Rapport Advanced ML 2

Santiago MARTIN & Léo RINGEISSEN

Data presentation

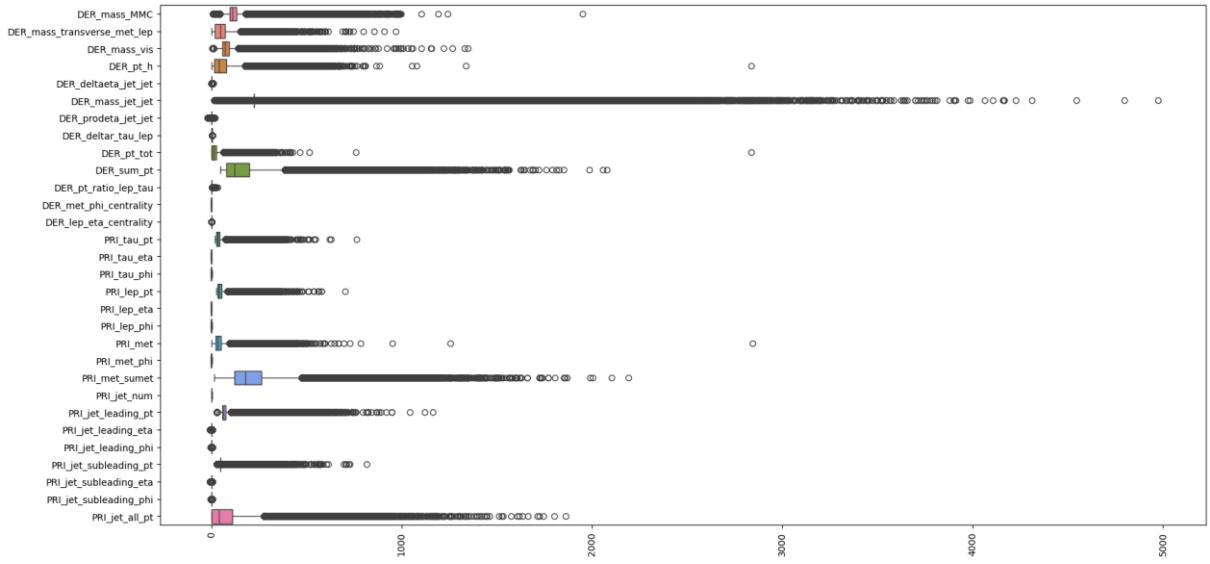
The Higgs Boson dataset is a widely-used benchmark dataset as part of a machine learning challenge by the ATLAS experiment at CERN. It contains 35 features and is based on simulated data used to train models for a binary classification task to determine whether a given event is a signal event or a background event.

Data Preprocessing

We started preparing our data by, dealing with superfluous data. Columns such as *EventId*, *Weight*, *KaggleSet*, and *KaggleWeight* were removed from the dataset. The target column, *Label*, was encoded into binary values, where "s" represents 1 (signal) and "b" represents 0 (background).

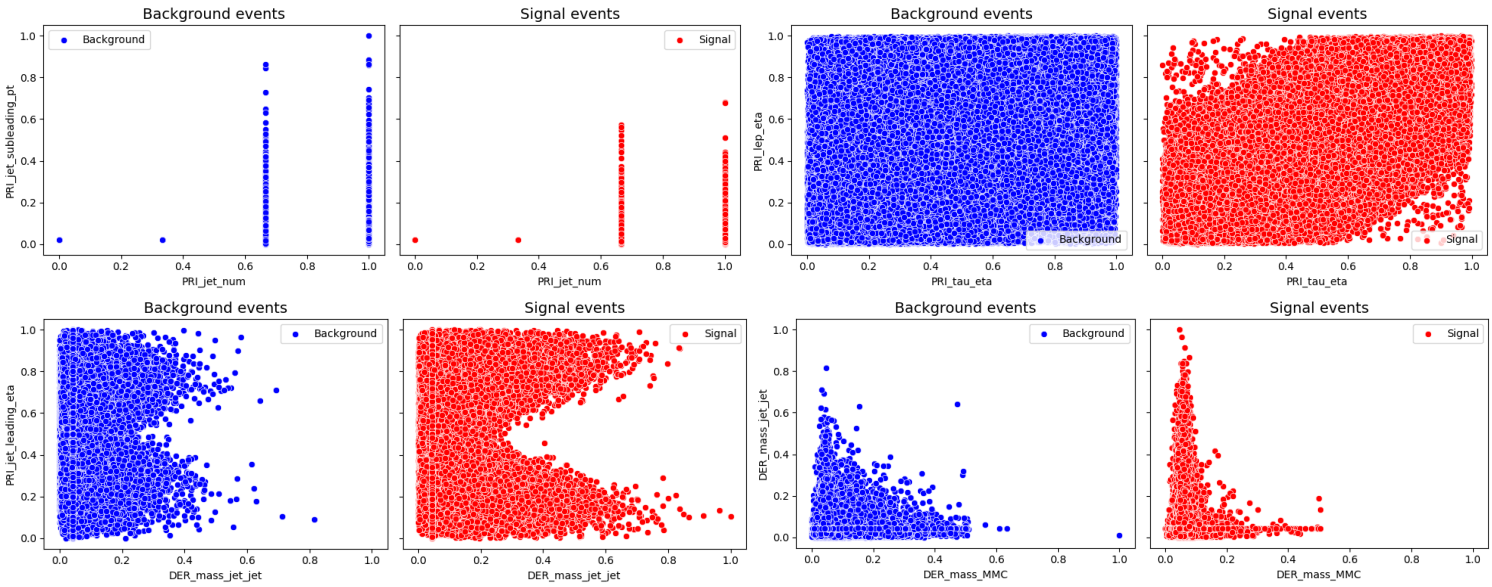
Then, missing values, initially designated by `-999`, were replaced with *NaN* to facilitate processing. The features (`X`) and target labels (`y`) were separated, and missing values in the features were imputed using the median value of each column.

We then generated boxplots for all features to examine the dataset for potential outliers.

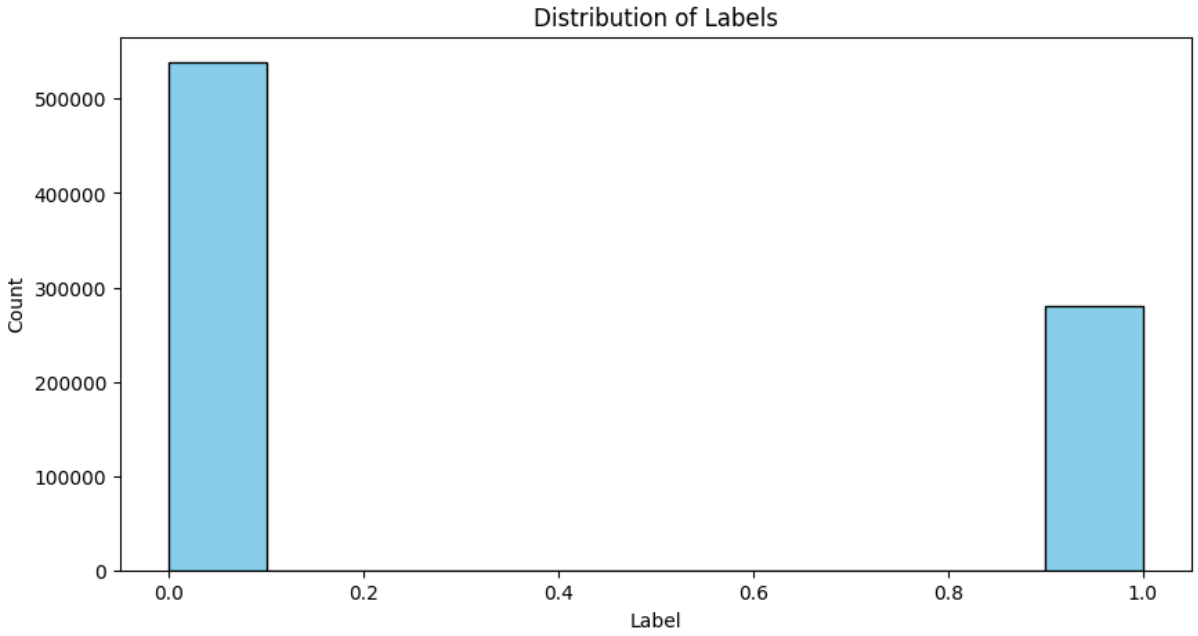


The features were then scaled using the *MinMaxScaler()* from Scikit-Learn, transforming all feature values into the range [0,1]. This step ensures that the data is normalized for algorithms sensitive to normalization.

To explore the data and get some insights, we selected pairs of features that we cross-plotted to explore potential relationships between them. Scatter plots were generated for both signal and background events to visually examine patterns and separability, and some differences in pattern can be visible.



We also plotted the distribution of signal labels across the dataset, to see how balanced it was.



There is some data imbalance, but it is acceptable.

Since the dataset is large and might take lot of computational resources, we implemented PCA to reduce training data size while retaining 90% variance, which lead us from having 30 features to only 9 features.

For the last step, we exported the preprocessed dataset as a csv to save time for future uses of the notebook, and we split the data into train and test datasets, with a ratio of 70:30.

Ensemble Learning

Bagging algorithm

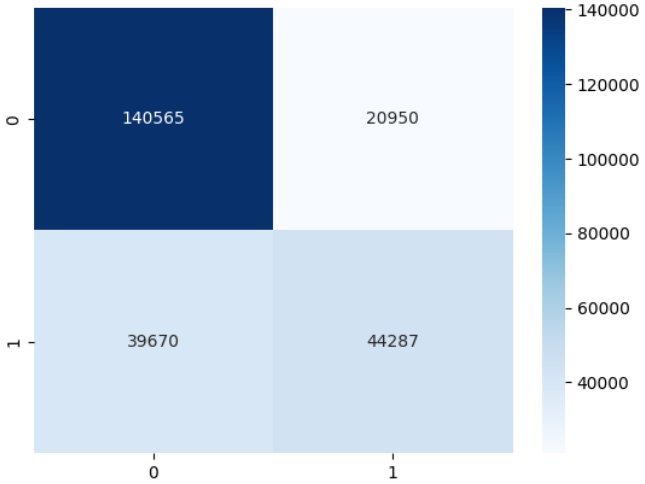
Presentation

For the bagging algorithm, we chose to run a RandomForest model.
The Random Forest Classifier is an ensemble learning algorithm that builds multiple decision trees and merges them to improve classification accuracy and reduce overfitting. Here's an overview of the hyperparameters we used :

```
clf = RandomForestClassifier(
    n_estimators=100,
    criterion='gini',
    max_depth=None,
    min_samples_split=2,
    min_samples_leaf=1,
    min_weight_fraction_leaf=0.0,
    max_features='sqrt',
    max_leaf_nodes=None,
    min_impurity_decrease=0.0,
    bootstrap=True,
    oob_score=False,
    n_jobs=None,
    random_state=None,
    verbose=0,
    warm_start=False,
    class_weight=None,
    ccp_alpha=0.0,
    max_samples=None)
```

Results

To analyze our models' performances, we chose to generate the correlation matrix, along with the classification report and the overall evaluation metrics. Our global metrics only consider the positive class, since our data is imbalanced.



	precision	recall	f1-score	support
0	0.78	0.87	0.82	161515
1	0.68	0.53	0.59	83957
accuracy			0.75	245472
macro avg	0.73	0.70	0.71	245472
weighted avg	0.75	0.75	0.74	245472

```
Accuracy:  0.7530471907182896
F1 Score:  0.5936833920935158
Recall:  0.5274962183022261
Precision:  0.6788632217913148
Time (s):  348.3387773036957
```

This model performed well on the dataset, achieving an accuracy of 0.75. The other metrics also show good consistency in both class predictions, although the recall and the f1 score for the signal class is a bit low.

Boosting algorithm

Presentation

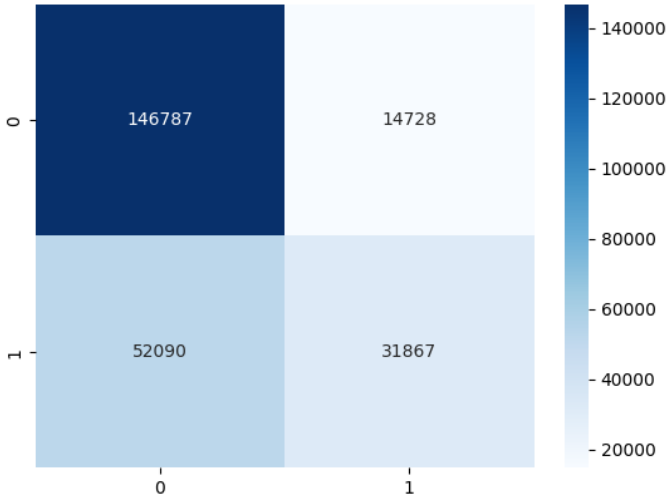
For the boosting algorithm, we chose to run and XGBoost Classifier.

XGBoost (Extreme Gradient Boosting) is a scalable machine learning algorithm that uses gradient boosting for classification and regression tasks. It builds a strong predictive model by combining the outputs of multiple weak models (like decision trees) in an iterative way.

Here's an overview of the hyperparemeters we used :

```
clf = GradientBoostingClassifier(
    loss='log_loss',
    learning_rate=0.1,
    n_estimators=100,
    subsample=1.0,
    criterion='friedman_mse',
    min_samples_split=2,
    min_samples_leaf=1,
    min_weight_fraction_leaf=0.0,
    max_depth=3,
    min_impurity_decrease=0.0,
    init=None,
    random_state=None,
    max_features=None,
    verbose=0,
    max_leaf_nodes=None,
    warm_start=False,
    validation_fraction=0.1,
    n_iter_no_change=None,
    tol=0.0001,
    ccp_alpha=0.0)
```

Results



	precision	recall	f1-score	support
0	0.74	0.91	0.81	161515
1	0.68	0.38	0.49	83957
accuracy			0.73	245472
macro avg	0.71	0.64	0.65	245472
weighted avg	0.72	0.73	0.70	245472

```
Accuracy:  0.727797875114066
F1 Score:  0.4881886144984374
Recall:  0.3795633479042843
Precision:  0.6839145831097757
Time (s):  270.936133146286
```

This model performed a bit worse than the RandomForest model, since all metrics except precision got worse. Only pros of this algorithm is that it ran for a shorter amount of time.

Evloutionary learning with DEAP

Presentation

We used Evolutionary Learning using Genetic Programming, included in the DEAP library, specifically for a classification task. The goal is to evolve mathematical expressions (functions) that can classify data into two categories. These functions are composed of basic mathematical operations like addition, subtraction, and multiplication. The evolutionary process aims to maximize classification performance, using F1 score as the fitness metric.

Setup :

Primitive Set

The first step is to define the primitive set for a classification problem. Here, we set the *arity* parameters so that the models takes one input for each feature, which means 9 inputs.

```
# Define a primitive set for a classification problem
pset = gp.PrimitiveSet("MAIN", arity=X_train.shape[1])  # One input for each feature
```

The primitive set defines the building blocks (operations) that can be used to construct individuals (models). Here, the operations available are additions, subtractions and multiplications.

```
# Add basic mathematical operations
pset.addPrimitive(np.add, 2)
pset.addPrimitive(np.subtract, 2)
pset.addPrimitive(np.multiply, 2)
```

Fitness functions

The model needs to be able to evaluate individuals on their ability to solve the classification task. Therefore, we setup a fitness function that is going to be based on maximizing the f1-score. We then setup the individual object as a tree of primitive function that can be evaluated by the fitness function previously created. It is then used to rank individuals during the phase of training. This function itself does not calculate the score, but receives it from the *evaluate_individual* function, which is presented later.

```
# Define fitness function (maximize accuracy or F1-score)
creator.create("FitnessMax", base.Fitness, weights=(1.0,))  # Maximize
creator.create("Individual", gp.PrimitiveTree, fitness=creator.FitnessMax)
```

Toolbox setup

The toolbox is used to register function that are going to be used to create and evolve individuals in the population. It is made of :

- *expr* : generates a random individual using the *genHalfAndHalf* method, which creates a random tree of operations with a specified depth?
- *individual* : initizalizes individuals using the generated expressions
- *population* : creates a population made of individuals

Evaluation function

The *evaluate_individual* function evaluates an individual by compiling it into a function using *gp.compile*, and then computing its fitness score, which is the f1-score in this case. It then returns the value to the fitness function which ranks the individuals.

```
# Define evaluation function
def evaluate_individual(individual):
    # Compile the individual to a callable function
    func = gp.compile(expr=individual, pset=pset)

    # Apply the function to the rows of X_train
    y_pred = [func(*row) for row in X_train_list]

    # Convert predictions to binary labels
    y_pred = np.where(np.array(y_pred) > 0.5, 1, 0)

    # Return F1 score as the fitness
    return f1_score(y_train, y_pred),
```

Genetic Operators :

Genetic operators are operations used by the evolutionary model to edit and select the individuals for the next generation. We used 3 of them :

- *Crossover* : combines two individuals to produce a new one, by swapping subtrees between two individuals
- *Mutation* : randomly alters parts of an individual by changing a random subtree to a new expression
- *Selection* : selects individual for the next generation using tournament selection (with a size of) based on their fitness

```
# Register genetic operators
toolbox.register("mate", gp.cxOnePoint)
toolbox.register("mutate", gp.mutUniform, expr=toolbox.expr, pset=pset)
toolbox.register("select", tools.selTournament, tournsize=3)
```

eaSimple

Presentation

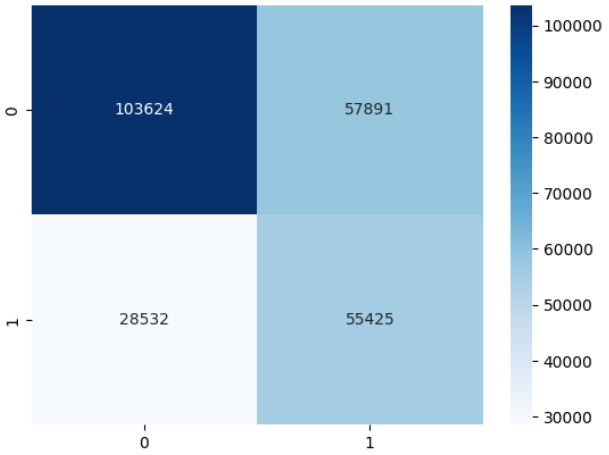
eaSimple is an evolutionary algorithm where the entire population is replaced by the offspring generated through crossover and mutation. It incorporates elitism, meaning the best individuals from the current generation are always carried over to the next. This simple approach works well for smaller problems or when a straightforward evolutionary process is needed.

We ran this model for 40 generations, with a population of 300.

```python
from deap import algorithms
# Population initialization
population = toolbox.population(n=300)

# Run the algorithm
final_pop, log = algorithms.eaSimple(population, toolbox, cxpb=0.5, mutpb=0.2, ngen=40, verbose=True)
```

Results



	precision	recall	f1-score	support
0	0.79	0.66	0.71	161515
1	0.50	0.65	0.57	83957
accuracy			0.66	245472
macro avg	0.64	0.66	0.64	245472
weighted avg	0.69	0.66	0.66	245472

```
Test Accuracy: 0.655528940164255
Test F1 Score: 0.5653170751768383
Test Recall: 0.6549185892778446
Test Precision: 0.4972822892078393
Time (s): 35830.54983274923
```

The first thing to notice here is that this model ran for a huge amount of time (almost 10 hours). Running it for 40 generations might have been too much, and we got surprised by the fact that its computational time got higher with the generation count, since the individual tend to become more complex. Sadly, this did not yield better results, as most global metrics are lower than for ensemble learning models. Something surprising we noticed, is that ensemble learning models had a lot of false negatives, whereas this model has more false positives.

eaMuPlusLambda

Presentation

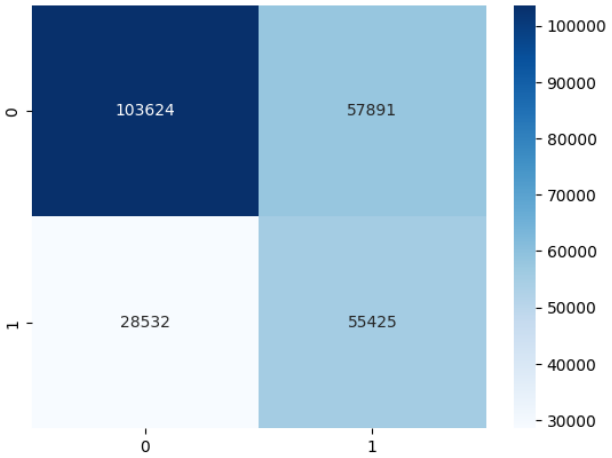
eaMuPlusLambda is an evolutionary strategy where a fixed number of parents (Mu) and a larger number of offspring (Lambda**)** are generated. The next generation is formed by selecting the best individuals from both parents and offspring. This method encourages **diversity** and allows for more exploration of the search space compared to eaSimple.

We ran this model for 25 generations (after seeing the computing time of the previous model), with a population of 300.

```python
from deap import algorithms
# Population initialization
population = toolbox.population(n=300)

# Run the algorithm
final_pop, log = algorithms.eaMuPlusLambda(population, toolbox, mu=200, lambda_=300, cxpb=0.5, mutpb=0.2, ngen=25, verbose=True)
```

Results



	precision	recall	f1-score	support
0	0.78	0.64	0.71	161515
1	0.49	0.66	0.56	83957
accuracy			0.65	245472
macro avg	0.64	0.65	0.63	245472
weighted avg	0.68	0.65	0.66	245472

```
Test Accuracy: 0.6479313322904445
Test F1 Score: 0.5619116655599093
Test Recall: 0.660159367295163
Test Precision: 0.48911892407074026
Time (s): 15945.515755631543
```

The results obtained are close to identical from the eaSimple algorithm. Given its computing time (over 4 hours), the results are once again disappointing. This algorithme also has way more false positives than false negatives.

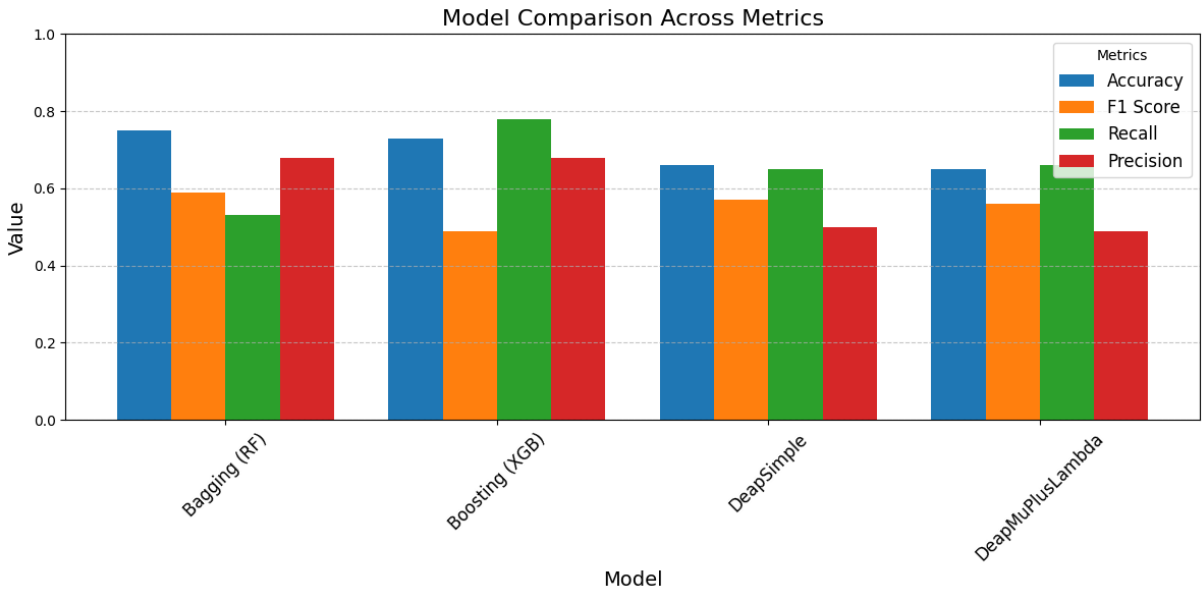
Models comparison

Metrics table

In the table below, we summarized the global metrics on the positive class for each model.

Model	Accuracy	F1 Score	Recall	Precision	Runtime (s)
Bagging (RF)	0.75	0.59	0.53	0.68	348
Boosting (XGB)	0.73	0.49	0.78	0.68	270
DeapSimple	0.66	0.57	0.65	0.50	35830
DeapMuPlusLambda	0.65	0.56	0.66	0.49	15945

We also plotted these metrics to be able to visualize the differences.



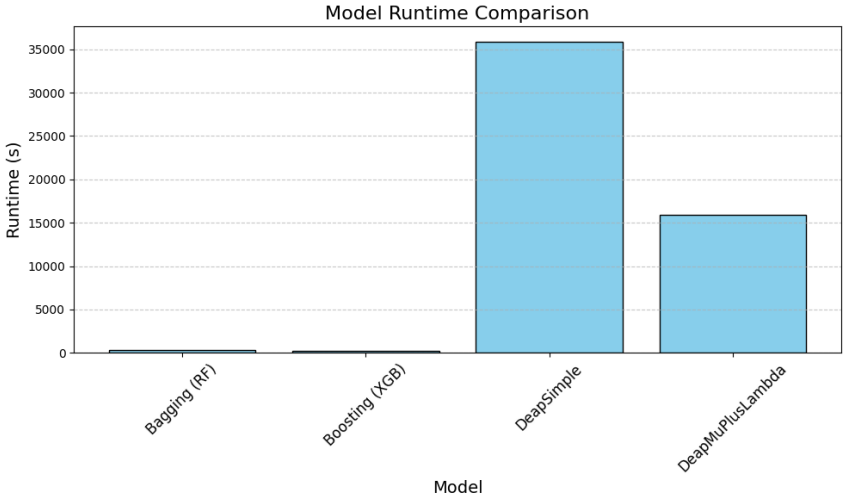
The bar plot highlights the performance of four models (Bagging (RF), Boosting (XGB), DeapSimple, and DeapMuPlusLambda) across Accuracy, F1 Score, Recall, and Precision:

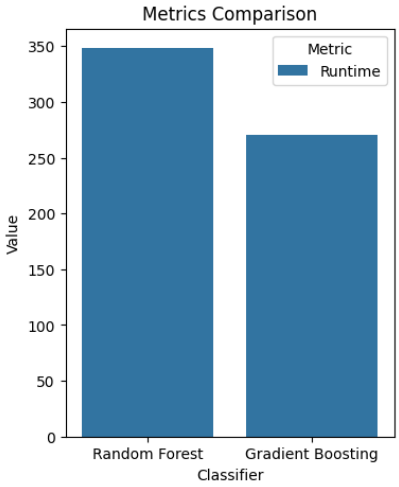
In terms of accuracy, Bagging (RF) achieves the highest value at 0.75, demonstrating its overall reliability in correctly classifying both positive and negative cases. Boosting (XGB) follows closely with an accuracy of 0.73, indicating it is also a robust model but slightly less consistent than Bagging. The evolutionary models, DeapSimple and DeapMuPlusLambda, show lower accuracy scores at 0.66 and 0.65, respectively, suggesting they are efficient at correct classifications overall.

For the F1 score, which balances precision and recall, Bagging (RF) again leads with a value of 0.59, reflecting its ability to manage the trade-off between false positives and false negatives effectively. DeapSimple achieves an F1 score of 0.57, closely trailing Bagging, and shows a reasonable balance between sensitivity and precision. DeapMuPlusLambda scores slightly lower at 0.56, highlighting its relatively weaker balance compared to DeapSimple. Boosting (XGB) performs the worst in this metric, with an F1 score of 0.49, which suggests it struggles to maintain a balance despite excelling in other areas.

Regarding recall, Boosting (XGB) significantly outperforms the other models with a recall of 0.78, making it highly effective at identifying positive cases. This feature makes Boosting particularly valuable in applications where minimizing false negatives is critical. DeapMuPlusLambda, with a recall of 0.66, and DeapSimple, at 0.65, follow Boosting in identifying true positive cases. Bagging (RF), with a recall of 0.53, performs the weakest in this area, indicating that it misses a considerable number of positive cases.

In the precision metric, Bagging (RF) and Boosting (XGB) both score the highest, with values of 0.68, showcasing their effectiveness in avoiding false positives when predicting positive cases. DeapSimple achieves a precision of 0.50, reflecting a noticeable drop in its ability to avoid false positives, while DeapMuPlusLambda trails slightly behind with a precision of 0.49, demonstrating a similar weakness.





For the runtime comparison, the Deap models are obviously inefficient on this specific classification tasks, since their high computational cost does not yield identical or better results. The Random Forest algorithm, which is a bit better than the XGB model, has a higher runtime, so both models have a close efficiency-to-runtime ratio.

Conclusion

Bagging emerges as the strongest model overall, particularly excelling in accuracy and precision, making it well-suited for general-purpose tasks. Boosting stands out for its high recall, making it the best choice for scenarios requiring the minimization of false negatives. The evolutionary algorithms, DeapSimple and DeapMuPlusLambda, provide balanced alternatives but underperform in terms of accuracy and precision, and come with the cost of a huge training time. Ultimately, the choice of the model depends on the specific requirements of the problem, such as whether prioritizing sensitivity (recall) or overall reliability (accuracy) is more important.

Deap models are probably too complex for this task, but we thought that this could be due to the fact that we applied dimension reduction to the dataset, which could have erased some links between features that more complex models could have captured.