chose to use a more advanced version: the Random Forest (RF). **I started with a standard RF**, which involves basic boosting and aggregation. **Later, I plan to explore more complex algorithms** that use cross-boosting or aggregation across multiple boosting methods.

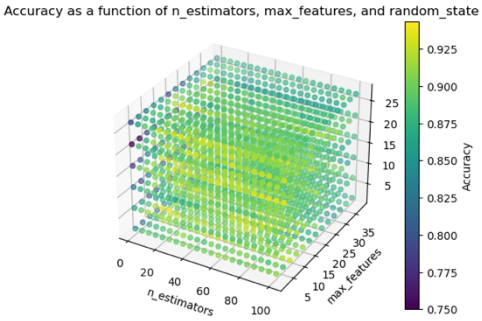
   The main idea of RF is to use many randomly generated decision trees and **classify based on the average result**. Two aspects are **randomized** here: the **training data for each tree** (a subset of the original training data) and the **features examined in each tree** (again, a subset of the original features). This method provides more reliable results.

For my first attempt, I used a basic configuration with **n_estimators=100, max_features=5, and random_state=4**, which resulted in a **private score of 0.492 and a public score of 0.365** (possibly indicating overfitting). After removing some columns (top 10% var), I achieved scores of **0.518 (private) and 0.38 (public).** I then decided to try selecting the best values for n_estimators, max_features, and random_state for this scenario:




```
Best Accuracy: 0.967741935483871
Best Configuration:
n_estimators: 31
max_features: 23
random_state: 2
```

I've tried to submit with this configuration, but I got bad results: **0.88 / 0.96**. **Without removing columns: 0.442 / 0.364.** This is the best result I've got so far.
Then, I've used again analysis of best configuration for a case I haven't removed columns:




```
Best Accuracy: 0.9435483870967742
Best Configuration:
n_estimators: 16
max_features: 5
random_state: 7
```

Result with (16,5,7): 0.53 / 0.322 (overfitting).

5. <u>Boosting:</u> After experimenting with Random Forest using basic and manual configurations, I realized I needed to try more advanced boosting methods such as **CatBoost, LightGBM, XGBoost and TabPFN**. Additionally, after inspecting the second-best result, I adopted its preprocessing approach.

I've used **scaling and transforming** to make the data more reliable. Then I used different methods of boosting:

a. XGBoost: Its a powerful, efficient, and scalable **implementation of gradient boosting**. Used mostly for structured/tabular data. Features like regularization and tree pruning **help reduce overfitting**.

   The basic configurations were: n_estimators = 100, max_depth = 7. Later on, I changed the configuration which gave me better results (using all 4 methods together). The results i got fot this scenario were: **0.58 / 0.41.**

b. LightGBM: Fast, efficient, and memory-optimized gradient boosting library. Uses techniques like **histogram-based learning** and leaf-wise tree growth for speed and accuracy. Well-suited for **large datasets**.

   Basic configurations were: max_depth=5, learning_rate=1e-2,metric = 'binary_logloss', n_estimators=100. Results: **0.47 / 0.47.**

c. CatBoost: Gradient boosting library **optimized for datasets with categorical features.** Handles categorical data natively without requiring extensive preprocessing. **Robust to overfitting** and requires minimal parameter tuning.

   Basic configurations: iterations=1000, learning_rate=1e-2, depth=5, loss_function='Logloss'. Results: **0.44 / 0.25.**

d. TabPFN: A neural network-based model for tabular data. Utilizes prior knowledge and probabilistic approaches to generalize well across various datasets. Known for achieving high accuracy with minimal configuration. Results: **0.44 / 0.28.**

Best Attempts:

First, I **combined all four methods and predicted using the average prediction across them.** Then, as mentioned in the first lesson, one of the most important aspects of this process is **preprocessing and deeply understanding** your data's behavior.

I realized that while boosting methods could yield the best results, the **data needed more careful handling**. Upon closely reviewing the dataset, I found that **approximately 80% of the training data was labeled as 0.** This meant that even a model classifying everything as 0 could achieve a score of around 0.8, which is misleadingly high and clearly unacceptable.

To address this imbalance and avoid noise or weak learning, I **adjusted** the configurations of XGBoost, LightGBM, and CatBoost to **use 'balanced' weights**. This adjustment ensured that the models accounted for the disproportionate number of 0 labels compared to 1 labels. As a result,

the learning process became **much more reliable,** and I achieved scores **around 4.2**, the lowest I had seen at that point.

I knew further improvements were possible through fine-tuning. The first adjustment I made was **assigning different weights to each booster**. I found that **TabPFN was the most reliable model** overall, though in some cases, it benefited from the support of other models. To leverage this, I configured TabPFN with a higher weight compared to the others.

One of my best results was a score of **0.398**, achieved when TabPFN was assigned a weight of 0.55, CatBoost 0.25, and the other models 0.1 each. This score is getting me to top 250.

Finally, I improved the preprocessing by using a **more effective transformer** from a different package and addressed an issue where the test data was being scaled independently. I ensured **both the training and test data used the same scaler** for consistent transformations. With this adjustment and a configuration of (0.55, 0.25, 0.1, 0.1), I achieved an excellent result of **0.387**, securing the **121st** place.

Additionally, after further fine-tuning and incorporating a **reweight function** for final processing after prediction, I achieved my best results, reaching the **top 15** with scores of **0.355 and 0.363.**

All other best attempts in the final table (next page).

lessons learned, potential extensions & improvements

There are several lessons I learned from this project:

a. **Preprocessing**: This is one of the **most critical steps**. Understanding the nature of your data and analyzing its structure can significantly help **reduce noise** and ensure that only reliable data is used. This approach enables faster execution and allows you to build a better model that **avoids overfitting** and isn't distracted by unwanted noise in the training data.
b. **Choosing Models**: This is obviously another key step. Each model performs better with **certain data types**. Therefore, it's important to test multiple models to identify where each one excels and where it struggles. Leveraging this information allows you to **select the most suitable model** for each specific task or dataset.
c. **Fine-Tuning**: After preprocessing the data and experimenting with different models, you will typically achieve initial results. In most cases, **no single model will fit your data perfectly**. To overcome this, **combining multiple models** and assigning different weights to each can help create an improved ensemble model that is more likely to yield accurate results.

After examined more solution, a further improvement and extenstions can be using deep learning to analyze the data even better and get the best results. This solutions demands very high proficiency with those networks in order to get better results then normal boosting methods. In addition, most people, including the 2nd best (and me), found the additional data file, 'greeks', is quite redundent. It doesn't has strong connection fot the traind data, and it's not really safe to use this data in a way that can help us train a better model.

Another improvement can be, to pass through large number of posabilities of bad columns until we find the most reliable columns to use. This part can make segnificant change (as my results has shown).

Results by models and more:

#	Preprocessing	Models	Configurations	Private Score	TOP
1	Remove bad cols	KNN	Avg by multy models	0.783	
2		Kmeans		4.122	
3		N-Networks	3 layers	0.741	
4	Remove bad cols	RF	100 est, 5 feat	0.492	
5	Without removing		31 est, 23 feat	0.442	
6	Scale and transform	XGBoost	100 est, 7 depth	0.58	
7		LGB	100, 5	0.47	
8		CatBoost	1000, 5	0.44	
9		TabPFN	64	0.44	
10		**XG + LGB+ Cat+PFN**	**Balanced wheights**	**0.42**	**500**
11			Balanced wheights + more wheight for PFN (0.55)	0.398	**250**
12	**Same scaler + better transformer**			**0.387**	
13	Scale and transform		Balanced wheights + more wheight for PFN (0.55) + best xg depth	0.395	
14	**Same scaler + better transformer**			**0.368**	**20**
15	quantile_transformer + standard			0.411	500
16	PowerTransformer alone			0.400	500
17	**Different bad cols + PowerTransformer**		**Reweight after prediction**	**0.371**	**23**
18	**PowerTransformer**			**0.363**	**15**
19			**Reweight after prediction + best Cat depth**	**0.362**	**15**
20			**Reweight after prediction + best PFN**	**0.355**	**11**