UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Dipartimento di Ingegneria ”Enzo Ferrari”

Corso di Laurea Magistrale in Ingegneria Informatica

Optimize Heterogeneous Ensemble Search

Supervisor:

Prof. Simone Calderara

Candidate:

Francesco Zampirollo

External Supervisors:

Giovanni Davoli

Alberto Zurli

A.Y. 2023/2024

”Difficulties are things that show a person what they are.”

— **Epictetus**

Riassunto Analitico

Negli ultimi anni, l'impiego dei metodi di Ensemble è diventato sempre più diffuso, rappresentando un paradigma altamente efficiente nel campo del Machine Learning. Questo approccio combina un insieme di singoli modelli definiti deboli per produrre predizioni robuste, trovando applicazione in numerosi settori.
Un problema noto all'interno dell'Ensemble Learning è avere diversità tra i modelli, consentendo l'esplorazione di pattern differenti all'interno dei dati e garantendo eterogeneità.
Questa tesi illustra lo sviluppo di un progetto volto all'ottimizzazione del processo di esplorazione nello spazio di ricerca dei parametri di ensemble eterogenei, costituiti da architetture e task differenti, sostenuto da uno studio relativo all'introduzione della diversità.
Il progetto è stato sviluppato presso Axyon AI, una società FinTech che supporta i gestori di asset quantitativi attraverso strategie guidate da Machine & Deep Learning.

Abstract

In recent years, the use of Ensemble methods has grown rapidly, becoming a highly effective approach in Machine Learning. This technique combines multiple individual models, known as weak learners, to produce stronger predictions and is used in various fields. A key challenge in Ensemble Learning is ensuring diversity among models, which helps explore different data patterns and maintain heterogeneity.

This thesis describes a project focused on optimizing the exploration of the search space for heterogeneous ensemble parameters, involving different architectures and tasks, and includes a study on promoting diversity. The project was developed at Axyon AI, a FinTech company that supports quantitative asset managers using Machine & Deep Learning strategies.

Contents

List of Figures

1. Introduction

1.1 Business Context

Founded in 2016 as an innovative start-up combining quantitative finance and AI, Axyon AI is a company located in Modena. The primary goal is to support asset manager with ad-hoc strategies built with powerful proprietary technologies of Machine and Deep Learning. From the date of its foundation to today, Axyon has become a benchmark in the sector thanks to the values that have always guided it:

1. Transparency
2. Growth and Learning
3. Excellence and Passion

A key point is the constant quest for new challenges and innovation, that motivates the company to improve and enhance its processes, ensuring a better service.

1.2 Ensemble Learning

In the last few years ensemble learning has enjoyed growing attention due to high efficiency and versatility in multiple areas. An ensemble is a collection of models trained to achieve the same task. The basic concept of EL is the aggregation of the single predictions in order to obtain a final, strong and more accurate prediction than single **weak learners**.

1.2.1 Ensemble Intuition

Suppose we have an ensemble of binary classification functions: $f_k(x)$ for $k = 1, ..., K$. Suppose that, on average, they have the same expected error rate:

$$\epsilon = E_{p(x,y)}[y \neq f_k(x)] < 0.5 \tag{1.1}$$

but that the errors they make are independent.
The intuition is that the majority of the K classifiers in the ensemble will be correct on many examples where any individual classifier makes an error. A simple majority vote can significantly improve classification performance by decreasing variance in this setting.

Proof of Majority Voting

Suppose again to have an ensemble of K binary classifiers with accuracy probability $\alpha > 0.5$, the majority voting ensemble makes an error when K/2+1 classifiers make a wrong predictions and error follows the cumulative binomial distribution:

$$P(x \leq k) = \sum_{i=0}^{k} \binom{K}{k} \alpha^i (1-\alpha)^{K-i} \tag{1.2}$$

so the ensemble accuracy probability will be:

$$1 - P(x \leq k)$$

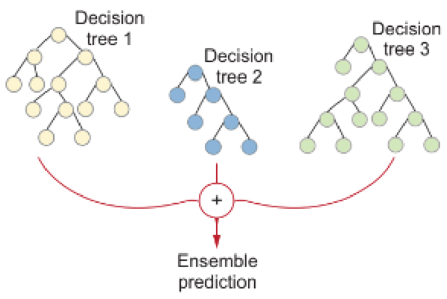
1.2.2 Ensemble Taxonomy

Ensemble methods can be classified into two types, depending on how they are trained: parallel and sequential ensembles.

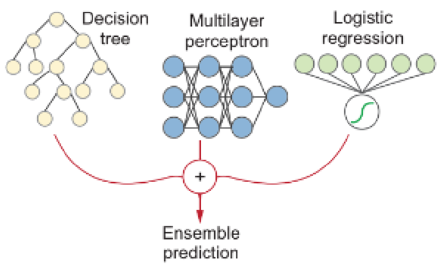
- **Parallel Ensemble** methods, train each component base model independently of the others, which means that they can be trained in parallel. This category can be also divided into:

    1. *Homogeneous parallel ensembles*: all the base learner are of the same type (e.g decision trees) and trained using the same base-learning algorithm.
    2. *Heterogeneous parallel ensemble*: base learners are trained using different base-learning algorithms.

- **Sequential Ensemble** methods, exploit the dependence of base learners. During training, sequential ensembles train a new base learner in such a manner that it minimizes the errors produced by the base learner trained in the previous step.



(a) Homogeneous Ensemble



(b) Heterogeneous Ensemble

Figure 1.1: Ensemble Taxonomy

1.2.3 Bagging

Bootstrap aggregation or *Bagging* is the most basic parallel ensemble method, that takes a single training set *Tr* and randomly sub-samples from it K times (with replacement) to form K training sets $Tr_1,...,Tr_K$.
Each of these is used to train different instance of the same classifier obtaining K classification functions: $f_1(x),...,f_K(x)$. The errors won't be **totally independent** because the data sets aren't independent, however the random re-sampling usually introduce enough diversity to reduce **variance** and give improved performance.

Bagging Proof

Suppose every classifiers learn:

$$y = f_i(x) + e_i$$

where e is the error of i-th classifier.
The average error of M classifiers on the dataset D is:

$$\epsilon_{AV} = \frac{1}{M}\sum_{i=1}^{M} E_D[e_i]$$

The bagged prediction is:

$$y_{bagged} = \frac{1}{M}\sum_{i=1}^{M} f_i(x) + e_i(x)$$

so, the average error of the bagged classifier is:

$$\epsilon = E_D[\frac{1}{M}\sum_{i=1}^{M} e_i(x)]$$

Random Forest

A kind of extension of bagging, *Random Forest* introduce additional randomization to ensemble. Specifically, refers to an ensemble of randomized *decision trees* as base estimators and perform bagging to generate training subset.

Component

- **Feature Vector**: feature of data sample.
- **Split Functions**: assigned to each node of tree, are functions relative to specific feature used to evaluate a sample.
- **Thresholds**: values used to evaluate the split functions going deep into the tree.
- **Predictions**: single prediction obtained once arrived in a leaf node.

Usually a single tree choose threshold and features into the features pool that maximize a particular function (e.g **Gain Information**) and obtained all predictions from the leaf nodes combine them for a single global prediction.

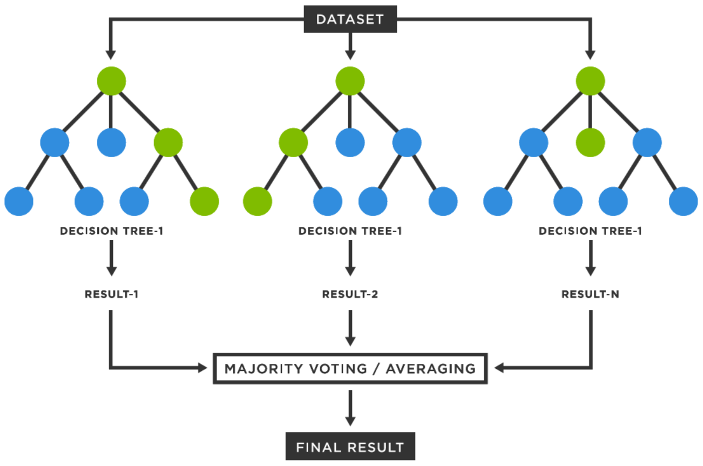


Figure 1.2: Random Forest

1.2.4 Boosting

Boosting is one of the most popular sequential ensemble method, the principal idea is train models in order to correct the models errors.

Operating

Boosting start from dataset D, sequentially train equal classifier f_i, focusing on errors from f_{i-1}. Assign to every $x_k \in D$ equal weights $w_k = \frac{1}{N}$
Iterating over the boosting stages:

1. sample dataset D^i from D using weights $\{w_k^i\}_{k=1}^N$ as the sampling probability for every record x.
2. train the i-th classifier f_i on D^i, measure the accuracy and record it as α_i.
3. if x_k has wrongly classified augment its weight at the next stage w_k^{i+1} and re-normalize weights.

The final decision rule is:

$$y(x_{new}) = sign(\sum_i^M \alpha_i f_i(x_{new}))$$

Important: every classifier in the boosting chain must have an accuracy ≥ 0.5

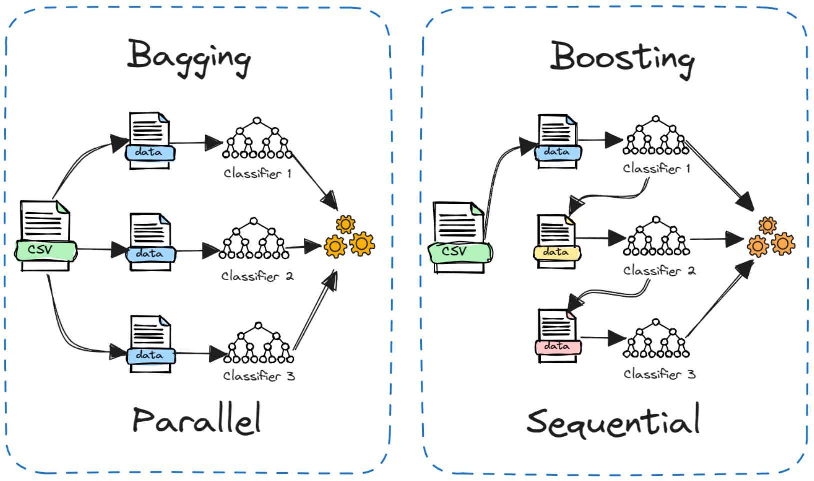


Figure 1.3: Bagging vs. Boosting

1.2.5 Stacking

Stacking is the most common meta-learning methods, gets its name because it stack second classifier (also called *meta learner*) after the first layer of the base estimators. The general stacking procedure has two steps:

1. First layer: fit base estimators on the training data. This step aims to create diverse and heterogeneous set of classifiers.

2. Second layer: using a new dataset from the predictions of the first layer (meta-features) obtain a final prediction to another estimator.

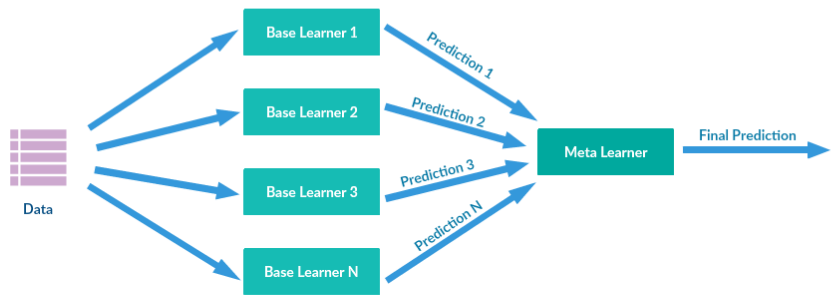


Figure 1.4: Stacking procedure

1.2.6 Bias Variance trade-off

Every model, producing predictions, is subject to these two components: **bias**, the accuracy of the classifier, and **variance**, the precision of the trained classifier on different training sets. In the EL it is possible to control these components: models with low bias tend to have high variance and vice versa. On the other hand, it is known that the ensemble aggregation process (e.g. averaging) has a smoothing (variance-reducing) effect. The objective of EL is therefore to obtain models with relatively fixed bias and, combining their outputs, to reduce the variance.

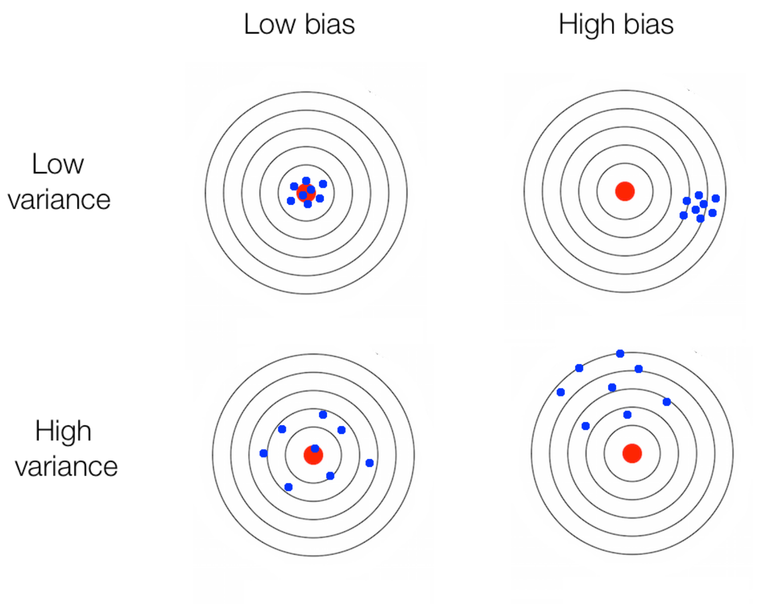


Figure 1.5: Bias-Variance Tradeoff

1.3 Genetics Algorithms

Also called **Evolutionary Algorithms**, these types of algorithms are inspired by the theory of natural evolution, especially the phenomenon of adaptation. This is considered a form of optimization, referring to the gradual change in the algorithm's properties in response to changes in the environment (or search space). They are used for both constrained and unconstrained optimization problems, aiming to find high-quality solutions at the cost of execution time and slow convergence. A key concept in evolution theory is the notion of *population*, a group of individuals of the same species that can produce evolved and adapted offspring. In an optimization problem, each individual have a fitness value, which is used to evaluate and select them. Through various processes like mutation, combination, and crossover, individuals undergo transformations to maximize their fitness and explore the search space for the best solution.

1.4 Random Search Algorithms

This family of algorithms is mainly used for global optimization problems, with the feature of quickly finding a good solution without guaranteeing the global optimum. The main goal of these algorithms is to create random configurations of hyperparameters in a well-defined search space, looking for the combination that maximizes a certain evaluation metric. Depending on the field of application, there are various methods to navigate the search space, accompanied by different stopping criteria.

1.5 Exploration vs Exploitation

Many optimization problems are subject to this kind of dilemma, based on these two concept: finding a local optimum around a point in the space (exploitation) against searching optimum in all the space with all that it involves (exploration). Depending on the target, is possible to implements algorithms oriented by one of those directions or using a trade-off that allow to moving on the searching space changing both the directions.

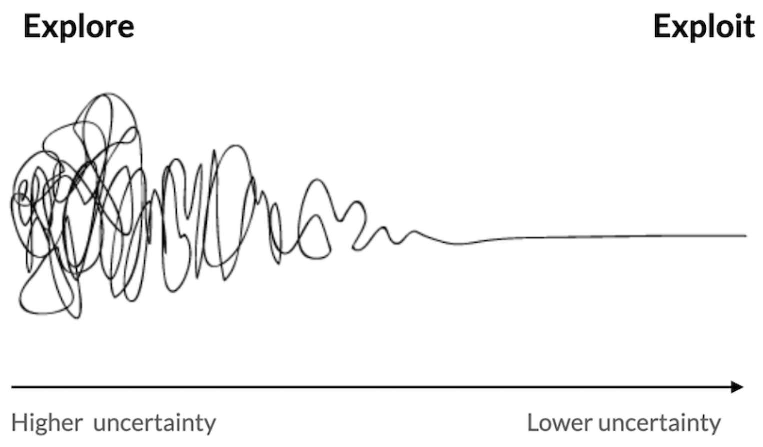


Figure 1.6: Exploration vs Exploitation

1.6 Motivations

Among the main business methodologies used to generate strategic predictions, the use of ensembles plays an important role. For a while, an algorithm with a strong genetic influence was used to create the pool of models that would form the ensemble. This first approach was replaced by a second algorithm more oriented towards Random Search, but still a hybrid of both, which over time revealed significant limitations, in particular:

1. **Invasive Architecture**: during the evolution of the genetic algorithm, simpler architectures like Dense Neural Networks are trained and evaluated much faster than more complex architectures like Convolutional Neural Networks. This process leads to the gradual elimination of Convolutional architectures within the population.

2. **Needs of different architectures**: it became increasingly evident after many applications that creating ensembles of different architectures leads to better results. This is because different architectures are more likely to learn different patterns in the data, and their diversity can ensure much more robust predictions.

3. **Exploitation vs Exploration**: all optimization problems face this dilemma, and the current version of the genetic algorithm is too biased for exploitation, leading the population to converge towards a purely local optimum.

4. **PHP Implementation**: the algorithm is implemented in PHP, making it difficult to maintain over time and limiting the possibilities of improving the previous limitations.

5. **Intrinsic Properties**: due to the influence of principles derived from genetic algorithms, the algorithm is particularly sophisticated. Additionally, some intrinsic characteristics seem forced when applied to the specific data and final objectives for which it is used.

2. Road to new Random Search

In this section will be presented the different steps which have led to build a new, dynamic and optimized, version of Random Search algorithm.
Starting from the previous version with its limitations that necessitated this project, the implementation pipeline: design, refactor, and optimization will be detailed.

2.1 One-to-One Porting

In order to figure out the main procedure of the previous algorithm, the first step of the pipeline is a kind of translation from PHP code to Python.
Since the old algorithm was inspired by concepts related to genetic algorithms, the heart of functionality was the ***policy*** class, which from different extensions were made depending on the type of evolutionary process.

2.1.1 Principal Components

Fundamental components of the previous version were:

1. **Pool**: contains all the information relative to ensemble desired features such as: *model architecture, number of elements (e.g population), type of problem (e.g regression) and subtype (single or multiple labels), dataset handler parameters, type of policy class, policy parameters* and additional ones.

2. **Job**: single instance of model (with related attributes *e.g feature mask, model architecture, model parameters and so on*) generated and ready to train; each job is assigned to a specific *pool id*.

3. **Policy**: class related to policy used into searching process (there are several Policy derived from evolutionary process) and each of them has specific behaviour during the algorithm.

Final Goal

Generate and train a number of *Jobs* equal to the number of jobs required by *max-Population* parameter of the pool; until then, the algorithm will continue to execute iteratively in a while loop.
The entire process starts from a pre-existing *pool* that contains all the information specified for the desired exploration.

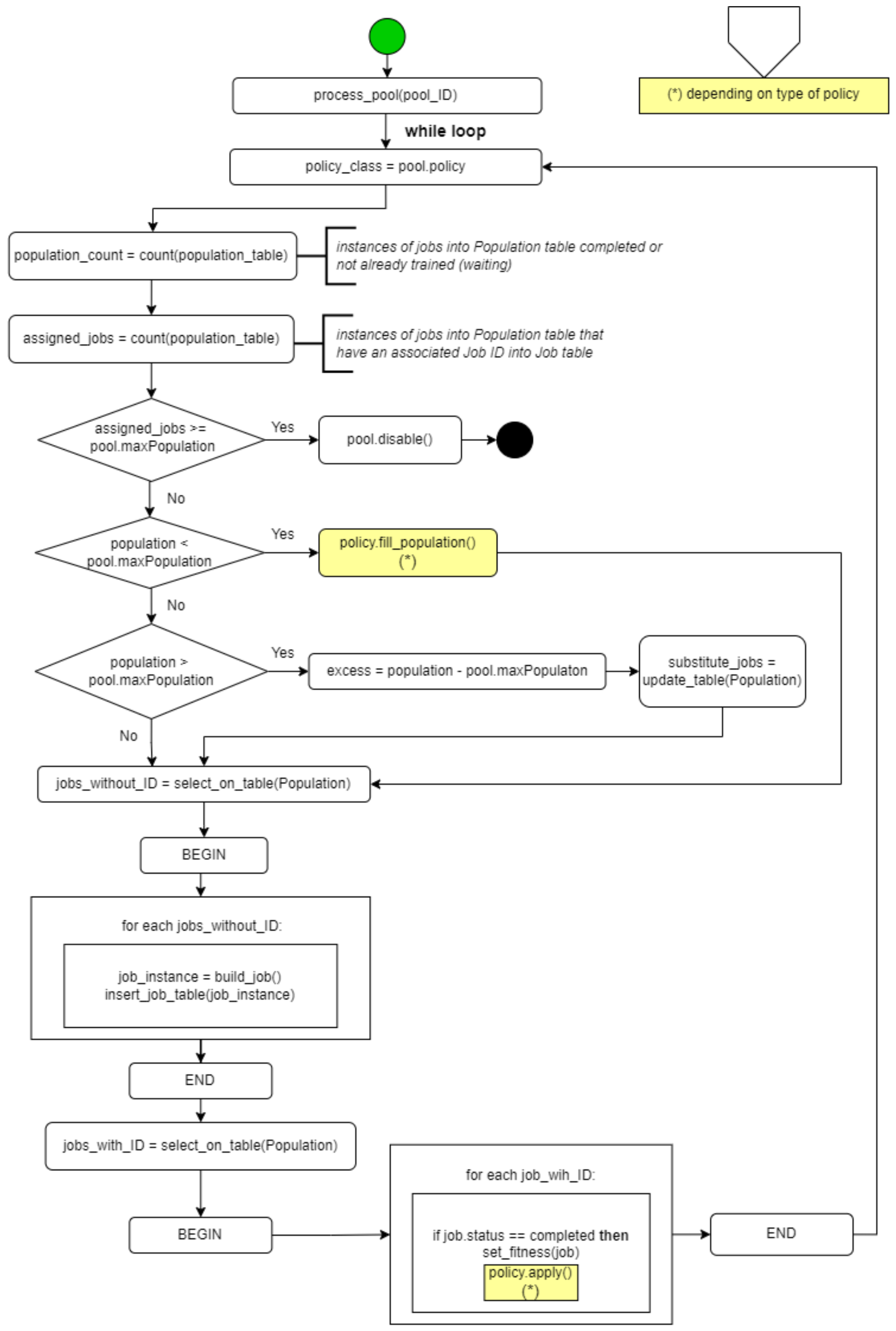
Concept of Population

During execution, jobs initially will be created in a table called *Population*, where they will be temporarily idle. For each instance in this table there are several attributes, such as:

- **Model Params**: model parameters of a specific architecture
- **Feature Mask**: binary mask of selected features
- **Job ID**: id of the associated instance into the *Job* table (initially NULL)
- **Fitness and Fitness Validation**: where metrics will be saved when training is completed (on different splits, e.g. cv-test)
- **Origin**: the type of job generation (e.g. creation, mutation, crossover)
- **In Population To**: by default, this is NULL; when a job is trained, it is set to the current time of completion.

Next, jobs will be created in a second table called *Job*, where they will be ready to be trained and additional fields (related to data saved during training) will be filled; this double creation process is mainly used to insert and substitute individual job instances within the population, which is the parameter that manage the stopping criterion.

2.1.2 Flow Chart Old Version



(*) depending on type of policy
process_pool(pool_ID)
while loop
policy_class = pool.policy
population_count = count(population_table)
instances of jobs into Population table completed or not already trained (waiting)
assigned_jobs = count(population_table)
instances of jobs into Population table that have an associated Job ID into Job table
assigned_jobs >= pool.maxPopulation
Yes
pool.disable()
No
population < pool.maxPopulation
Yes
policy.fill_population() (*)
No
population > pool.maxPopulation
Yes
excess = population - pool.maxPopulaton
substitute_jobs = update_table(Population)
No
jobs_without_ID = select_on_table(Population)
BEGIN
for each jobs_without_ID:
job_instance = build_job()
insert_job_table(job_instance)
END
jobs_with_ID = select_on_table(Population)
BEGIN
for each job_wih_ID:
if job.status == completed then
set_fitness(job)
policy.apply() (*)
END

Figure 2.1: Flow Chart PHP algorithm

2.1.3 Policy Random

The most commonly used policy is the **Random Policy**. This type of policy was introduced to adapt the evolutionary algorithm to a kind of *Random Search*. The random policy is an extension of the generic *Policy* class, which contains generic methods that distinguish between different types of policies (e.g. *apply_policy*, *fill_population*, and others).

Feature

The version of this policy that simulates random search focuses primarily on two elements:

1. **Feature Mask**: consists of a defined number of features (contained within the Pool information) as a binary mask vector.
2. **Model Parameters**: within the Pool, the architecture for exploration is defined, and this element contains a dictionary with all the parameters that will complete the desired architecture.

Making Population

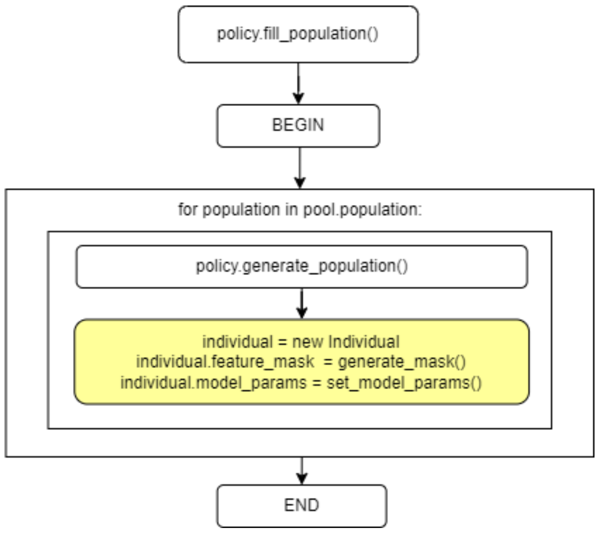

policy.fill_population()
BEGIN
for population in pool.population:
policy.generate_population()
individual = new Individual
individual.feature_mask = generate_mask()
individual.model_params = set_model_params()
END

Figure 2.2: Fill Population method

As shown in 2.2, the first step in generating the population is done using the *fill_population* method of the policy class. This method is important because it creates the first instance of an *Individual*, assigning it both a *feature mask* and *model parameters*.

The **feature mask** is created in a completely **random** way. This mask is binary, meaning each position can be either 0 or 1, with an equal chance to obtain both of them. This randomness is key, because it ensures a wide variety of features mask, allowing the algorithm to explore a large range of possibilities during optimization.

Similarly, the **model parameters** are chosen **randomly** from a repository containing different configurations for the same model architecture. This random selection ensures that the population starts with a different set of configurations, which is important for exploring the search space effectively.
At the end, all the individuals are added to the *Population* table.

Apply Policy

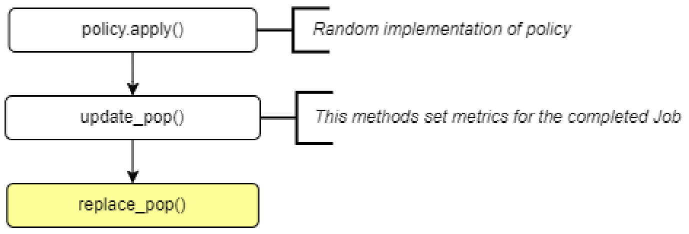

policy.apply()
Random implementation of policy
update_pop()
This methods set metrics for the completed Job
replace_pop()

Figure 2.3: Apply Policy method

When a job complete its training, this method ensures the progression of the exploration process, particularly through the *replace_pop* method. With other types of policies, a new individual is created by inheriting from the completed one; instead, with the *random policy*, a new individual is generated randomly using the same method employed for creating the initial population.

Prospective

Combining a random feature mask and random parameters for each individual introduces a level of randomness that creates weak learners oriented towards a balance between exploration and exploitation. This approach allows the model to explore different parts of the search space, as the random feature mask ensures a diverse selection of features, while the random parameters introduce variability in how these features are weighted and interpreted.

However, the method tends to favor exploitation because, as the population evolves, certain feature masks and parameter combinations that show better performance are likely to be reinforced. This leads the algorithm to focus more on optimizing known good solutions rather than continuously exploring new ones. This focus on exploitation helps the model to find effective solutions more quickly, but it also risks missing better options that could be discovered through more exploration. One of the main reasons for this is the use of the same architecture across all individuals, even though different parameters are applied.

2.2 Refactor Components

Given the limitations discussed in the previous section, it is clear that to allow for greater exploration, it is necessary to use **heterogeneous ensembles** supported by a more dynamic version of the algorithm, particularly in terms of parameter exploration. To create this software structure, it was necessary to implement and integrate the following components:

1. **Baselines**: repository of different architectures, created ad-hoc for *regression* and *classification* problems (detailed in the next chapter).

2. **Optuna Framework**: automatic hyperparameter optimization software framework, used to optimize the parameter space for each architecture during the exploration.

3. **Routine Default**: a specific type of feature mask assigned to a defined number of jobs, that can be included within the ensemble in order to obtain dedicated weak learner that adopt forms not otherwise explored.

2.2.1 Optuna Framework

Optuna is an optimize open-source framework particulary designed for machine learning, used to automate hyperparameter search.

The main concept behind Optuna is the mechanism of **sampler**, an agent class used to smart parameter sampling. Through sampler, Optuna dynamically balances the exploration of new areas of the parameter space with the exploitation of already promising areas.

Another important concept is the **study**, a process of optimization managed using **trials** where each of these is a single evaluation of a parameter configuration.

Random Sampler

Among all the available sampler, for our purposes, the most suitable is the **Random Sampler**: based on concept of *independent sampling*, which means that determines a value of a single parameter without considering any relationship between parameters.

Combinations - V1

Initially, during the development of the **Baselines**, the first idea of integrating Optuna was designed and implemented in the pool of the single available architecture (Neural Network), using the set of different parameters employed by the previous version of the algorithm.

The idea was to develop a function that, by iterating through all the existing parameter configurations of the Neural Network, would generate a dictionary of variability containing:

- **keys**: name of single parameter that assume different values into some or all configurations.
- **values**: list of values assumed by relative parameter

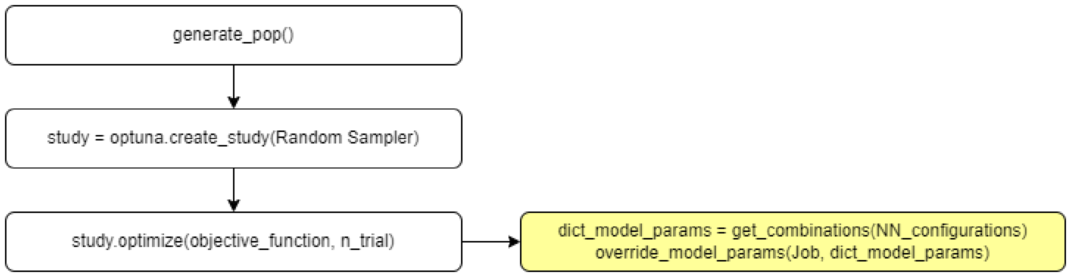

generate_pop()
study = optuna.create_study(Random Sampler)
study.optimize(objective_function, n_trial)
dict_model_params = get_combinations(NN_configurations)
override_model_params(Job, dict_model_params)

Figure 2.4: Combination - v1

Once the dictionary containing the variability of individual parameters within the configurations is obtained, the method **override_model_params** is called. This method replaces the parameter values (only if it appears in the dictionary) by randomly sampling a value from the available options.

With a single architecture available, this initial methodology allows for obtaining hybrid combinations of different fixed parameter sets, enabling a more extensive search for better weak learners.

Sampling Architecture by ID - V2

Having developed **Baselines Architecture** (explained in the next chapter), the second version of the optimization focused on using multiple heterogeneous architectures.

Just to introduce, **Baselines** are 14 architecture (10 for *regression* problems and 4 *classification*) created in order to obtain multiple uncorrelated (as much as possible, without losing good performance) predictions with different type of architecture (e.g *Random Forest Regressor*, *XGBRegressor*, *Lasso* and others).

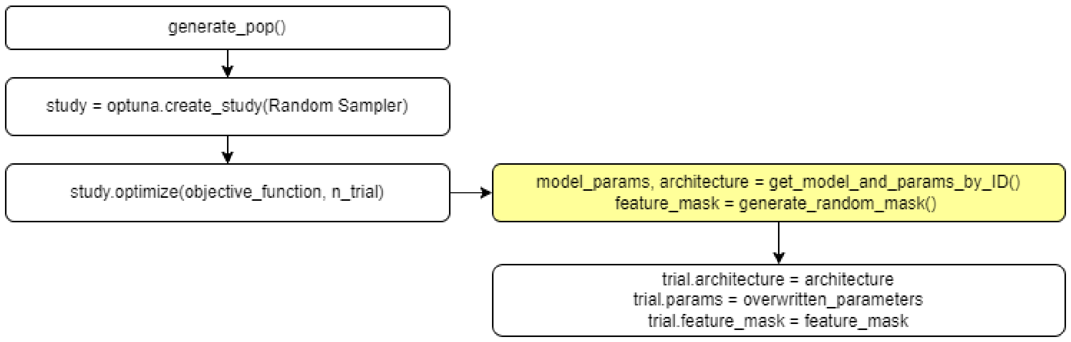

generate_pop()
study = optuna.create_study(Random Sampler)
study.optimize(objective_function, n_trial)
model_params, architecture = get_model_and_params_by_ID()
feature_mask = generate_random_mask()
trial.architecture = architecture
trial.params = overwritten_parameters
trial.feature_mask = feature_mask

Figure 2.5: Sampling by ID - v2

As shown, the *study* takes **model parameters** and **type of architecture** from *get_model_params_by_ID*: this method will sample random an id (associated to relative name and parameters of model) into a collections of *Baselines*; after that, will assign to the trial the components that will build the Job.

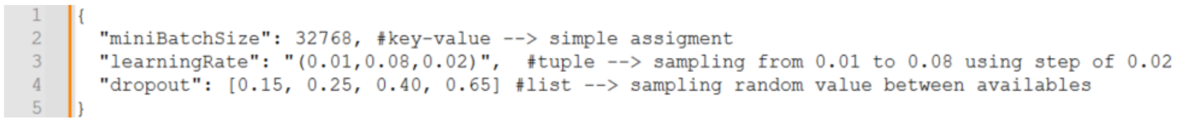
Override - V3

To take full advantage of Optuna capabilities, the focus was redirected to using different methods to introduce variety and multiple sources of optimization within the Baselines. Specifically, **different types of sampling** were developed to further modify the architectures and enhance exploration.

By incorporating these sampling methods, aimed to introduce approaches for exploring different architectures more comprehensively. On one hand, using various models encouraged exploration by diversifying the types of architectures considered. On the other hand, the ability to adjust individual parameters within each architecture enabled a little focus on exploitation, refining each model performance through targeted optimization managed by Optuna.

This feature uses a specific field in the Pool (*JSON field*) to specify parameters that should be overwritten in all architectures (e.g. the desired target type). Three types of sampling were developed based on how these parameters are written in the JSON field:

1. **List**: sampling a random value from the available options in the list.
2. **Tuple**: by specifying a start, end, and step, a value will be chosen between the start and end, skipping by the given step.
3. **Key-Value pair**: simply assigment to a specific key.



```
{
  "miniBatchSize": 32768, #key-value --> simple assigment
  "learningRate": "(0.01,0.08,0.02)",  #tuple --> sampling from 0.01 to 0.08 using step of 0.02
  "dropout": [0.15, 0.25, 0.40, 0.65] #list --> sampling random value between availables
}
```

Figure 2.6: Types of sampling - v3

As shown in 2.6, any architecture that contains one or more of these parameters will be overridden using the specific sampling method.

Ultimate Version

Since the strategy focuses on using heterogeneous architectures, V1 was removed because it was too complex to manage with multiple architectures; however, for the final version, we decided to create a single feature that utilizes both versions (V2 and V3), allowing for extensive variability in parameter and architecture selection during exploration and completion of the weak learners pool.

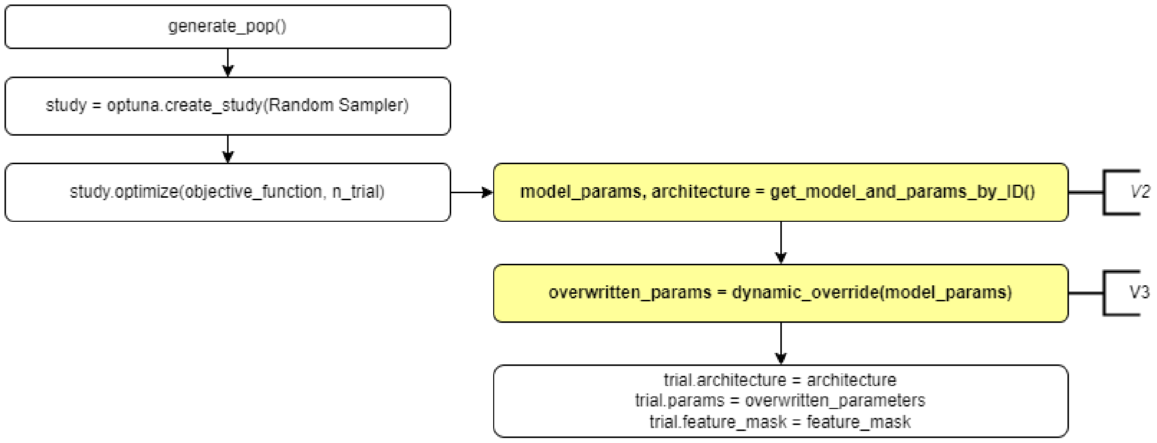

generate_pop()
study = optuna.create_study(Random Sampler)
study.optimize(objective_function, n_trial)
model_params, architecture = get_model_and_params_by_ID()
V2
overwritten_params = dynamic_override(model_params)
V3
trial.architecture = architecture
trial.params = overwritten_parameters
trial.feature_mask = feature_mask

Figure 2.7: Ultimate version

2.2.2 Routine Types

As mentioned before, the common procedure for generate each features mask is a random generation of binary mask with equal probability to obtain both values. This procedure does not account for which features are enabled or disabled, even if each of them are more predictive than others.

In order to consider all possible features mask, it was necessary to develop a kind of procedure to generate a specific number of Jobs associated to particular mask which would never be cosidered with random generation.

So, the process of mask generation enables three types of routine:

1. **Random**: standard procedure used

2. **Full Mask**: generate a features mask with all features enabled

3. **No context**: generate a features mask that enables only the features considered out of *Instrument Related* (also called *Context Feature*). These features are considered into the Context or out of Context depending on *Investable Universe* (if a specific feature are into the investable universe or not)

Therefore, before starting the algorithm, it is possible to specify whether to use a particular type of routine before the standard procedure or not (the default is to use the random routine).

Both the *full mask* routine and the *no context* routine will initially generate a Job with the desired routine for each architecture available in the models pool. This ensures that each model is included with the specific feature mask in the weak learners pool obtained at the end of the algorithm.

2.2.3 Flow Chart Optimized Random Search

Shown here is the diagram of the refactored and optimized version of the Random Search. The final goal remains to obtain the required number of Jobs specified into *Pool* attributes. The code has been set up as a script that, by taking **Pool ID** and **type of routine**, completes the generation of all the weak learners using the preavious *Refactor Components*.

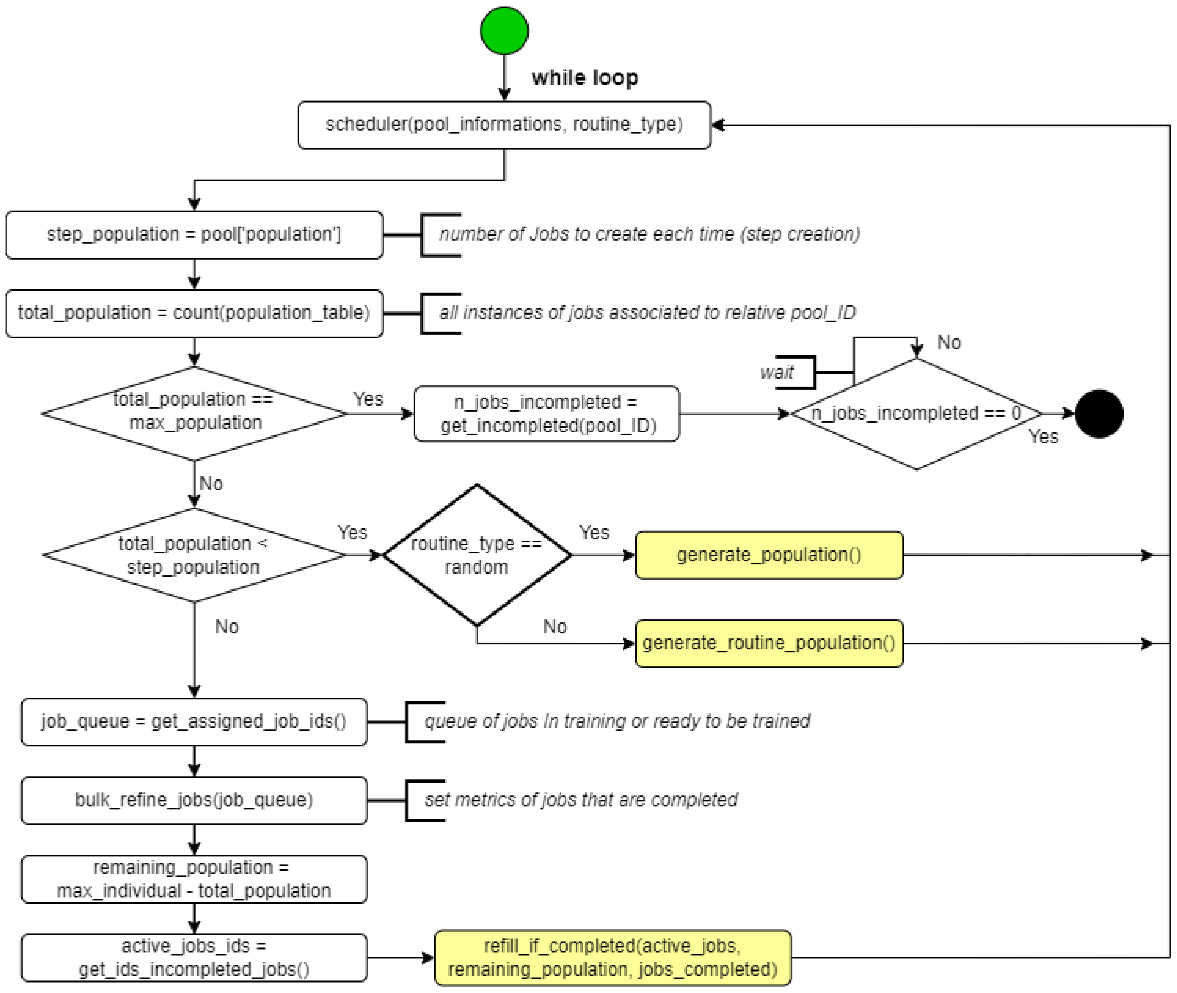

while loop
scheduler(pool_informations, routine_type)
step_population = pool['population']
number of Jobs to create each time (step creation)
total_population = count(population_table)
all instances of jobs associated to relative pool_ID
total_population == max_population
Yes
n_jobs_incompleted = get_incompleted(pool_ID)
wait
No
n_jobs_incompleted == 0
Yes
No
total_population < step_population
Yes
routine_type == random
Yes
generate_population()
No
generate_routine_population()
No
job_queue = get_assigned_job_ids()
queue of jobs In training or ready to be trained
bulk_refine_jobs(job_queue)
set metrics of jobs that are completed
remaining_population = max_individual - total_population
active_jobs_ids = get_ids_incompleted_jobs()
refill_if_completed(active_jobs, remaining_population, jobs_completed)

Figure 2.8: Flow Chart Optimized RS

The script terminates once all the required jobs are trained and the respective metrics are saved; as shown, the population management occurs in steps: in the first iteration, a number of instances equal to the **step size** is generated. Subsequently, as the jobs are completed, there is a gradual jobs generation mechanism **(progressive decay)** until completion (as shown later).

Making Population

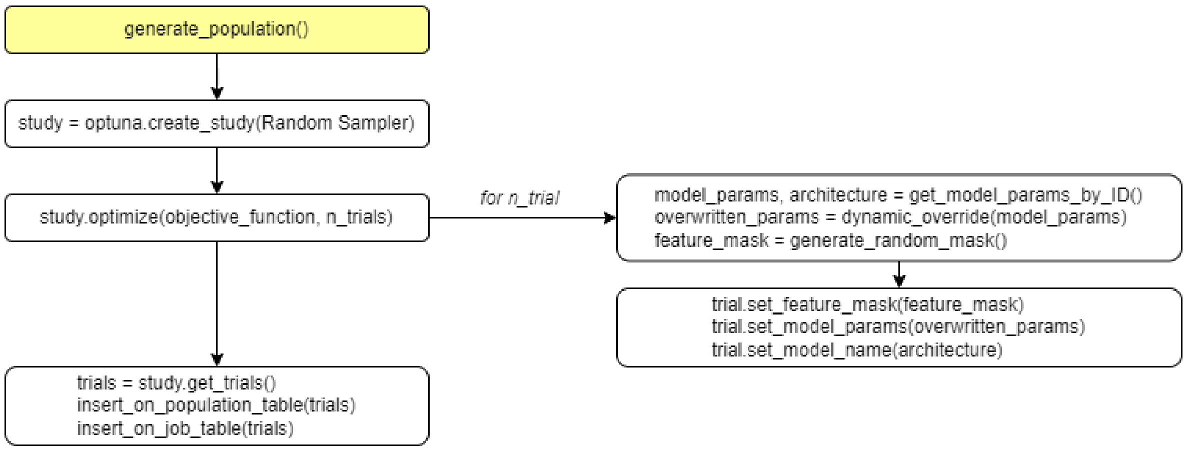

generate_population()
study = optuna.create_study(Random Sampler)
study.optimize(objective_function, n_trials)
for n_trial
model_params, architecture = get_model_params_by_ID()
overwritten_params = dynamic_override(model_params)
feature_mask = generate_random_mask()
trial.set_feature_mask(feature_mask)
trial.set_model_params(overwritten_params)
trial.set_model_name(architecture)
trials = study.get_trials()
insert_on_population_table(trials)
insert_on_job_table(trials)

Figure 2.9: Ultimate Generate Population

Different from previous version, after Optuna generates all the trials, they are spawned into both tables so that they are ready to be trained.

Making Population using Routine

The only difference in population generation using **routines** is that, instead of using sampling by ID to retrieve the model parameters and architecture, a trial is created for each baseline element; then, a mask is created satisfying the required routine type. Once this is done, normal operation resumes, returning to the use of the *random routine*.

Refill: Progressive Decay

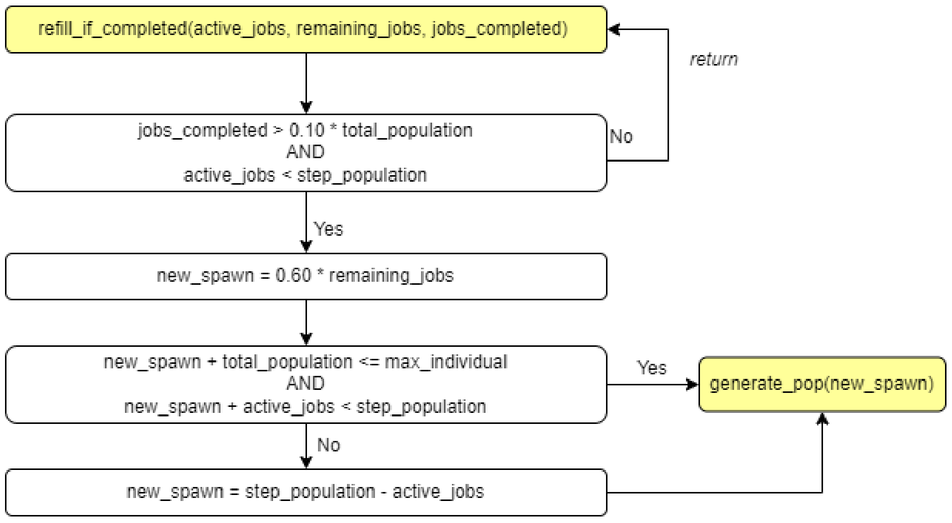

refill_if_completed(active_jobs, remaining_jobs, jobs_completed)
return
jobs_completed > 0.10 * total_population AND active_jobs < step_population
No
Yes
new_spawn = 0.60 * remaining_jobs
new_spawn + total_population <= max_individual AND new_spawn + active_jobs < step_population
Yes
generate_pop(new_spawn)
No
new_spawn = step_population - active_jobs

Figure 2.10: Progressive Decay

This methods wait until the 10% of total Jobs are completed and at this point, compute the number of Jobs for refill the *step population* (60% of remaining Jobs) through the **generate_pop** methods.

With this method, the number of Jobs created will progressively reduce to 1 until the exact maximum number of individuals is reached.

3. Baselines Models

As previously explained, the old algorithm initially required the use of a single architecture for exploration. Since the goal was to obtain a heterogeneous set of architectures, a study was conducted in parallel with the development of the new algorithm to create selected architectures for producing a repository of templates to be used with the new algorithm. The focus during this study was to identify, for each chosen architecture, two or three configurations (selected from a set of configurations for the same architecture) that ensured the least correlation between predictions and demonstrated good performance. This chapter will detail the development and selection process of these configurations, the **Baselines**.

3.1 Architectures and Methodologies

For the creation of *Baselines*, the following architecture were picked to investigate in order to obtain the final configurations for each oh them:

- **XGB Regressor**
- **Neural Network**
- **Random Forest**
- **Lasso**
- **Gradient Boosting Regressor**
- **Logistic Regression**
- **Linear Regression**

Indipendently of the problem at hand is *classification* or *regression*, the context for creating the configurations is Learning-to-Rank (LTR).

In detail, each dataset contains multiple **financial assets** in a defined time-series. After the training phase, in order to evaluate the configurations, models will produce a final ranking (owned by Axyon). Each asset will be evaluated in a time horizon defined by the target related to the specific problem; the ranking of the best-performing assets within that horizon is determined based on return of the assets over horizon.

Axyon Platform

Through Axyon proprietary platform, it is possible to train models of various types (e.g. *sklearn*, *xgboost*) using existing wrappers that handle:

1. **General Settings**: type of Job (e.g classification, regression) and subtype (e.g single or multiple label), cluster options and others.
2. **Training Parameters**: data manipulator and splitting.
3. **Model and Dataset**: model type, model parameters, feature mask (always **full** during the configuration tests) and dataset information.

By setting all of these, it is possible to schedule a list of jobs that are pending training.

Generate Configurations

Using the candidate architectures for both of problems, in order to find the final configurations of the Baselines I applied the following approach:

1. For all the initial test, the dataset used is **European Equity**
2. Create for each architecture a **group**, where all tests conducted with that specific model were included.
3. Starting from the most standard architecture possible, I tried varying the parameters from job to job, aiming to find a value for each customizable parameter within the model that provided acceptable performance, while gradually changing the others and keeping those already analyzed fixed.

4. Once obtaining the first configurations, the focus was to create uncorrelated and also good performance therefore, I started to create and train multiple opposing configuration (e.g more or less deep, using regularization terms or not, using high and low values of dropout, increasing or decreasing learning rate and so on), depending on model and parameters.

5. At this point, for each group of architectures, I achieved a number of configurations that allowed me to analyze the trade-off between correlation and performance (removing the configuration with poor performance).

3.1.1 Analysis

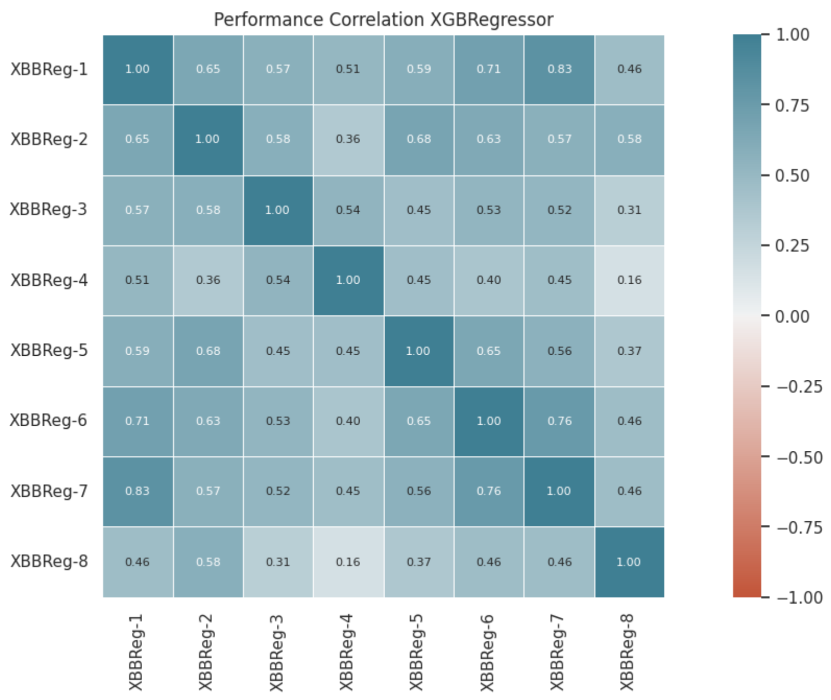
In this section will be illustrate all the analysis for each type of architecture through which has been obtained the final configurations of *Baselines*. Within the paragraph dedicated to each architecture, correlation matrices on the performance of some configurations **(out of the many tested, keeping those that performed best)** will also displayed.

XGB Regressor

From *XGBoost Library*, this architecture implements a ML algorithm of *Gradient Boosting Framework*, using parallel tree boosting. Below, the main parameters considered are shown, along with their default values and the values used during the search for candidate architecture(s).

Parameter	Default Value	Adopted Values
objective	reg:squarederror	reg:squarederror, reg:absoluteerror
eval metric	dep. on objective	mae,rmse
max depth	6	3,5,6,8,12,14
learning rate	0.1	0.0009,0.08,0.03,0.1
earlystoppingrounds	/	10,18,30,50,100
subsample	1	0.5,0.8,1
colsample bytree	1	0.5,0.8,1
min child weight	1	1,4,20,35,50
random state	/	none, random
alpha	0	0.005,0,0.3

In contrast to the other architectures, this one still requires further investigation as the weak learners assigned to it tend to demonstrate overfitting during actual use.



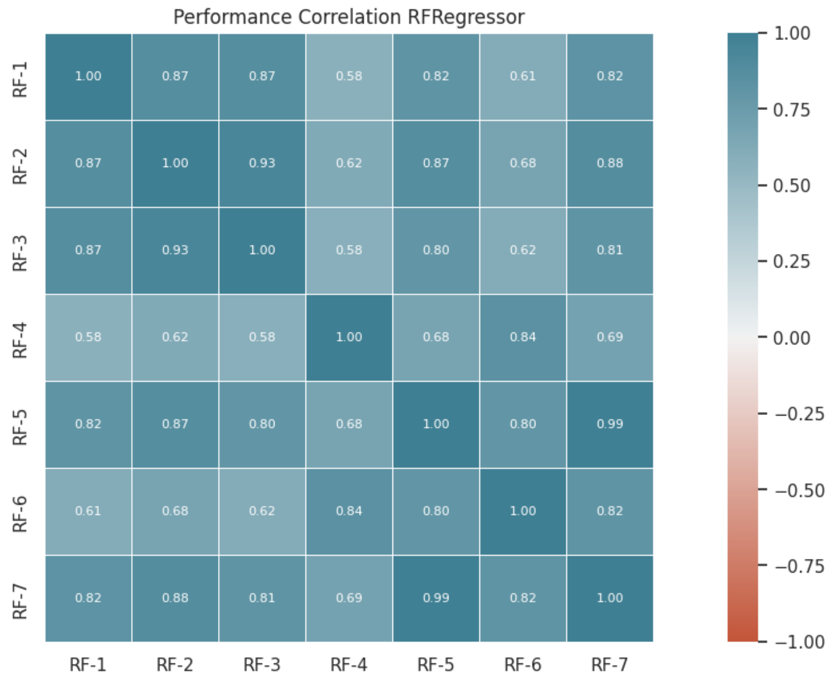
	XBBReg-1	XBBReg-2	XBBReg-3	XBBReg-4	XBBReg-5	XBBReg-6	XBBReg-7	XBBReg-8
XBBReg-1	1.00	0.65	0.57	0.51	0.59	0.71	0.83	0.46
XBBReg-2	0.65	1.00	0.58	0.36	0.68	0.63	0.57	0.58
XBBReg-3	0.57	0.58	1.00	0.54	0.45	0.53	0.52	0.31
XBBReg-4	0.51	0.36	0.54	1.00	0.45	0.40	0.45	0.16
XBBReg-5	0.59	0.68	0.45	0.45	1.00	0.65	0.56	0.37
XBBReg-6	0.71	0.63	0.53	0.40	0.65	1.00	0.76	0.46
XBBReg-7	0.83	0.57	0.52	0.45	0.56	0.76	1.00	0.46
XBBReg-8	0.46	0.58	0.31	0.16	0.37	0.46	0.46	1.00

Figure 3.1: XGB Regressor Performance Correlation

Random Forest Regressor

From *Scikit-Learn Library*, Random Forest Regressor fits a number of *decision-trees* on various subsample of dataset.

Parameter	Default Value	Adopted Values
n estimators	100	10,30,100,150,200,300
criterion	squared error	squared error,absolute error
max depth	none	none,5,10,20
min sample split	2	none,0.2,0.6,2,5,30
min sample leaf	1	none,1,5,10,30
max features	1.0	none,log2,sqrt,0.8
random state	none	none,random



	RF-1	RF-2	RF-3	RF-4	RF-5	RF-6	RF-7
RF-1	1.00	0.87	0.87	0.58	0.82	0.61	0.82
RF-2	0.87	1.00	0.93	0.62	0.87	0.68	0.88
RF-3	0.87	0.93	1.00	0.58	0.80	0.62	0.81
RF-4	0.58	0.62	0.58	1.00	0.68	0.84	0.69
RF-5	0.82	0.87	0.80	0.68	1.00	0.80	0.99
RF-6	0.61	0.68	0.62	0.84	0.80	1.00	0.82
RF-7	0.82	0.88	0.81	0.69	0.99	0.82	1.00

Figure 3.2: Random Forest Regressor Performance Correlation

Lasso

From *Scikit-Learn Library*, Lasso is a linear model trained with L1 prior as regularizer.

Parameter	Default Value	Adopted Values
alpha	1.0	0.0001,0.001,0.004,1.0
max iter	1000	50,70,100,250,1000,1500
random state	none	none,42
selection	cyclic	cyclic,random
tol	1e-4	1e-4,1e-2

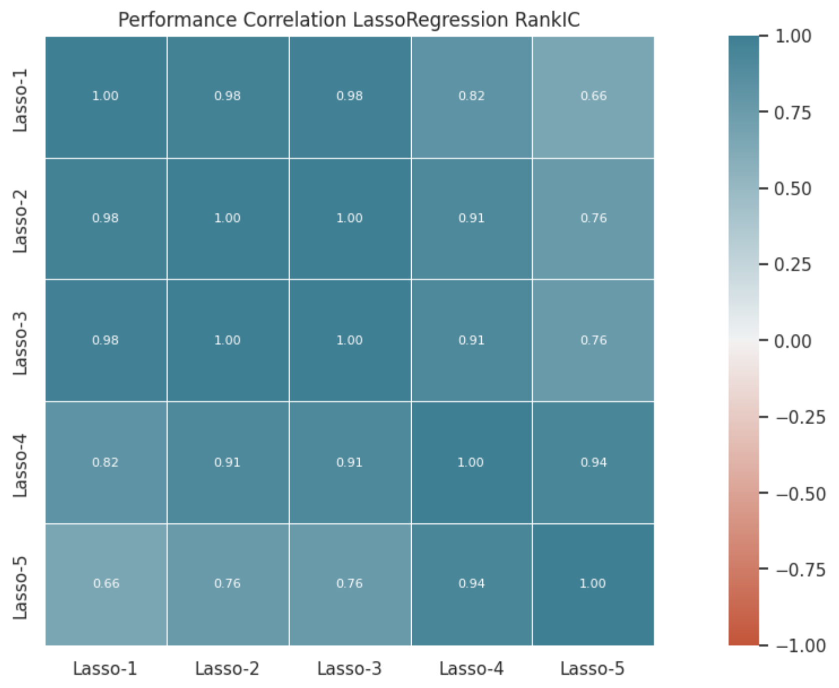


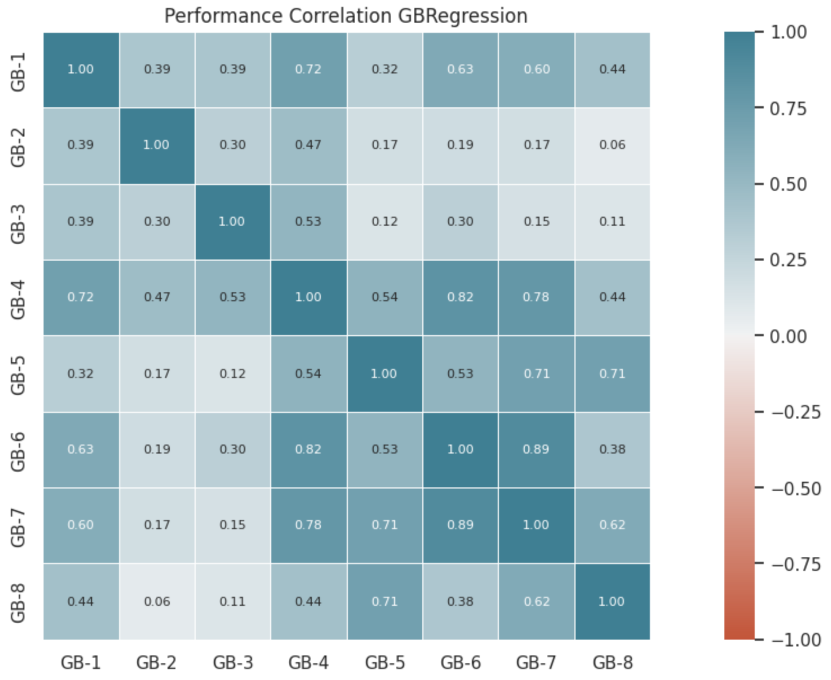
Figure 3.3: Lasso Performance Correlation

As evidenced in the correlation matrix, Lasso (and also other linear models) exhibit a very high correlation between different configurations. Compared to other architectures, they consistently maintained high performance during tests.

Gradient Boosting Regressor

As mentioned into *Scikit-Learn Library*, this architecture train in each stage regression tree on the negative gradient of specific loss function.

Parameter	Default Value	Adopted Values
learning rate	0.1	0.001,0.02,0.05,0.08,0.1
n estimators	100	10,50,100,200,500
subsample	1.0	none,0.6,0.7,1
max depth	3	2,3,4,5,10,15,25
validation fractions	0.1	0.1,0.3,0.5
loss	squared	squared,quantile
max leaf nodes	none	none,3,5,8,10,20



	GB-1	GB-2	GB-3	GB-4	GB-5	GB-6	GB-7	GB-8
GB-1	1.00	0.39	0.39	0.72	0.32	0.63	0.60	0.44
GB-2	0.39	1.00	0.30	0.47	0.17	0.19	0.17	0.06
GB-3	0.39	0.30	1.00	0.53	0.12	0.30	0.15	0.11
GB-4	0.72	0.47	0.53	1.00	0.54	0.82	0.78	0.44
GB-5	0.32	0.17	0.12	0.54	1.00	0.53	0.71	0.71
GB-6	0.63	0.19	0.30	0.82	0.53	1.00	0.89	0.38
GB-7	0.60	0.17	0.15	0.78	0.71	0.89	1.00	0.62
GB-8	0.44	0.06	0.11	0.44	0.71	0.38	0.62	1.00

Figure 3.4: Gradient Boosting Regressor Performance Correlation

Neural Network

For Neural Networks in classification (and also regression) problem, the Axyon's platform uses a proprietary model to encapsulate the parameters related to this architecture and train the configuration. To find the configurations for this architecture, the common parameters explored are:

Parameter	Adopted Values
layers	[8,8],[16,16,16],[64,64,64],[64,32],[128,64],[256]
dropout	none,0.25
hidden dropout	none,0.4,0.5
L1	0.0002,0.01,0.05
L2	0.0005,0.05,0.02,0.01
epochs	100,150,200,350

Neural Network - Regression

Parameter	Adopted Values
loss	mse
output:hidden activation	linear
output:output counts	1

Neural Network - Classification

Parameter	Adopted Values
loss	binary cross-entropy
output:hidden activation	sigmoid
output:output counts	4

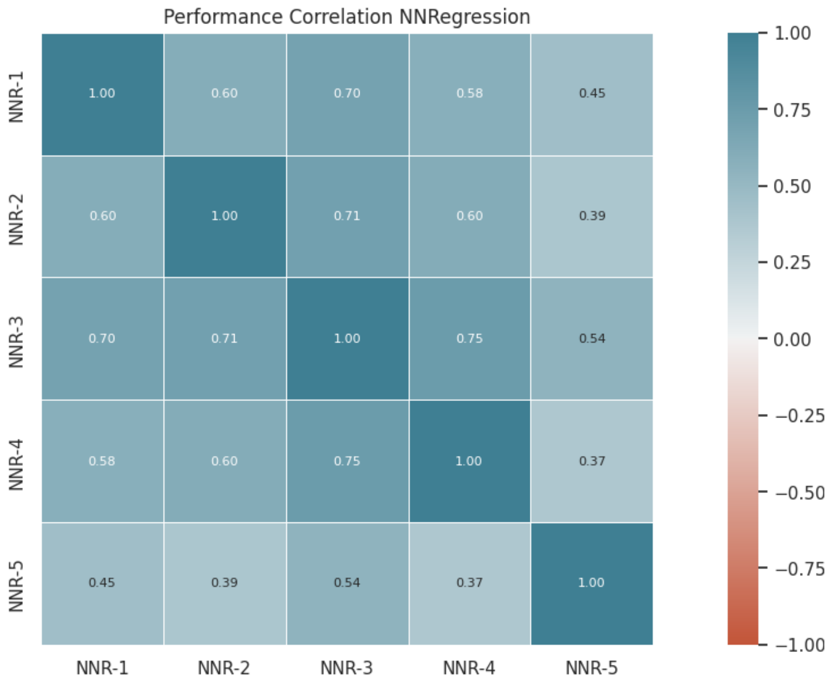


Figure 3.5: NN Regression Performance Correlation

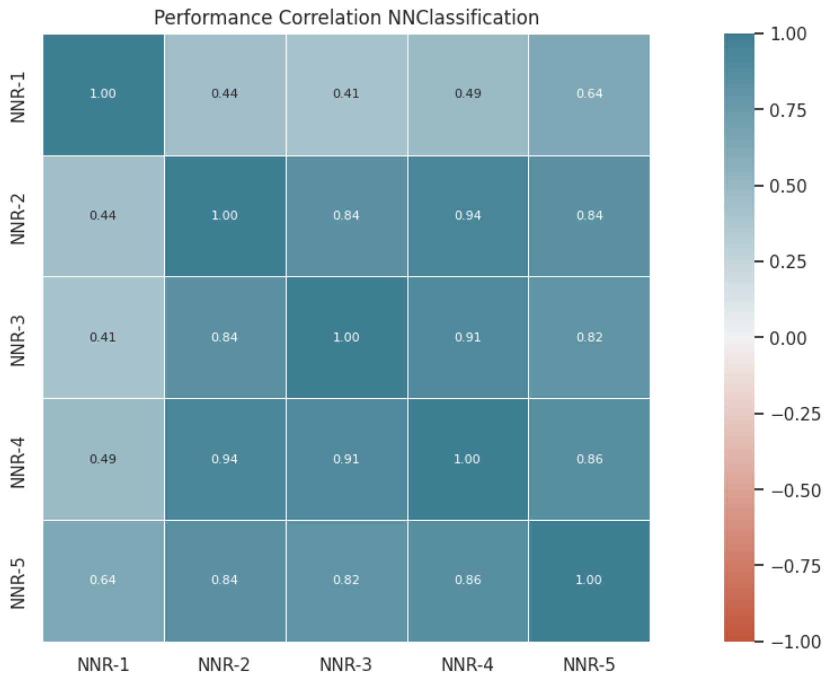


Figure 3.6: NN Classsification Performance Correlation

Linear Regression

After observing the behavior of linear models, the standard architecture of *Scikit-Learn* was used for *Linear Regression*.

Logistic Regression - Classsification

For this particular architecture, in a classification problem, the (previous) standard Neural Network architecture was used, adapted to become a *Logistic Regression* model: all hidden layers were removed, and a sigmoid activation function was applied at the output to maintain the four neurons required for classification.

3.1.2 Selected Configurations

The correlation results within each individual architecture group confirmed the previous expectation: it is necessary to use **heterogeneous ensembles** to achieve robust predictions. This is because combining different architectures helps to leverage their individual predictivity and mitigate their weaknesses, to improve overall performance and generalization on unseen pattern through the input data. For this reason, from the entire group of different configurations of all the architectures, we selected the architecture configurations that had a good balance of performance and low intra-architecture correlation.

Regression Architecture	Parameters
Lasso Reg.	alpha=0.001, max_iter=50, selection=random
Random Forest Reg. small	max_depth=5, max_features=log2, min_samples_leaf=30 min_sample_split=0.2, n_estimators=10
Random Forest Reg. deep	n_estimators=150, max_depth=10, min_sample_split=5 min_samples_leaf=10, max_features=sqrt
XGB Reg. medium	n_estimators=150, max_depth=5, lr=0.08, subsample=0.5 earlystoppingrounds=100, colsample_bytree=0.5 min_child_weight=35, random_state=random
XGB Reg. deep	n_estimators=100, max_depth=8, lr=0.1 earlystoppingrounds=30, subsample=0.5, lambda=1.3 random_state=random
NN Reg. small	layers[16,8], no dropout, no L1, no L2
NN Reg. medium	layers[64, 64], dropout=0.7
GB Reg. medium	n_estimators=100, lr=0.02, max_depth=6 subsample=0.7, max_leaf_nodes=8, max_features=log2 n_iter_no_change=8
Linear Reg.	standard model
Linear Avg.	mean of features

Classification Architecture	Parameters
Logistic Reg.	no dropout, outputs_counts=4, no L1, no L2
NN Cls. small	layers[8,8], dropout=0.5, no L1, no L2
NN Cls. medium no reg	layers[16,16,16], no dropout, no L1, no L2
NN Cls. medium w reg	layers[16,16,16], dropout=0.3, L1=0.002, L2=0.002

3.1.3 Performance Correlation of Baselines

The construction and subsequent selection of these heterogeneous configurations were developed in parallel with the refactoring and optimization of the Random Search. It was necessary to modify the exploration algorithm to enable the execution of Random Search with models of different natures and various types of feature masks, in order to obtain the desired pool of weak learners. Below, the correlation between all Baselines will be shown, measured on the same dataset used for testing and production.

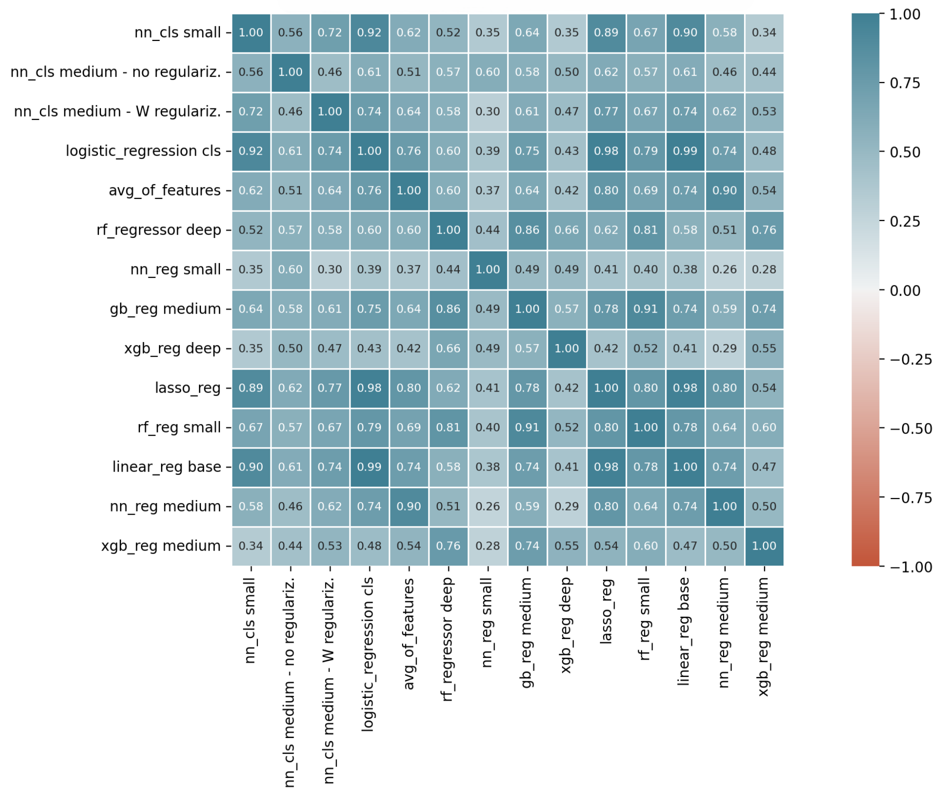


Figure 3.7: Performance Correlation between Baselines

4. Selection on Ensemble

After completing the Random Search, a set of weak learners consisting of hundreds of jobs is obtained (depending on how the exploration pool is configured). These will serve as the building blocks to select a smaller number of jobs that will establish the final ensemble, which will then be used to generate predictions and produce customers strategies. Currently, there is a well-defined process for selecting the weak learners within the pool:

1. **Aggregate Predictions**: using ad-hoc script, from a specific *pool ID*, an **.hdf** file is generated that will contain the aggregated predictions (each contained into *Pandas DataFrame*) of the individual weak learners. This type of file is a kind of big dictionary of dataframes, each corresponding to a specific job accessible using relative *Job ID* (key).

2. **Select Ensemble on predictions**: starting from **.hdf** file generated in the previous step, a **Greedy selection algorithm** (explained later) is executed, with the goal of selecting the weak learners where maximized a desired metric in a specific data split (e.g *cv*).

3. At the end of this process, the *Greedy Algorithm* produce a JSON file containing all the ids of the selected weak learners, ready to produce signals for strategies.

4.1 Limitations

This process is crucial for generating strategies, as the extraction of the most predictive weak learners depends on its effectiveness. The main limitation of this process lies within the *selection algorithm* itself: being a greedy algorithm, it repeatedly iterates over all available jobs, calculating the desired metric and selecting the candidate that maximize metric when combined with those previously selected. This procedure is very slow and **time consuming** (e.g. under certain conditions, it requires dozens of hours), primarily due to the huge size of the HDF file, the number of weak learners to be selected and the metrics computation (also their implications).

4.2 Greedy Algorithm

Algorithm 1 Greedy Selection on Ensemble

```
Require: specify max_ensemble_size
  $chosen\_ids = list()$
  $ensemble\_test\_metrics = list()$
  for i in max_ensemble_size do
      $available\_candidates = get\_ids\_from\_hdf()$
      if $len(available\_candidates) == 0$ then
          $break$
      else
          $best\_metric\_cv = -np.inf$
          $metric\_test = -np.inf$
          for candidate_id in available_candidates do
              $ensemble\_dataframe = build\_df\_from\_hdf()$
              $candidate\_metrics\_cv = evaluate\_ensemble(ensemble\_dataframe)$
              $candidate\_metrics\_test = evaluate\_ensemble(ensemble\_dataframe)$
              if $candidate\_metrics\_cv > best\_metric\_cv$ then
                  $best\_id = candidate\_id$
                  $best\_metric\_cv = candidate\_metrics\_cv$
                  $metric\_test = candidate\_metrics\_test$

          $chosen\_id.append(best\_id)$
          $ensemble\_test\_metrics.append(metric\_test)$
```

Main steps of the algorithm:

1. Required *max ensemble size*: number of maximum weak learner obtained at the end of selection.
2. Each time, obtain the available candidates from *HDF* file (list of ids).
3. If there are available candidates, for each of those: compute metrics on **cv** and **test** data split and check if metrics just obtained are better than the current best.
4. At the end of candidates evaluation, save the *candidate best id* and update metrics on test.

In particular, this algorithm optimize on **cv** and select on **test**. During the execution, the chosen ids are the weak learner that represent the ensemble; for this reason, each candidate will be added and evaluated (on metrics calculation) into the ensemble (so aggregated with other chosen members) and will be chose only if the new ensemble metrics is higher than the previous.

To obtain the ensemble (aggregated) data used during the evaluation, the algorithm retrieve and concatenate from HDF file each **dataframe** associated to an **id** in the **chosen_ids** list.

4.2.1 Key objective

Due to limitations, i focused on trying to speeding up the current execution time of the algorithm by analyzing the main bottlenecks through **code profiling** and attempting to implement potential solutions that could improve execution and computation; since this process run into dedicated cluster, performance analisys were made locally with a restricted HDF file (small number of weak learner) and low *max_ensemble_size* parameter, consequence of limited hardware resources.

4.2.2 Performance Analysis on time execution

In order to find bottleneck, a first analisys were made using *SnakeViz* tool, on five Jobs HDF using *max_ensemble_size* of four. As shown below, during the entire execution of *greedy_selection*, the *build_df_from_hdf()* function takes around of 60% of the execution time. This is due to the fact that, for each candidate in the list of available ones, the respective dataframe (from the HDF) containing the predictions of the weak learner is repeatedly read, the metrics are calculated with the ensemble members, and if the candidate is not included in the *chosen_ids*, it will continuously be read and evaluated multiple times (during the achievement of max ensemble size). Assuming there are 100 jobs and a max ensemble size of 25 jobs, for each non-selected candidate, 25 reads of the same dataframe from the HDF will be performed (at least 75 jobs each 25 times will be read).

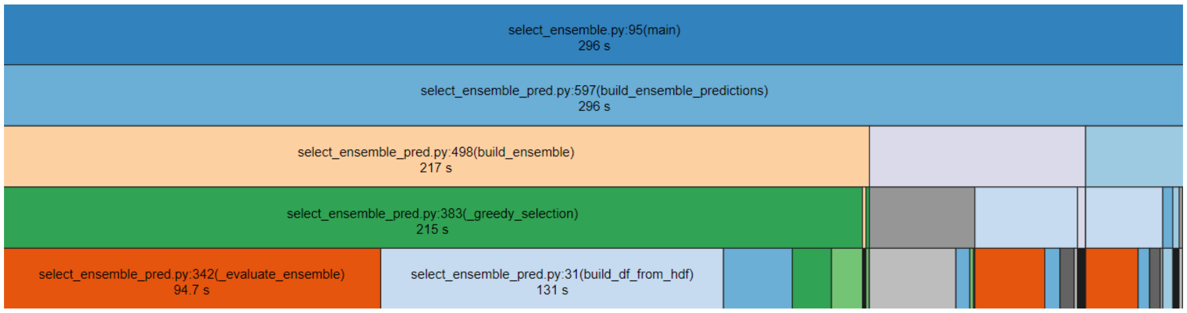


Figure 4.1: Profiling greedy selection on five job

The *evaluate_ensemble* function cover the remaining time of execution, due to metrics calculation with specific methods (already optimized).

4.2.3 Performance Analysis on resources

Metrics function use aggregate data obtained from the *ensemble_dataframe*: this dataframe is a concatenation of single dataframe (retrieved from HDF) related to *chosen_ids* with *candidate_id*. For this reason the process of metrics evaluations is slow and during the execution gets worst when ensemble size increase: this behaviour is a consequence of the fact that there are repeated informations into columns that lead to increasing memory (RAM) occupation; moreover, on this huge dataframe will be performed some *group_by* operation (where the execution time is strictly related to dataframe size) in order to obtain a single dataframe for evaluation.

4.3 Optimization Proposal

Identified the main bottleneck in the algorithm's execution, the focus was on finding a solution to speed up the algorithm without modifying the selection process, specifically the greedy selection, to avoid a radical change in the selection methodology and consequently the associated metrics. Since each Job (i.e *weak learner*) is always read at least once, the main idea is to create a pre-build dataframe (**global dataframe**) containing the relevant useful information for each job: in this manner, all data used for ensemble evaluation are preloaded and ready to use in each iteration, instead of repeated reading.

Global Dataframe from scratch

The procedure to create global dataframe is:

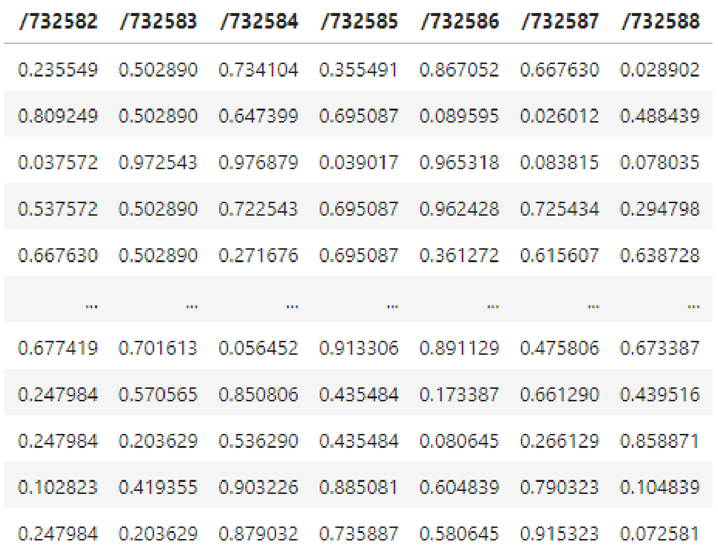
1. Create an initial *global dataframe* of fixed information: all the information shared across all jobs (e.g date, asset ids, split period and ground truth labels)

2. Since the **predictions vector** from each weak learners will be primarily useful for evaluating the ensemble, the idea is to retrieve this data and group it as needed for metric calculations.

3. For each job, read the related dataframe into HDF file, getting the predictions

vector of the weak learner, compute required groupby operations and merge this data within base dataframe previous build.

4. In this manner, only the prediction vector will be added to global dataframe **as a column** named as relative Job ID (without repeating shared information).



/732582	/732583	/732584	/732585	/732586	/732587	/732588
0.235549	0.502890	0.734104	0.355491	0.867052	0.667630	0.028902
0.809249	0.502890	0.647399	0.695087	0.089595	0.026012	0.488439
0.037572	0.972543	0.976879	0.039017	0.965318	0.083815	0.078035
0.537572	0.502890	0.722543	0.695087	0.962428	0.725434	0.294798
0.667630	0.502890	0.271676	0.695087	0.361272	0.615607	0.638728
...	...	...	...	...	...	...
0.677419	0.701613	0.056452	0.913306	0.891129	0.475806	0.673387
0.247984	0.570565	0.850806	0.435484	0.173387	0.661290	0.439516
0.247984	0.203629	0.536290	0.435484	0.080645	0.266129	0.858871
0.102823	0.419355	0.903226	0.885081	0.604839	0.790323	0.104839
0.247984	0.203629	0.879032	0.735887	0.580645	0.915323	0.072581

Figure 4.2: Jobs Predictions Vector on Greedy optimization

Once all the prediction vectors have been extracted and merged with the common columns for all jobs, the global dataframe is split into cross-validation and test sets according to the data split; this makes the dataframe used for metric calculations on a specific subsample of the data even more manageable.

Prepare Data for Metrics Computations

When the *global dataframe* has been created, the evaluate ensemble procedure start: in the first iteration, all the available candidates will be individually evaluated in order to find the candidate which provide the best metric on cv data split (first member of ensemble). Instead of using the previous candidate dataframe and concatenating each dataframe for next comparison, the chosen IDs will be saved: in the following iterations the data will be retrieved from the global dataframe using only the corresponding column name, which is the Job ID.

Since data are already grouped, ensemble ids (chosen) and relative prediction vector will be averaged on row axis and next evaluated with the candidate prediction vector.

4.3.1 Memory Consumption Comparison

The old dataframe management (using concatenation) involved to an increase in resource usage as the algorithm increased the ensemble size: this was because, once a candidate was selected and added to the ensemble, the dataframe for metric calculation permanently contained its data, which was then concatenated with the data of each candidate during comparison. As the ensemble size grew (up to the maximum ensemble size), this procedure allowed for the concatenation of dozens of dataframes.Performing local tests, with an ensemble size of five jobs, the RAM consumption was approximately 3.5 GB for the dataframe (for an ensemble size of 20, it would become approximately 10 GB); which is why, locally, with an HDF file consisting of more than ten jobs, the execution runs out of memory.

By utilizing the *global dataframe* containing all pre-loaded data, the memory savings were significant even locally, allowing tests to be executed with up to 25 jobs (compared to 10 locally) and avoiding issues related to max ensemble size (since all dataframes have already been read and processed to obtain the prediction vectors used). The global dataframe alone, build during the first phase of execution using an HDF of 100 jobs, weighs only 5 GB. This produce considerable savings on cluster side, where the selection process is usually executed.

4.3.2 Execution Time Benchmarking

In this section will be shown tests performed locally to conduct a comparative analysis between the previous version of the code and the optimized version according to my proposal, by varying the number of jobs (and therefore the size of the HDF file) but keeping the same *max ensemble size* parameter (to maintain the same number of global iterations, so increasing the number of jobs will only increase the number of candidates and comparisons between them).

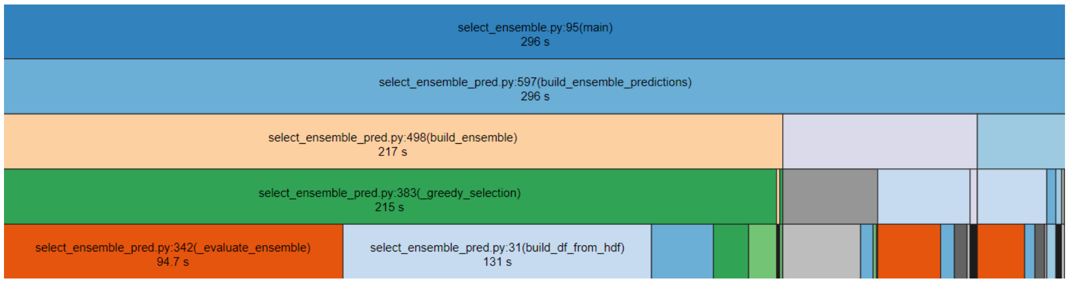


Figure 4.3: OLD - Execution time on five Jobs

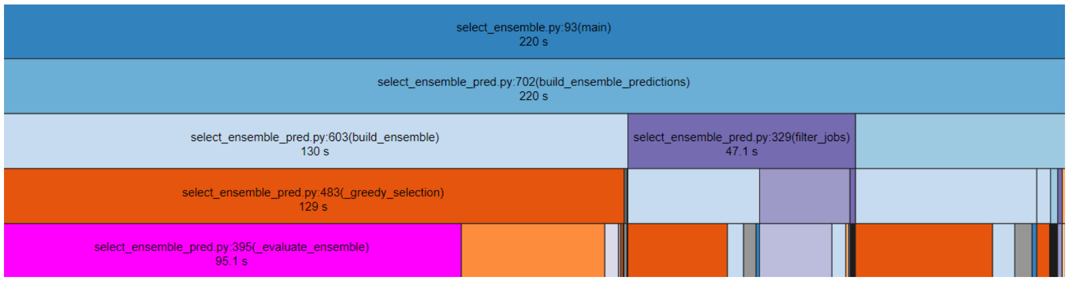


Figure 4.4: NEW - Execution time on five Jobs

Point of analysis (5 Jobs)	Old Version	New Version	Time Savings
Entire execution	296s	220s	25%
Greedy selection	215s	129s	40%
Build df from hdf	131s	34s	74%

As shown in the table on 5 jobs execution (old vs new) provide an overall performance improvement of 25% (considering the execution time of built-in functions and the time required to compute metrics in *evaluate_ensemble*, which cannot be optimized). Regarding the performance of the greedy algorithm alone, a time saving of 40% was achieved with the new dataframe management. Additionally, there is a significant time saving in reading from HDF (previously done with *build_df_from_hdf()*), which now takes 34 seconds to build the *global dataframe*.

4.3.3 Scaling on Jobs number

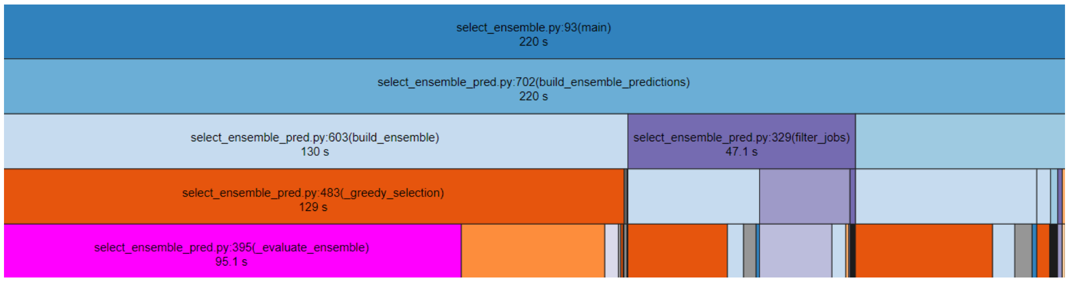


Figure 4.5: Execution time on five Jobs

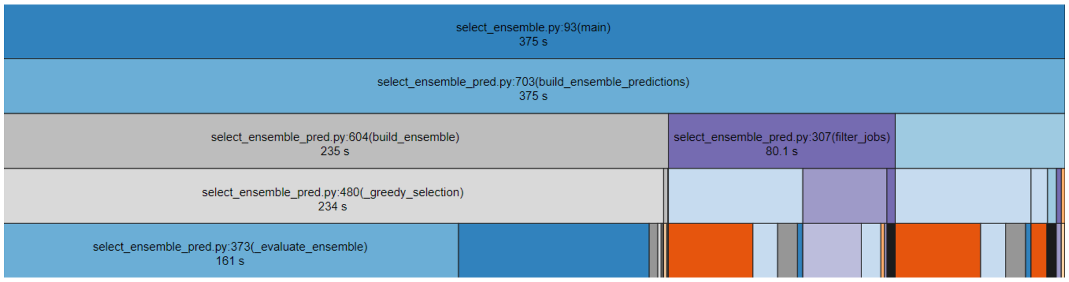


Figure 4.6: Execution time on ten Jobs

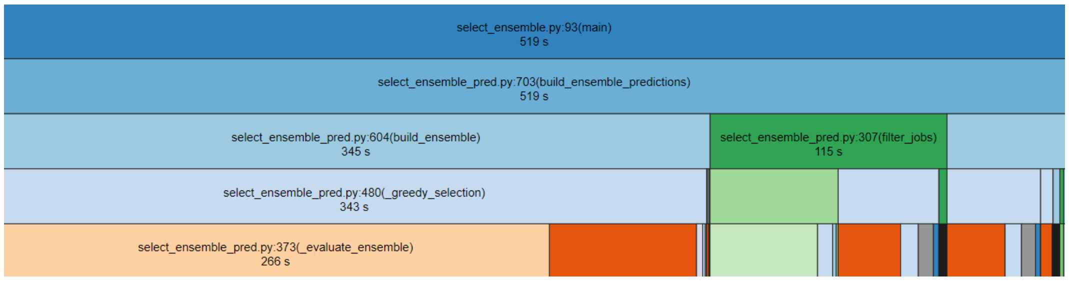
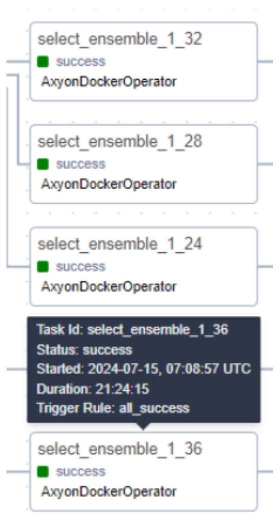


Figure 4.7: Execution time on twenty Jobs

Without the functions on which optimization cannot have an effect (e.g. evaluate ensemble used for metric calculation and filter jobs for removing any incompleted jobs), the greedy algorithm shows an increase of 105 seconds from 5 to 10 jobs and 109 seconds from 10 to 20 jobs. This data indicates a trend that appears to be linear with jobs increase, contrary the previous version which seemed to go towards a quadratic trend (also considering the high number of execution hours) due to the high and incremental use of computational resources required, which inevitably cause a slowdown in performance.

4.3.4 Global Performance Running on cluster

At the end of development and review of Greedy Optimization, an optimized select ensemble was executed on Axyon cluster using the same parameters as the last selection. Below, the total duration of the two select ensembles (optimized and non-optimized) running on high-performance hardware.



select_ensemble_1_32
success
AxyonDockerOperator

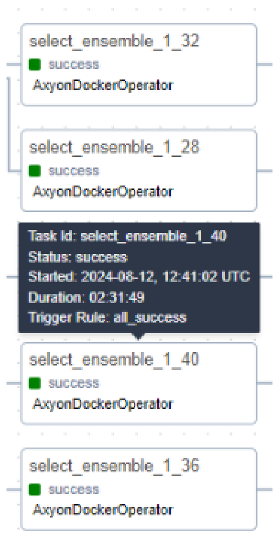
select_ensemble_1_28
success
AxyonDockerOperator

select_ensemble_1_24
success
AxyonDockerOperator

Task Id: select_ensemble_1_36
Status: success
Started: 2024-07-15, 07:08:57 UTC
Duration: 21:24:15
Trigger Rule: all_success

select_ensemble_1_36
success
AxyonDockerOperator

Figure 4.8: Old Select Ensemble



select_ensemble_1_32
success
AxyonDockerOperator

select_ensemble_1_28
success
AxyonDockerOperator

Task Id: select_ensemble_1_40
Status: success
Started: 2024-08-12, 12:41:02 UTC
Duration: 02:31:49
Trigger Rule: all_success

select_ensemble_1_40
success
AxyonDockerOperator

select_ensemble_1_36
success
AxyonDockerOperator

Figure 4.9: Optimized Select Ensemble

These tests were both executed with the same HDF file of *100 Jobs*, the same *max ensemble size* parameter and the same dataset on which the individual weak learners were trained. From the overall duration of both versions, it is evident that the optimization led to an approximately **8x** improvement in performance with related benefits on memory consumption.

5. Diversity

Diversity problem is a ”holy grail” in Ensemble Learning since many years. Another certainty in ensembles is that works better with learners that have ”diversity” in predictions; building on base concept behind Ensemble Learning, it guarantees an ’average out’ of individuals errors. This chapter will detail studies and research to achieve and obtain a specific measure of *diversity* within our heterogeneous ensemble. Although it remains an open problem, the primary difficulty encountered in the literature lies in adapting certain proposed methodologies to the context of Learning to Rank used into Axyon’s process.

During the literature review phase, the key reference became the paper [5], which compiles all the studies conducted over the past 25 years on the topic of diversity.
On this side, literature is divided into two macro-categories:

1. **maximization of diversity** as an optimization problem.

2. **quantify and manage diversity** as a measure of models fit.

5.0.1 Preface

We are looking for a definition of diversity as a measure of disagreement between ensemble weak learners. Using the paper [5] as a reference, it is evident how, over the years, the concept of ”diversity” has evolved into a various idea linked to a totally open problem. Supported by the theoretical studies of previous years, the authors followed the approach of revealing that diversity is a hidden dimension within the bias-variance decomposition and can be quantified as a measurable aspect of *model training*, thus it depends to the **loss**.

5.1 Bias-Variance-Diversity Decomposition

In the [5] approach, authors use the same methodology for many different losses in classification and regression scenarios.
Given an ensemble of $\{q_i\}_{i=1}^m$, combined by centroid combiner:

$$\overline{q} = argmin_{z \in Y}[\frac{1}{m}\sum_{i=1}^{m} l(\mathbf{z}, q_i)]$$

using any loss l, the expected risk of the ensemble:

$$\mathbb{E}_{\mathbb{D}}[R(\overline{q})] = \underbrace{R(y^*)}_{noise} + \underbrace{\frac{1}{m}\sum_{i=1}^{m}[R(\dot{q}_i) - R(y^*)]}_{bias} + \underbrace{\frac{1}{m}\sum_{i=1}^{m}\mathbb{E}_{\mathbb{D}}[R(q_i) - R(\dot{q}_i)]}_{variance} - \underbrace{\mathbb{E}_{\mathbb{D}}[\frac{1}{m}\sum_{i=1}^{m}[R(q_i) - R(\overline{q})]]}_{diversity}$$

we have four terms: noiose and effects of *bias, variance* and *diversity.* Thus, as diversity effect increases, it reduces expected risk but may potentially take negative values because it is simply the difference between average risk and ensemble risk, so when ensemble performs worse than the average, this will be negative; all of these terms can be estimated from data.

5.1.1 Squared Loss

Considering $l(y,q) = (y-q)^2$, which implies:

- $\overline{q} = \frac{1}{m}\sum_{i=1}^{m} q_i$
- $\dot{q} = \mathbb{E}_{\mathbb{D}}[q]$

The decomposition at each test point *(x, y)* is:

$$\mathbb{E}_{\mathbb{D}}[(\overline{q} - y)^2] = \underbrace{\frac{1}{m}\sum_{i=1}^{m}[(\dot{q}_i - y)^2]}_{bias} + \underbrace{\frac{1}{m}\sum_{i=1}^{m}\mathbb{E}_{\mathbb{D}}[(q_i - \dot{q}_i)^2]}_{variance} - \underbrace{\mathbb{E}_{\mathbb{D}}[\frac{1}{m}\sum_{i=1}^{m}[(q_i - \overline{q})^2]]}_{diversity}$$

The autors of [5], shows an experiment using the squared error in a Bagged regression trees, increasing before the ensemble size and after the maximum depth:

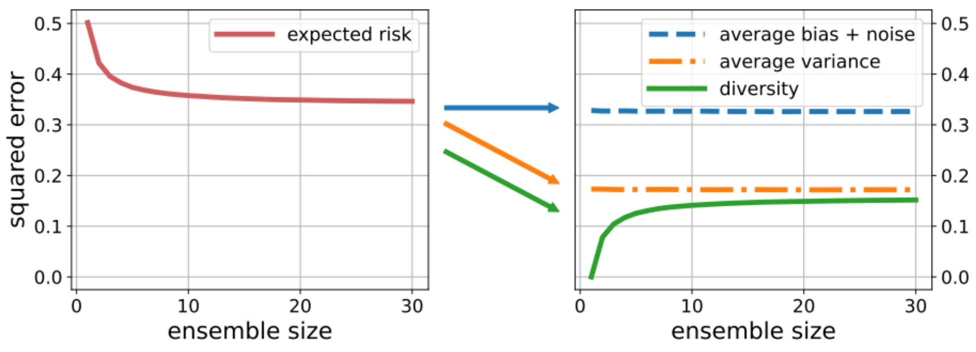


Figure 5.1: Bagging increasing ensemble size

As shown in 5.1, ranging the *ensemble size*, the expected risk decrease until the average bias: however the average bias and average variance are constant; in contrast, the *diversity* term increases with m-subtracting from the expected risk so the improvement obtained from the expected risk is determined entirely by diversity.

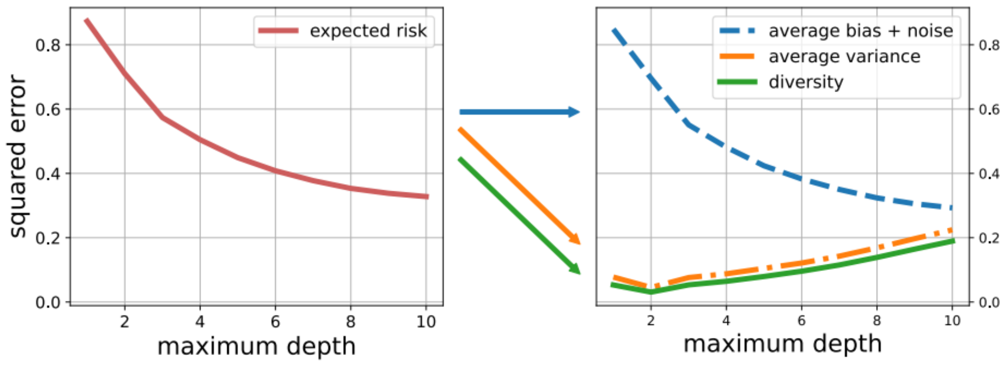


Figure 5.2: Bagging increasing maximum depth

In 5.2, varying the maximum depth of regression trees, we have a *bias-variance-diversity* trade-off because all of the three components change.

5.1.2 Estimating Bias, Variance and Diversity

In order to estimate three components, the authors of [5] employs concept of *trial*: the centroid $\dot{q}_i$ is estimated based on the clones of the single model extracted from the trials.

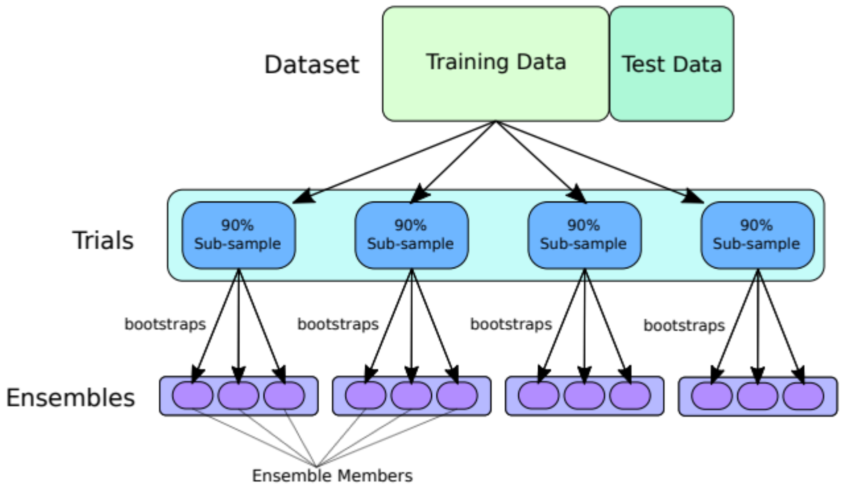


Figure 5.3: Estimating Components on [5]

Since we use Random Search to obtain weak learners pool, the concept of trials **is not applicable** within our applications: for this reason, we can approximate the centroid for model i on a single test point x on trial t ($\dot{q_i}^t$) as the predictions of the single weak learner. The result of this applied to **squared loss**:

$$\mathbb{E}_{\mathbb{D}}[(\overline{q}-y)^2] = \underbrace{\frac{1}{m}\sum_{i=1}^{m}[(q_i-y)^2]}_{bias} - \underbrace{\mathbb{E}_{\mathbb{D}}[\frac{1}{m}\sum_{i=1}^{m}[(q_i-\overline{q})^2]]}_{diversity}$$

where $\overline{q}$:

$$\overline{q} = \frac{1}{m}\sum_{i=1}^{m} q_i$$

Without considering $\dot{q}_i^t$, in our applications we cannot esitmate *variance therm* and, due to this, we can assume the *variance* **constant**. For what concern *bias*, is obtained with mean squared error between predictions of weak learner and ground truth; *diversity* therm instead, is obtained computing mean squared error between weak learner and the mean of predictions to the remaining weak learners into the pool.

5.2 MSE vs. Spearman Correlation: Relationship

Once the *aggregation phase* of predictions is completed, each model produces a ranked vector of predictions (both for classification and regression problems). This vector is used to measure the quality of predictions respect to the ground truth: to do this the **Spearman Rank Correlation** is computed. In this section will be detailed steps to investigate how the *adapted squared error* (inspired by [5]) behaves in relation to *Spearman Correlation Coefficient*, typically used into the ensemble evaluation procedure.

5.2.1 Components

Having a pool of 100 jobs:

1. Considering each time the *ensemble* made up of all the jobs except the one under analysis.
2. Without considering *variance* therm (5.1.2).
3. Using before the Mean Squared Error and after the Spearman Correlation Coefficient, into *cv* and *test* subset of data will be computed and evaluated:
   - **bias**: $\frac{1}{m}\sum_{i=1}^{m}[(q_i - y)^2]$
   - **diversity**: $\frac{1}{m}\sum_{i=1}^{m}[(q_i - \overline{q})^2]]$

   obtaining a single vector for each component related to Mean Squared Error and Spearman Coefficient.

We want to confirm with certainty whether the relationship between MSE and Spearman Rank Correlation is inversely correlated (as expected) or not, into our use-cases.

5.2.2 Diversity Therm

Averaging all the vectors obtained from MSE and Spearman Correlation computations, it can be observed:

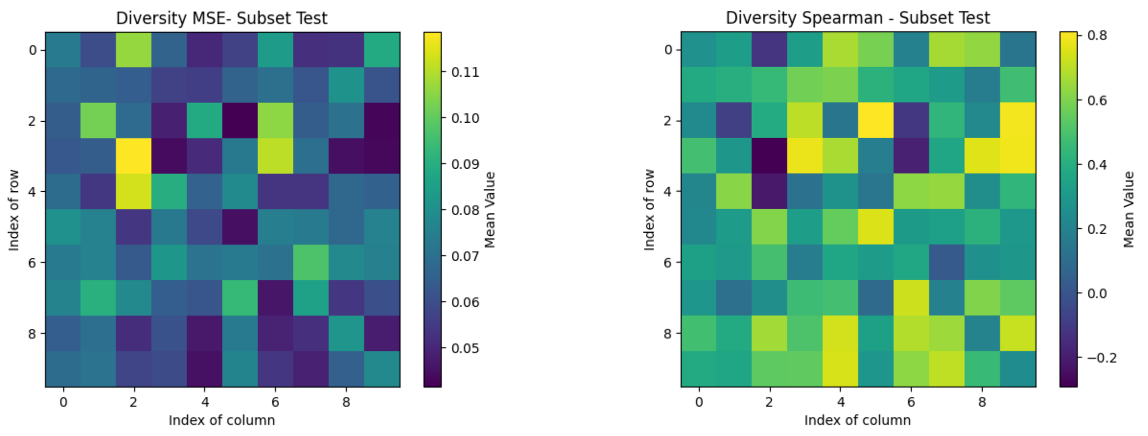


Figure 5.4: MSE vs Spearman Correlation - Diversity Therm

From these matrices, it can already be seen how the two metrics are inversely related: high MSE values indicate low Spearman Correlation values. To be more specific:

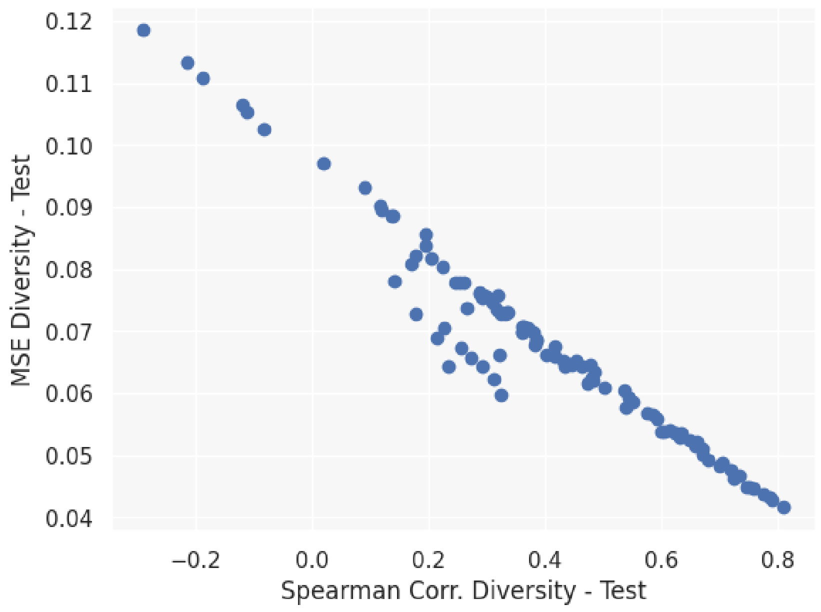
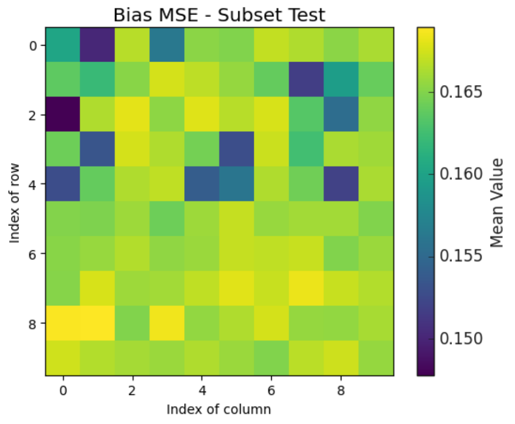


Figure 5.5: MSE/Spearman Diversity Visualization

Computing the correlation between values of two matrices yields: *-0.976*

5.2.3 Bias Therm

Averaging all the vectors obtained from MSE and Spearman Correlation computations, for *Bias therm*:



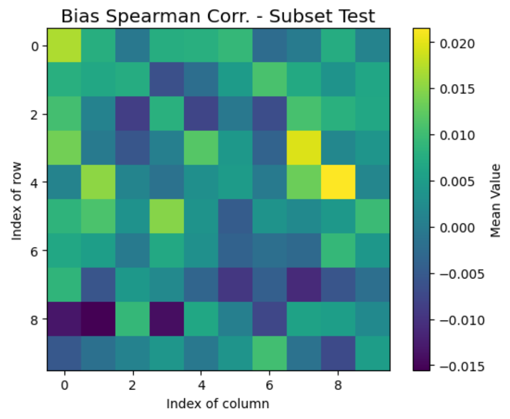


Figure 5.6: MSE vs Spearman Correlation - Bias Therm

Analyzing the values within these matrices and considering that values are related to the bias term, it has been observed that some jobs collapsed during the training and aggregation phase. Considering the mean standard deviation of the entire pool, it is possible to identify and remove all the jobs under a certain threshold (e.g 4%) of single standard deviation. The representation before and after removing the collapsed jobs will be shown.

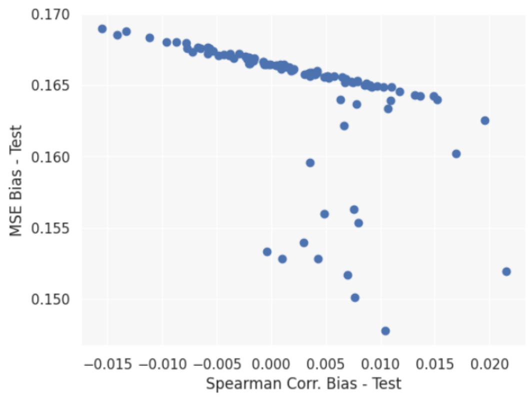


Figure 5.7: Before

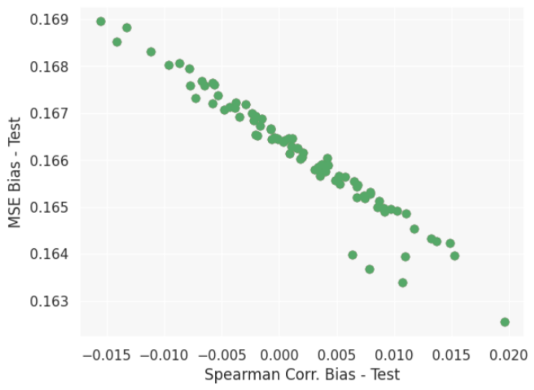


Figure 5.8: After

Figure 5.9: MSE/Spearman Bias Visualization

Computing the correlation between values of two matrices (removing collapsed jobs) yields: *-0.970*.

5.3 Metric Conclusion

Using the tests conducted on the relationship between the two metrics, it can be demonstrated that the two metrics are almost perfectly inversely correlated; this is the desired result, which allows the metric to be used to measure and quantify diversity within each pool of weak learners. In addition, with [5] is also demonstrated that diversity can be seen as a hidden dimension in the bias-variance decomposition of an ensemble **loss**; the exact functional form is specific on the loss being used, but the common structure applicable for any losses is:

$$\textbf{expected loss} = (\textbf{average bias}) + (\textbf{average variance}) - (\textbf{diversity})$$

6. Experiments on New Dataset

After the completion of the Random Search algorithm, the *select ensemble* procedure produce a JSON file containing the selected jobs on the entire pool of weak learners. In this section will be detailed some results and experiments conduct on a different dataset, at the end of strategy production process maximizing the usual metric.

Performance Correlation of Selected Jobs

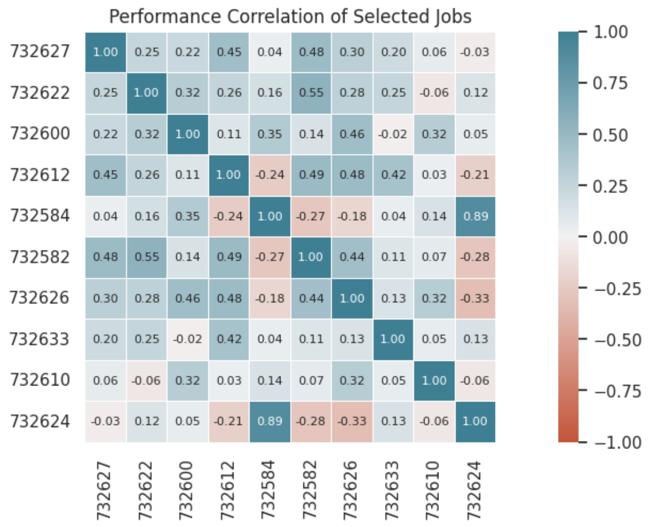


Figure 6.1: Selected Jobs Correlation

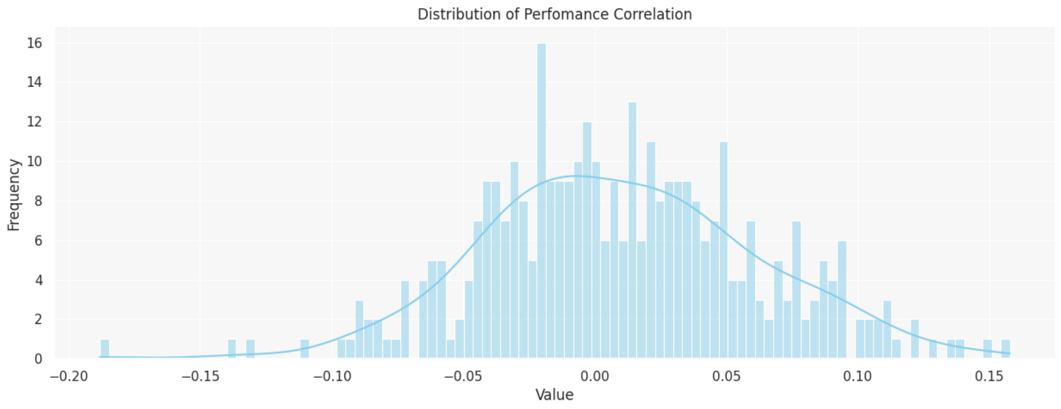


Figure 6.2: Distribution of Performance Correlation Values

Predictions of Selected Jobs through time

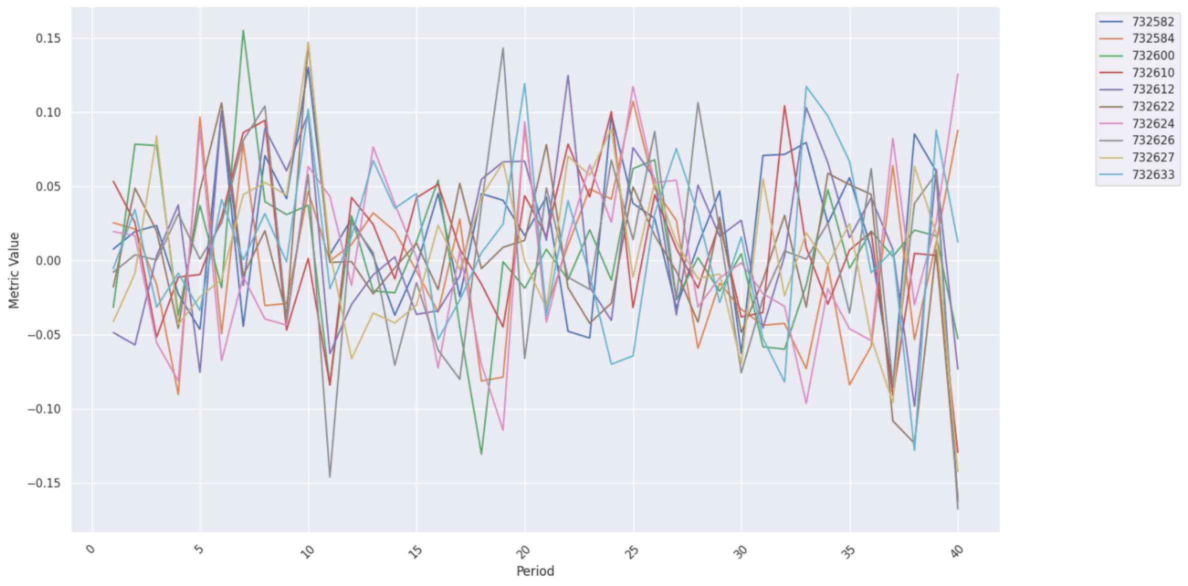


Figure 6.3: Selected Jobs Predictions

Diversity of Selected Jobs

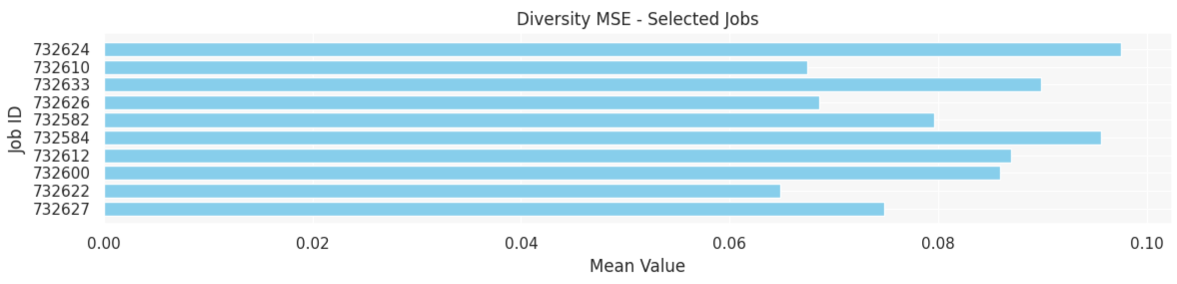
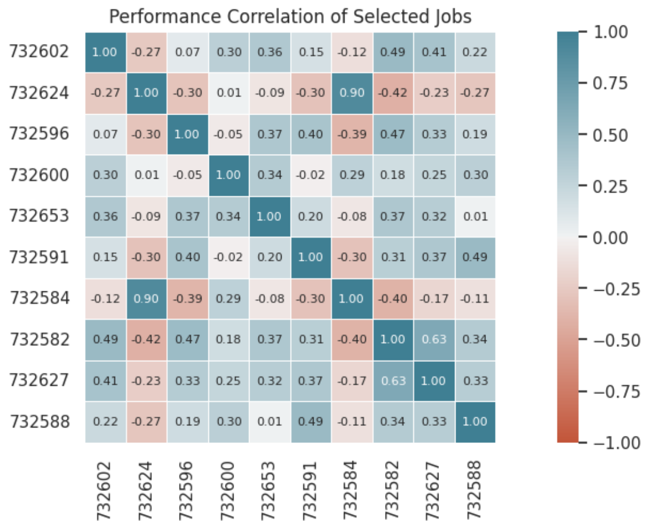


Figure 6.4: Diversity Selected Jobs

Once obtained the selected jobs, the same pipeline was executed by introducing the previously explained **diversity metric** into the selection process, on the same dataset and using the same parameters for the algorithm.

Performance Correlation Selected Jobs Using New Metric



	732602	732624	732596	732600	732653	732591	732584	732582	732627	732588
732602	1.00	-0.27	0.07	0.30	0.36	0.15	-0.12	0.49	0.41	0.22
732624	-0.27	1.00	-0.30	0.01	-0.09	-0.30	0.90	-0.42	-0.23	-0.27
732596	0.07	-0.30	1.00	-0.05	0.37	0.40	-0.39	0.47	0.33	0.19
732600	0.30	0.01	-0.05	1.00	0.34	-0.02	0.29	0.18	0.25	0.30
732653	0.36	-0.09	0.37	0.34	1.00	0.20	-0.08	0.37	0.32	0.01
732591	0.15	-0.30	0.40	-0.02	0.20	1.00	-0.30	0.31	0.37	0.49
732584	-0.12	0.90	-0.39	0.29	-0.08	-0.30	1.00	-0.40	-0.17	-0.11
732582	0.49	-0.42	0.47	0.18	0.37	0.31	-0.40	1.00	0.63	0.34
732627	0.41	-0.23	0.33	0.25	0.32	0.37	-0.17	0.63	1.00	0.33
732588	0.22	-0.27	0.19	0.30	0.01	0.49	-0.11	0.34	0.33	1.00

Figure 6.5: Selected Jobs Correlation - New Metric

As shown in 6.5 and 6.1, five jobs (out of ten) from those selected with the new metric are also present among those selected with the old metric.

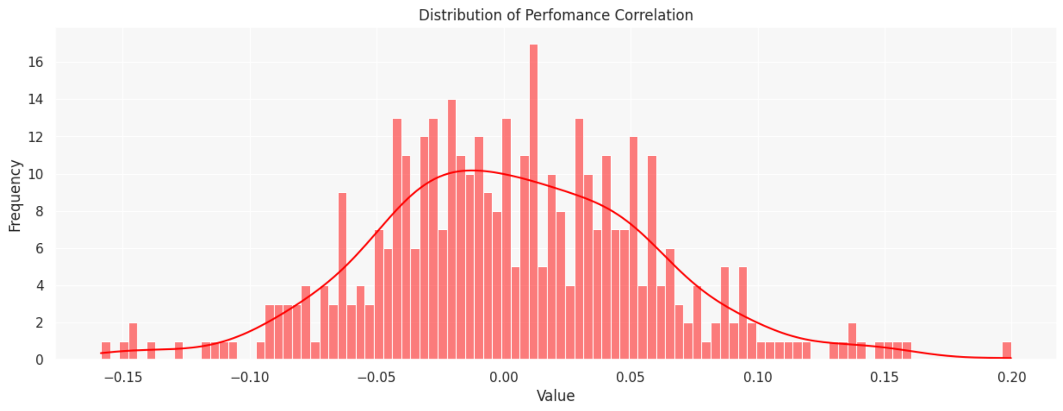


Figure 6.6: Distribution of Performance Correlation Values - New Metric

Predictions of Selected Jobs through time New Metric

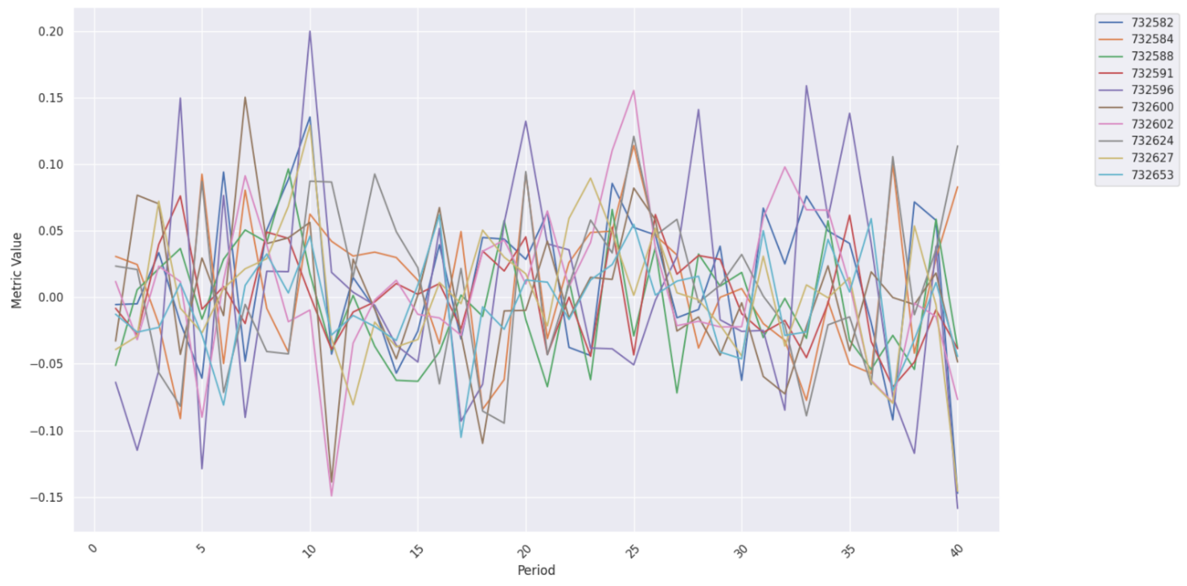


Figure 6.7: Selected Jobs Predictions using New Metric

Diversity of Selected Jobs - New Metric

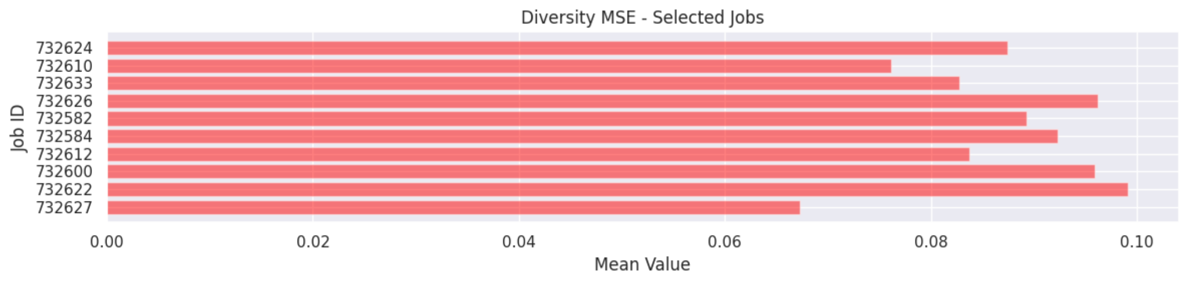


Figure 6.8: Diversity Selected Jobs using New Metric

Table 6.1: Bias and Diversity - Selected Jobs vs Entire Pool

Metric	Selected Jobs	Entire Pool
Mean Diversity (MSE)	0.08112	0.06762
Mean Bias (MSE)	0.15886	0.16384

Table 6.2: Bias and Diversity - Selected Jobs vs Entire Pool using New Metric

Metric	Selected Jobs	Entire Pool
Mean Diversity (MSE)	0.08701	0.06762
Mean Bias (MSE)	0.16164	0.16384

Comparing Tables 6.1 and 6.2, it can be observed that the jobs selected with the new metric have a **higher diversity** term than those selected with the old metric, while the bias term is also **slightly higher**. Additionally, the difference between bias and diversity for the jobs selected with the new metric is lower compared to that for the jobs selected with the old metric.

By adding this metric and using it to select the jobs within the ensemble (not just as an intrinsic measure of the terms), it was possible to obtain an ensemble of weak learners that are more diverse compared to those obtained with the existing metrics. This is notable considering that the tests conducted on this dataset were based on a Random Search of 100 elements, which is smaller than those typically performed (sometimes up to 500 weak learners) so having a smaller range of possibilities compared to usual.

Conclusions and Further Improvements

Through my thesis project, I was able to contribute to the new strategy production process at Axyon; this allowed for the integration of new features with some improvements within the tools used, while at the same time opening new avenues for research and further improvements. With the obtained results, we were able to create heterogeneous ensembles and introduce a considerable level of diversity within the pool of weak learners by using the tools provided by Random Search integrated with Optuna. The analysis of the selection process results, conducted using the diversity metric, revealed that although the selected weak learners present some differences, the diversity remains limited within the pool.

This observation leads us to consider that the current approach has inherent limitations. Currently, the generation of weak learners during the exploration phase, is based on models trained on the entire time period of the data. This approach is limiting not only because the entire time period is treated as a single block during the algorithm, but also because the number of weak learners generated may not be sufficient to effectively explore the entire solution space and identify the best individuals. As a result, there is a risk of not achieving the optimal level of diversity and quality within the pool.

A possible solution to this problem lies in the adoption of a metaheuristic approach. By training weak learners on specific time intervals, it would be possible to explore the space incrementally, allowing for the creation of models that are progressively influenced by previous exploration on reduced time periods. This approach would enable more targeted and optimized space exploration, where diversity is not just a measure aspect, but a central parameter driving the entire exploration.

We believe that integrating a metaheuristic approach could significantly improve the effectiveness of model space exploration, leading to the construction of more robust and diversified ensembles. As a result, this might improve the chances of identifying the best weak learners and enhancing overall performance.

Bibliography

[1] Yunquian Ma Cha Zhang. *Ensemble Machine Learning: Methods and Applications*. 1st ed. 233 Spring Street, New York, NY 10013: Springer New York, 2012. ISBN: 978-1-4419-9325-0.

[2] M.Wahde. *Biologically Inspired Optimization Methods*. 1st ed. 25 Bridge Street, Billerica, MA 01821, USA: WIT Press, 2008. ISBN: 978-1-84564-148-1.

[5] Danny Wood et al. *A Unified Theory of Diversity in Ensemble Learning*. 2024. arXiv: 2301.03962 [cs.LG]. URL: https://arxiv.org/abs/2301.03962.

Online Sources

[3] ***OptunaFramework***. *Documentation.* URL: https://optuna.readthedocs.io/en/stable/index.html.

[4] ***Scikit-Learn***. *API.* URL: https://scikit-learn.org/stable/api/index.html.

[6] ***XGBoost***. *Documentation.* URL: https://xgboost.readthedocs.io/en/stable/index.html.

Acknowledgments

I want to sincerely thank everyone who has supported me during this journey. My deepest gratitude goes to my family for their love and support, which has been crucial in helping me reach this milestone. I also want to thank my friends for always being there for me, providing encouragement and companionship.

A special thanks to my colleagues at Axyon, especially the ML Team Alessandro, Giovanni, Alberto, and Jacopo for believing in me, giving me the chance to work on this project, and continuing my career in such an inspiring and amazing environment.

I am incredibly grateful to my partner, Carlotta, whose constant love, patience, and encouragement have been a source of strength throughout this journey. Her belief in me, even during the most challenging moments, has been invaluable, and I could not have completed this work without her support.

Finally, not least in importance, I would like to thank prof. Simone Calderara, my supervisor, for his valuable guidance and mentorship during this project and my studies.