Astronomy & Astrophysics output
November 3, 2022
©ESO 2022

Application of a Machine Learning on COMPAS and STROOPWAFEL to Classify BBH Mergers

Baroni M.[1], Beduzzi L.[1], and Rizza C.[1]

[1] Department of Physics and Astronomy "G. Galilei", University of Padova, Vicolo dell'Osservatorio 3, 35122 Padova, Italy

ABSTRACT

The final goal of this project consists in using some random forest in order to predict which initial conditions will bring a given binary system to evolve and produce a binary black holes merger through a common envelope phase. The evolution of each binary system is obtained thanks to the binary population synthesis code COMPAS. Since this kind of target is quite rare, the importance sampling algorithm STROOPWAFEL is used to optimize the parameter space and reduce the computational cost of the simulation. The RandomForest package of Scikit Learn will allow to predict the final evolutionary stage of a binary system already from its initial conditions, without the need to run COMPAS further.

1. The Target

The target of this project consists of binary systems of black holes that merge through a common envelope phase within the Hubble time.
It is known that the final fate of two stars in a tight binary system is very different from the one of two single stars or two stars in a loose binary. Indeed, two stars in a binary might exchange mass, a fact which can alter their lifetime and properties.
There are three different processes of mass transfer, i.e. the wind mass transfer, the Roche lobe overflow and the common envelope. In particular, the common envelope phase describes an unstable mass transfer, which generally occurs if at least one of the two stars of a binary has a helium or carbon-oxygen core or if the accretor is a compact object. This process represents the formation of a diffuse envelope that surrounds the entire system an it can be summarized in the following steps:

– Two massive stars initially do not fill their Roche lobe, until the first one evolves out of the main sequence, it expands and starts to transfer mass onto the second.
– If matter does not flow only through the inner Lagrangian point, then the mass transfer becomes dynamically unstable (i.e. the star cannot shrink fast enough to keep hydrostatic equilibrium) and the common envelope phase begins.
– The drag exercised by the envelope leads the two cores to spiral in. If the energy released during the spiral in is large enough to remove the envelope, then the two cores will form a new tighter binary. Otherwise, they will merge becoming a single star.

What we are interested in are those systems which are able to eject the envelope and form a tighter binary, since they can evolve and produce binary black holes. Unfortunately, this kind of target is quite rare, so we will be forced to take advantage of an importance sampling algorithm in order to increase the number of samples of interest.

2. COMPAS

COMPAS is a binary population synthesis code which is designed so that evolution prescriptions and model parameters are easily adjustable. COMPAS draws properties for a binary star system from a set of initial distributions and evolves it from zero-age main sequence to the end of its life as two compact remnants.

2.1. Program Options

COMPAS is a command-line application. Interaction with COMPAS is entirely through the terminal and shell, so there is no visual or graphical user interface (GUI). It reads input files where necessary and produces output files, which are not interactive.
COMPAS provides a rich set of configuration parameters via program options, allowing users to vary many parameters that affect the evolution of binary stars. Furthermore, it accepts some parameters to be specified as sets of values, allowing users to specify a grid of parameter values on the command line. This fact allows users more flexibility and the ability to specify more complex combinations of parameter values.
As an alternative to the command line, users can make use of a grid file. Each line of a grid file is used by COMPAS to set the initial conditions and evolutionary parameters for a binary star, so each binary star is evolved using those values.

2.2. Output Files

All COMPAS output files are created inside a container directory, specified by the output_container program option. Detailed information about the simulation is written in the standard log files. In particular:

– The System_Parameters directory records summary information for all binary stars during evolution.
– The Supernovae directory records summary information for all stars that experience a supernova event during evolution.
– The Double_Compact_Objects directory records summary information for all binary systems that form double compact object during binary star evolution.
– The Common_ Envelopes directory records summary information for all binary systems that experience common envelope events during binary star evolution.

- The RLOF directory records detailed information about Roche-lobe overflow events during binary star evolution.
- The Run_Details directory records information for COMPAS program options (e.g. the COMPAS version, the start time of the simulation and the command-line option values).

COMPAS can produce log files in several formats, the two main important being the Hierarchical Data Format version 5 (HDF5) and the Comma Separated Values (CSV).

3. STROOPWAFEL

STROOPWAFEL is a python sampling package which provides an efficient sampling of the COMPAS input parameters. More precisely, it is an adaptive importance sampling algorithm that improves the computational efficiency of population studies of rare events, by focusing the simulation on regions of the initial parameter space found to produce outputs of interest.

3.1. The Method

The algorithm consists of three main steps, i.e. an exploration phase, an adaptation phase and a refinement phase.
In the exploration phase, STROOPWAFEL first explores the initial parameter space by sampling directly from the birth distribution until eventually a sufficient population of events of interest is found. Some examples of these initial parameters are the initial masses $m_{1,i}$ and $m_{2,i}$ of the two stars, their initial separation a_i, the metallicity m_i and the eccentricity e_i. The distribution of the initial conditions is often taken as the Monte Carlo sampling distribution.
In the adaptation phase, the algorithm constructs multivariate Gaussian distributions in the initial parameter space around each of the events of interest found during the exploration phase. Then it scales the width of each of the Gaussians with the local sampling density to obtain unbiased estimates of the target population. Finally, it creates the adapted sampling distribution $q(x)$ by combining the Gaussians into a mixture distribution, which is called instrumental distribution.
In the refinement phase, the remaining binaries are sampled from this instrumental distribution. To each sample is assigned a weight so that the predicted population reflects the birth distribution. However, when drawing from the instrumental distribution some samples will fall outside the physical range of the parameter space. Such samples can immediately be rejected and redrawn. By doing so, we in practice sample from the normalized physical mixture distribution $\tilde{q}(x)$.

3.2. The Exploratory Phase

An important choice in adaptive importance sampling algorithms is deciding when to switch from the exploratory phase to the refinement phase, since this choice can have a substantial impact on the performance of the algorithm. Indeed, leaving the exploratory phase too early can result in missing important regions of the initial parameter space that produce systems in the target population. On the other hand, switching to the refinement sampling phase too late will miss out on the advantages of the algorithm.
However, the fraction of the total number of samples that should be spent on the exploration phase f_{expl} is automatically chosen by STROOPWAFEL when the algorithm estimates the formation rate R_T of the population under study from the birth probability distribution (i.e. it is calculated during the exploration phase).
Let's assume to divide the volume of the input parameter space that successfully produces systems of interest into two parts, one which we have accurately found and one which remains missing. We then want to minimize the event rate uncertainty, so we require that the contribution to the event rate from potentially undiscovered islands is smaller than the sampling uncertainty in the rate contributed by the islands that are successfully found. Assuming that the target population is a rare outcome of the initial conditions, we obtain:

$$f_{expl} = 1 - \frac{z_1}{z_1 + \sqrt{z_2}}$$

In this expression z_1 is the total weight of the target binary-forming region, while z_2 is the weight of a region yet undiscovered. It is clear that for a rarer target population, a larger fraction of the total number of samples should be spent on the exploratory phase, because it takes longer to determine a good sampling distribution when the target formation rate is low. On the other hand, a more common target population can be optimally simulated with a relatively small exploratory phase, since we expect this will be enough to build up a good adaptive distribution.

3.3. The Adaptation Phase

In the adaptation phase, the Gaussian distributions constructed around the events of interest are parametrized by their means and covariance matrices. In particular, the covariance matrices determine the width of the Gaussian distributions and they are assumed to be diagonal matrices. The subsequent rescaling of the width of each Gaussian with the local density allows the algorithm to construct broader Gaussian distributions in the regions of the parameter space that are less densely explored.
It is also used a free parameter κ, which scales the widths of the Gaussian distributions and enables to regulate how tightly they cover the parameter space near the successful binaries. We have that small values of κ will increase the efficiency of STROOPWAFEL, but they also increase the chance of missing an important region of the output surface, since the Gaussian distributions are too narrow to properly cover it. On the other hand, large values of κ will decrease the efficiency of finding samples of interest in the refinement phase and lower the gain of STROOPWAFEL. From these considerations, a value of $\kappa = 2$ is usually adopted.

3.4. The Refinement Phase

The efficiency in the refinement phase is not perfect, i.e. not all the samples drawn in this phase will find an outcome from the target population. We have that during the refinement phase the algorithm is 45 – 650 times more efficient at finding hits than during the exploratory phase.
In particular, the different rareness of the target populations influences how much the efficiency increases during the refinement phase and also the duration of the exploratory phase. As a result, the overall gain in efficiency will be greater for rarer events and large simulations.

3.5. The Advantages

The great advantage of using STROOPWAFEL is that it increases the computational efficiency, allowing to find more events of the target population and so speeding up the simulation. Moreover, STROOPWAFEL allows to:

– Map the parameter space with higher resolution, which leads to a better knowledge of the initial conditions of a binary system that yield a binary of the target population.
– Obtain smaller variances in the distribution functions, which leads to a significant decrease in the statistical uncertainty of the predictions for the output parameter spaces.
– Recover the tails of distribution functions.
– Handle bifurcations and stochasticity that naturally occur in the parameter spaces of binary population synthesis simulations.

All these points show how STROOPWAFEL is superior to the traditional Monte Carlo sampling from the birth probability distributions. In particular this is true for rare events, where the gain in efficiency is very high.

4. THE CODE

A STROOPWAFEL[1] package has been created for python by Lieke Van Son and it is based on the work of **?** . This package contains eight scripts that completely define the STROOPWAFEL code.

4.1. Sw.py

The script *sw.py* is the main core of the code and it includes several functions:

– The function *_init_* creates an instance of the STROOPWAFEL class, initializing the total number of binary systems to evolve, the number of batches to run in parallel, the number of samples to run per batch, the location of the output folder, the output filename and three boolean parameters introduced to decide if we want to run the simulation on slurm based cluster HPC, if we want to include an adaptive importance sampling and if we want to show COMPAS output/errors.
– The function *update_fraction_explored* updates the fraction of region which has already been explored by the algorithm.
– The function *should_continue_exploring* estimates if the algorithm should continue exploring or it is ready to end the exploratory phase, giving a boolean value as output.
– The function *determine_rate* determines the rate of hits produced by the algorithm, giving its value and the relative uncertainty as output.
– The function *calculate_mixture_weights* calculates the weights for all the locations provided that need this computation.
– The function *process_batches* waits for the completion of the commands which were running in batches.
– The function *initialize* is run only once in the STROOPWAFEL class and it initializes the associated variables and the function calls that the user will specify (e.g. the interesting systems method, the rejected systems method, the update properties method and the configure code run method).
– The function *explore* defines the exploration phase of the STROOPWAFEL algorithm. It requires as input an initial distribution function which shows how to sample from.
– The function *adapt* defines the adaptive phase of the STROOPWAFEL algorithm. It requires to know what kind of distribution has to be adapted for the refinement phase.
– The function *refine* defines the refinement phase of the STROOPWAFEL algorithm.
– The function *postprocess* defines the post-processing phase of the STROOPWAFEL algorithm. Usually it is used to print only the hits.

[1] https://pypi.org/project/stroopwafel/

4.2. Utils.py

In the script *utils.py*:

– The function *generate_grid* generates a .txt file with the locations specified.
– The function *print_samples* prints all the hits to a file that will be saved.
– The function *read_samples* reads the samples from a given file and converts them to location objects. As a result a list of locations is generated.
– The function *run_code* runs the commands specified on the command shell.

This script also provides two useful functions to calculate the radius of a star at the zero age main sequence phase (given the mass and the metallicity) and the Roche lobe radius of a binary (given the masses of the two objects).

4.3. Classes.py

The script *classes.py* first defines the class *Dimension*, where:

– The function *_init_* defines the dimensions that will be stroopwafelized, fixing the maximum and minimum value that they can take. It also requires to choose the type of sampling to be used for the given dimension during the exploratory phase (it should be chosen from the class *sampler*), as well as the function that will be used to calculate its prior.
– The function *run_sampler* samples the variable based on the given sampler class.
– The function *is_sample_within_bounds* returns which samples are within the bounds of this variable.

Then this script defines the class *Location*, which describes a point in a N-dimensional space. Of fundamental importance are the dictionaries *dimensions* and *properties*, used to create a map between the class *Dimension* and its float value and to store properties which we do not want to stroopwafelize, respectively. Within this class:

– The function *_init_* initializes the dictionaries *dimensions* and *properties*.
– The function *create_location* creates a location instance when supplied with a dimension hash and the row of samples. Each key of the row which corresponds to a dimension goes to the dimensions property and the rest goes to the properties property.
– The function *to_array* converts the current object of the Location class to an array sorted by the key name.
– The function *revert_variables_to_original_scales* converts back each value of the location to the original scale defined in the interface.
– The function *transform_variables_to_new_scales* converts each value of the location to the new transformed scale.

4.4. Distributions.py

The script *distributions.py* first of all defines the class *NDimensionalDistribution*, which is a parent class for any N-dimensional distribution, i.e. any subclass of this class must implement all the methods defined in this class.
Examples of N-dimensional distributions are the *Gaussian* class and the *InitialDistribution* class. The former will be used during the refinement phase to draw adapted distributions, the latter will be used during the exploratory phase.

4.5. Other Scripts

The script *sampler.py* defines different kinds of sampler, such as the uniform sampling, the flat sampling and its logarithmic version, the kroupa sampling, the uniform cosine sampling and the uniform sine sampling, while *prior.py* defines the related birth priors. Finally, the script *constants.py* contains all the constants.

5. STROOPWAFEL INTERFACE

The COMPAS team[2] has provided an interface version of STROOPWAFEL named *StroopwafelInterface.py*, which enables the users to run the population synthesis code COMPAS together with all the benefits supplied by this importance sampling algorithm.
The script has been split into two main parts: the first one defines all the functions required by STROOPWAFEL, while the second one is the main, i.e. where the STROOPWAFEL functions are effectively used.
In the following subsections we will briefly discuss what are the functions implemented in this script.

5.1. Create_dimensions

The function *create_dimensions* creates all the dimensions for STROOPWAFEL, i.e. it defines what are the variables that we want to sample. The *Dimension* class defined in *classes.py* is invoked to create objects for each variable. Note that it is mandatory to choose for each dimension the maximum and minimum value that it can take, the type of sampling and the function that will be used to calculate the related prior.
As output, this function returns a list containing all the instances of the *Dimension* class, i.e. it provides a grid file with all the dimensions the user has chosen to print with the argument *should_print*.

5.2. Update_properties

The function *update_properties* is not mandatory and it is required only if you have some dependent variable. For example, in our case we wanted to sample $Mass-1$ and q, then $Mass-2$ is a dependent variable which is the product of the two.
As input it takes two arguments:

- The first one is *locations*, i.e. a list containing objects of the *Location* class in *classes.py*.
- The second one is *dimensions*, which contains the values that has been returned by the previously discussed *create_dimensions* function.

5.3. Configure_code_run

The function *configure_code_run* tells STROOPWAFEL what program to run, along with its arguments. As input it takes the batch, i.e. the dictionary which stores some information about the run. Note that for every run a number key is defined, which stores the unique id of the run.
This function also has a sub-process which will run under the key process and that will store additional information for each batch run depending on the user. For example, in our code we have stored the *output_container* and the *grid_filename* so that it is possible to read them during the subsequent discovery of interesting systems.

[2] https://github.com/TeamCOMPAS

5.4. Interesting_systems

The function *interesting_systems* is the most important one of the script since it tells STROOPWAFEL what an interesting system is, i.e. what is the target of our simulation. It takes the batch dictionary as input and returns the number of interesting systems found as output. All the of them will be identified at the end by the key *is_hit*.
Note that this function also allows to add some information in the output files. For example, in our code we have stored the first and second common envelope times (if they happen) and the final mass of the two black holes.

5.5. Selection_effects

The function *selection_effects* is not a mandatory function and it is written to support selection effects. In particular, it takes STROOPWAFEL objects as input and it adds a weight for each target system.

5.6. Rejected_systems

The function *rejected_systems* selects which systems cannot be evolved in order to save computational time. It takes as input a list of locations to inspect and gives as output the number of systems which can be rejected.

5.7. The Main

The main of the code is defined in the second part of the script and it consists of four steps:

- The first step imports and assigns the input parameters for STROOPWAFEL, e.g. the total number of systems, the number of cores to run in parallel, the number of systems to generate in one core, if debug of COMPAS has to be printed, if we want to run in MC simulation mode only, if we are running on a slurm-based HPC, the output filename and the output folder name. These parameters can be overridden with *pythonSubmit* parameters, if desired.
- The second step creates an instance of the STROOPWAFEL class.
- The third step initializes the STROOPWAFEL objects with the user defined functions and creates the dimensions and the initial distributions.
- The fourth and final step runs STROOPWAFEL. It starts with the exploratory phase, followed by the adaptation phase and the application of the selection effects. Finally, we have the refinement phase and the weighting of the samples.

6. The Run

In order to produce a large sample of double compact objects, we chose to run $2 \cdot 10^6$ binary systems (500 binaries per batch). Obviously, the number of batches will follow from the number of binaries, the number of cores and the number of samples per core.
We defined as our target all the binaries that, at the end of their evolution, will give as output a black hole-black hole merger within the Hubble time. This means that the code will read all the outputs of the simulation, assigning the key *is_hit* to all the interesting systems.
We have chosen to use a uniform sampling for the initial distribution of all the parameters, with the exception of the flat in logarithm sampling applied to the semi-major axis. This was done

Table 1: Parameters ranges, given as maximum and minimum values, for the initial samplings.

Parameter	Minimum value	Maximum value
Initial mass 1 [$M_\odot$]	5	150
Semi-major axis [$R_\odot$]	0.01	10^3
Mass ratio	0.01	1
Metallicity	10^{-4}	0.03
Eccentricity	0	0.9

to increase the number of targets found at the end of the simulation. The parameters ranges are shown in table 1: the choice of their maximum and minimum values refers to the limits of the COMPAS input parameters and to the physical limits of real systems.
After having initialized STROOPWAFEL, the code starts with the exploratory phase, which successfully simulates around the 22.8% of the sample. The number of hits in this first phase gives about 1624 binary systems, which corresponds to the 0.6% of the simulated sample. At this point the adaptation phase begins, the code constructs the Gaussians around the interesting points in the parameters space and it combines them to create the adapted sampling distribution. Finally, the refinement phase starts and the code samples all the remaining binary systems from this instrumental distribution.
The result is that during the refinement phase the code finds 55423 interesting systems over a total of 771500 binary systems simulated, which corresponds to the 7.2% of the final sample. Therefore, it is clear how the implementation of STROOPWAFEL within COMPAS improves the computational efficiency of our population study of binary black hole mergers, having an increase of one order of magnitude in finding targets of interest. The plots in figure 1 show the initial distribution of the mass ratio q between the two objects of a binary, before and after the adaptation phase.

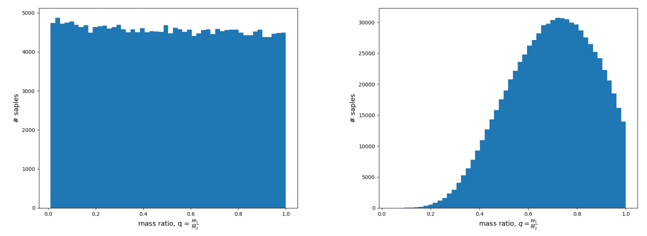

Fig. 1: *Left*: initial distribution of the mass ratio q before the adaptation phase; *right*: same distribution after the adaptation phase.

It is worth to note that the final sample produces a larger fraction binary black holes that do not merge within the Hubble time compared to the ones that successfully merge within it. Nevertheless, the algorithm is very efficient, allowing to study rare events with a significantly lower computational cost.

7. RANDOM FOREST CLASSIFIER

RandomForestClassifier is a class of the *RandomForest* algorithm included in the module *sklearn.ensemble* of scikit-learn, which is a free software machine learning library for python. It is based on randomized decision trees and in particular it consists of a perturb-and-combine technique. This means that a diverse set of classifiers is created by introducing randomness in the classifier construction and the prediction of the ensemble is given by the averaged prediction of the individual classifiers.
More specifically, the relative rank of a feature used as a decision node in a tree can be used to assess the relative importance of that feature with respect to the predictability of the target variable, i.e. features used at the top of the tree contribute to the final prediction decision of a larger fraction of the input samples. By averaging the estimates of predictive ability over several randomized trees, RandomForestClassifier is able to decrease the variance of such an estimate and use it for feature selection (this is known as the mean decrease in impurity, or MDI).
Note that forest classifiers have to be fitted with two arrays:

- An X array of shape *(n_samples, n_features)* holding the training samples.
- An Y array of shape *(n_samples)* holding the target values for the training samples.

7.1. Parameters

In RandomForestClassifier each tree in the ensemble is built from a sample drawn with replacement from the training set, i.e. from a bootstrap sample. Furthermore, when splitting each node during the construction of a tree, the best split is found either from all input features or from a random subset of size *max_features*.
Let's now take a look at its main parameters:

- *n_estimators* is the number of trees in the forest. The larger the better, but also the longer it will take to compute. Note that the results will stop getting significantly better beyond a critical number of trees. It must be an integer and its default value is equal to 100.
- *criterion* refers to the function used to measure the quality of a split. Supported criteria are *gini* for the Gini impurity and *log_loss* and *entropy* both for the Shannon information gain. In particular, the *gini* criterion is defined as:

  $$H(Q_m) = \sum_k p_{mk}(1 - p_{mk})$$

  where p_{mk} is the proportion of class k observations in node m. The *entropy* on the other hand is effectively the minimization of the *log_loss* criterion:

  $$LL(D,T) = -\frac{1}{n} \sum_{(x_i,y_i)\in D} \sum_k I(y_i = k) \log(T_k(x_i))$$

  where D is a training dataset of n pairs (x_i, y_i) and $T_k(x_i)$ is the probabilistic prediction. For our model we used the *gini* and *entropy* criterions.
- *max_depth* is the maximum depth of the tree. It also must be an integer and its default value is *None* (in this case the nodes are expanded until all leaves are pure or until all leaves contain less than *min_samples_split* samples).
- *min_samples_split* is the minimum number of samples required to split an internal node. It can be an integer or a float and its default value is equal to 2.
- *min_samples_leaf* is the minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least *min_samples_leaf* training samples in each of the left and right branches. It can be an integer or a float and its default value is equal to 1.

- *max_features* is the number of features to consider when looking for the best split. The lower the greater the reduction of variance, but also the greater the increase in bias. It can be an integer or a float, but supported arguments are also *sqrt*, *log2* and *auto*. The default value is *sqrt* and in this case *max_features=sqrt(n_features)*. For this work we used the *sqrt*, *log2* and *auto* options.
- *max_leaf_nodes* allows to grow trees with a maximum number of leaf nodes. It must be an integer and its default value is *None* (in this case we will have an unlimited number of leaf nodes).
- *bootstrap* is a parameter required if bootstrap samples are used when building trees. It must be a boolean and its default value is *True* (if *False* the whole dataset is used to build each tree).
- *class_weight* defines the weights associated with classes. It can be a dictionary or a list of dictionaries, but accepted modes are also *balanced* and *balanced_subsample*. The default value is *None*, in which all classes are supposed to have weight equal to one. We did not used this hyperparameter in this work and left it at its default value.

7.2. Cross Validation

Learning the parameters of a prediction function and testing it on the same data is a methodological mistake: a model that would just repeat the labels of the samples that it has just seen would have a perfect score but it would fail to predict anything useful on yet-unseen data (this situation is called over-fitting). To avoid it, it is common practice when performing a machine learning experiment to hold out part of the available data as a test set. In scikit-learn a random split into training and test sets can be quickly computed with the *train_test_split* helper function.
However, when evaluating different settings for estimators there is still a risk of over-fitting on the test set, because the parameters can be tweaked until the estimator performs optimally. To solve this problem, yet another part of the dataset can be held out as a so-called validation set: training proceeds on the training set, after which evaluation is done on the validation set and when the experiment seems to be successful the final evaluation can be done on the test set.
The problem is that by partitioning the available data into three sets, we drastically reduce the number of samples which can be used for learning the model and the results can depend on a particular random choice for the pair of *(train, validation)* sets. A possible solution is given by a procedure called cross-validation (CV): a test set should still be held out for final evaluation, but the validation set is no longer needed when doing CV. In the basic approach, called k-fold CV, the training set is split into k smaller sets. The following procedure is followed for each of the k folds:

- A model is trained using $k-1$ of the folds as training data.
- The resulting model is validated on the remaining part of the data.

The performance measure reported by k-fold cross-validation is then the average of the values computed in the loop. This approach can be computationally expensive but does not waste too much data, allowing the control the over-fitting.

7.3. Hyper-parameters

Hyper-parameters are parameters that are not directly learnt within estimators. In scikit-learn, two generic approaches to parameter search for the best cross validation score are provided:

- For given values, GridSearchCV exhaustively considers all parameter combinations.
- RandomizedSearchCV can sample a given number of candidates from a parameter space with a specified distribution.

7.4. RandomizedSearchCV

The RandomizedSearchCV method implements a randomized search over parameters, where each setting is sampled from a distribution over possible parameter values. This has two main benefits over an exhaustive search: a budget can be chosen independently on the number of parameters and possible values; moreover, adding parameters that do not influence the performance does not decrease efficiency. The main parameters of *sklearn.model_selection.RandomizedSearchCV* are listed below:

- *estimator*, *param_grid*, *n_jobs*, *cv* and*verbose* are the same parameters we have discussed for GridSearchCV.
- *n_iter* defines the number of parameter settings that are sampled. It must be an integer and its default value is equal to 10.
- *pre_dispatch* controls the number of jobs that get dispatched during parallel execution. Reducing this number can be useful to avoid an explosion of memory consumption when more jobs get dispatched than CPUs can process. This parameter can be *None* (in which case all the jobs are immediately created and spawned), an integer (giving the exact number of total jobs that are spawned) or a string (giving an expression as a function of *n_jobs*).
- *random_state* allows to use a pseudo random number generator state for random uniform sampling from lists of possible values instead of *scipy.stats* distributions (it must be an integer).

The main advantage of the randomized search over the grid search is the fact that, given a probability p that our results lay in a quantile q, it can be proven that the minimum number of iterations that we need to do in order to obtain those values is:

$$n \geq \frac{\log(1-p)}{\log(q)}$$

If we set $p = q = 0.99$, then we have that $n_{min} \approx 458$. For this work we will use a value of $n = 500$.

7.5. GridSearchCV

The grid search provided by GridSearchCV exhaustively generates candidates from a grid of parameter values specified with the *param_grid* parameter. In this way all the possible combinations of parameter values are evaluated and the best combination is retained. The main parameters of *sklearn.model_selection.GridSearchCV* are listed below:

- *estimator* is assumed to implement the scikit-learn estimator interface and it requires as input the estimator object.
- *param_grid* requires a dictionary with parameters names as keys and lists of parameter settings to try as values, or a list of such dictionaries. This enables searching over any sequence of parameter settings.

- *n_jobs* defines the number of jobs to run in parallel. It must be and integer and the default value is *None*, which means 1 (while -1 means using all processors).
- *cv* determines the cross-validation splitting strategy. Possible inputs are *None* (to use the default 5-fold cross validation) or an integer (to specify the number of folds).
- *verbose* controls the verbosity (the higher, the more messages). In particular, if > 1 the computation time for each fold and parameter candidate is displayed; if > 2 the score is also displayed; if > 3 the fold and candidate parameter indexes are also displayed together with the starting time of the computation.

7.6. Useful Tools

A very useful attribute of both GridSearchCV and RandomizedSearchCV is *best_params_*, which is a a parameter setting that gives the best results on the hold out data (its input must be a dictionary).
GridSearchCV and RandomizedSearchCV also implement some useful methods:

- *fit* is used to run a fit with all sets of parameters.
- *predict* allows to work on the estimator with the best found parameters.

The *sklearn.utils* and the *sklearn.tree* modules include various utilities, among which *sklearn.utils.shuffle* allows to shuffle arrays or sparse matrices in a consistent way, while *sklearn.tree.plot_tree* can be used to plot a decision tree.

8. Machine Learning Methods

We designed a code that we used either for the multi-channel classification and the 2-channel classification.
Firstly we designed two functions:

- *Hot_Encode* is a Cython written function used to perform the one-hot encoding task. In particular, it checks which of the labels are flagged for each element in the dataset and it assigns to each element a vector of given length with just a one in a given position that is different for each class and zeros elsewhere. The classes where divided and encoded as described below:

$$\text{5-channel} \longrightarrow \begin{cases} \text{BBH + CE + merge} \longrightarrow [1,0,0,0,0] \\ \text{BBH + merge} \longrightarrow [0,1,0,0,0] \\ \text{BBH + CE} \longrightarrow [0,0,1,0,0] \\ \text{BBH} \longrightarrow [0,0,0,1,0] \\ \text{No BBH} \longrightarrow [0,0,0,0,1] \end{cases}$$

$$\text{2-channel} \longrightarrow \begin{cases} \text{BBH + merge} \longrightarrow [1,0] \\ \text{BBH} \longrightarrow [0,1] \end{cases}$$

  This function was written in Cython to take advantage of its native parallelization in order to compensate for the high computational cost of the one-hot encoding procedure.
  The function takes as inputs the length of the dataset L, the number of classes *n_class* in which the dataset will be subdivided , the number of variables the dataset has *n_var*, the variable dataset given as an array *dataset*, the label dataset given as an array *labels* and the number of threads for the parallelization *num_t*. The final output of this function consists of a 3-dimensional matrix of shape $L \times n \times 2$, where n is defined as:

$$\begin{cases} n = n\_class \iff n\_class \geq n\_var \\ n = n\_var \iff n\_class < n\_var \end{cases}$$

- *Equalize* is a Python written function used to flatten the dataset up to the point in which each class has the same numerosity, which will be the numerosity of the less represented class. This function takes as inputs the variable dataset as a pandas dataframe *variable*, the label dataset as a pandas dataframe *label*, the numerosity of the less represented class *minimum*, the number of classes n and the length of the dataset L. The final output consists of two numpy arrays, when referring to the variables and the other to the labels.

After the functions definition we organized the work in the following order:

1. We uploaded the dataset and set the number of classes, the number of variables and the number of threads for the parallel computation of the one-hot encoded dataset. Moreover, for this phase we restricted the portion of dataset to a length L, in order to prepare it for the hyper-parameters optimization phase. The value of L will be different between the 2-channel classification ($L = 10^4$) and the multi-channel classification ($L = 10^5$) to compensate the less numerosity of the classes in the latter classification task.
   At this point we took the dataframe components we needed:
   - As *variables* we chose the initial mass of the more massive star, the mass ratio of the binary, the metallicity, the semi-major axis and the orbital eccentricity.
   - As *labels* we chose the merge flag, the binary black holes (BBH) flag and the two times at which the common envelope phase happens.

   After this, we passed the datasets into the one-hot encoding function and we re-separated the output into a variable and a label datasets. Then we transformed the numpy arrays into two pandas dataframes and we passed them into the equalize function, thus obtaining our final datasets in the form of numpy arrays.
2. Then we used the *train_test_split* function of sklearn to separate the datasets into train and test sets for the randomized search phase. In these two sets the percentage of the training size was put above 99% to not consider the test dataset, but we anyhow used the before mentioned function to make the data readable to the sklearn functions of the randomized search and the subsequent grid search.
3. Moving on, we defined the hyper-parameters grid from which the RandomizedSearchCV function would sample and we started the randomized search with 5 folds. Then we defined a smaller 2-dimensional grid (*n_estimators*, *max_depth*) in the surrounding of the best model found by the randomized search and we started a grid search always using 5 folds, thus obtaining our final best model. We chose to use only these two hyper-parameters because of the computational time associated with the grid search and of the fact that they represent the two main parameters of a random forest. The randomized search and grid search grids used in both tasks are presented in tables 2, 3, 4 and 5.
4. We repeated points 1 and 2, but this time with the parameter L set equal to the length of the dataset and the train set size of 80%, with the remaining part being the test set.

5. Then we performed the final training of the model with the complete equalized dataset and we estimated the accuracy of the model on the test set, remembering that the accuracy is calculated as the following:

$$A = \frac{TP + TN}{n}$$

where TP and TN are the True-Positive and True-Negative outcomes of the classification, while n is the total number of elements in the dataset.
6. After, using the *feature_importance_* attribute of the model, we extracted the feature importance score for each variable and, using the *metrics* module of sklearn, we also plotted the confusion matrix.
7. Finally, as last step we saved the model.

Hyper-parameter	**Range**	**Step**
n estimators	[100,10000]	100
max features	[sqrt,log2,auto]	×
max depth	[0,1000] w None	50
min samples split	[2,10]	1
min samples leaf	[2,10]	1
bootstrap	[True,False]	×
criterion	[gini,entropy]	×

Table 2: Hyper-parameters ranges used for the randomized search in the 5-channel classification task.

Hyper-parameter	**Range**	**Step**
n estimators	[6225,6375]	25
max depth	[70,130]	10

Table 3: Hyper-parameters ranges used for the grid search in the 5-channel classification task.

Hyper-parameter	**Range**	**Step**
n estimators	[100,1000]	100
max features	[sqrt,log2,auto]	×
max depth	[0,100] w None	10
min samples split	[2,10]	1
min samples leaf	[2,10]	1
bootstrap	[True,False]	×
criterion	[gini,entropy]	×

Table 4: Hyper-parameters ranges used for the randomized search in the 2-channel classification task.

Hyper-parameter	**Range**	**Step**
n estimators	[325,475]	25
max depth	[10,40] w/ None	10

Table 5: Hyper-parameters ranges used for the grid search in the 2-channel classification task.

9. Results

The final goal of this project is to run a random forest on our samples, in order to predict which values of the initial parameters space will bring a given binary system to evolve and produce a binary black holes merger within the Hubble time.
In the following sections we will present our results, both for the multi-channel classification task and the 2-channel classification task.

9.1. 5-Channel Classification

Thanks to the randomized search a best hyper-parameters set was obtained, leading to an accuracy value when applied to the reduced dataset of:

$$Accuracy_{RS} = 0.546$$

The grid search gave us a new best hyper-parameters set (it is visible in table 6), that when applied to the reduced dataset leads to an accuracy value more or less equal to the previous one:

$$Accuracy_{GS} = 0.546$$

Hyper-parameter	**Value**
n estimators	6375
max features	sqrt
max depth	80
min samples split	4
min samples leaf	2
bootstrap	False
criterion	entropy

Table 6: Hyper-parameters obtained by the grid search.

This probably means that the variability inside the hyper-parameters space is too small to have some real effects on the classification accuracy.
The training and further testing gave us the following results:

$$Accuracy = 0.820$$

We can assert that this value is quite good, especially considering the reduced dimension of the equalized dataset. Obviously, it is possible that bigger datasets could lead to increasing values of accuracy. In any case, it is surely notable the growth in accuracy from the reduced dataset used in the hyper-parameters optimization phase compared to the actual value obtained from the whole equalized dataset.
The performance of the feature importance score is provided in figure 2 and table 7. We can note that almost 50% of the total variance is explained by the semi-major axis and metallicity of the binary. This fact makes perfectly sense if we consider that either for the merging within the Hubble time and for the occurring of a common envelope phase the separation of the binaries cannot be too high. On the other hand, the metallicity is highly correlated with the mass-loss rate in the RGB phase, thus correlating also with the formation of a common envelope in the binary ($\dot{M} \propto Z^{0.7}$). In any case, we can easily see that each variable we considered plays an important role, with the least explained variance coming from the orbital eccentricity and leading to more than the 13% of the total.
In the end, from the confusion matrix given in figure 3 it is clear that there is a systematic error that causes our model to misclassify a not so small percentage of the non *BBH_MERGE_CE* classes into this one. This could may be due to the fact that common envelope events are anyway chaotic processes. Modelling these events as sparse points in the parameters space could cause the crossing of some well-bound class zones, hence leading to a misclassification of the samples.

Variable	**% of explained variance**
a	25.18
Z	23.48
M_1	20.95
q	17.19
e	13.20

Table 7: Feature importance table.

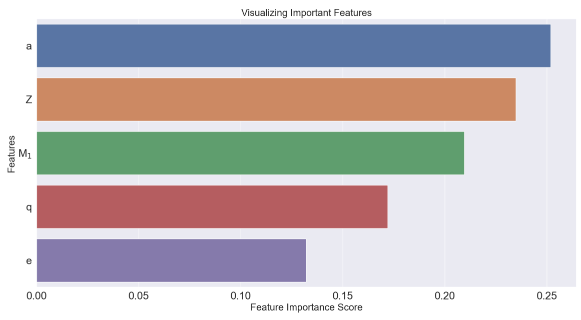

Fig. 2: Histogram of the features relative importance.

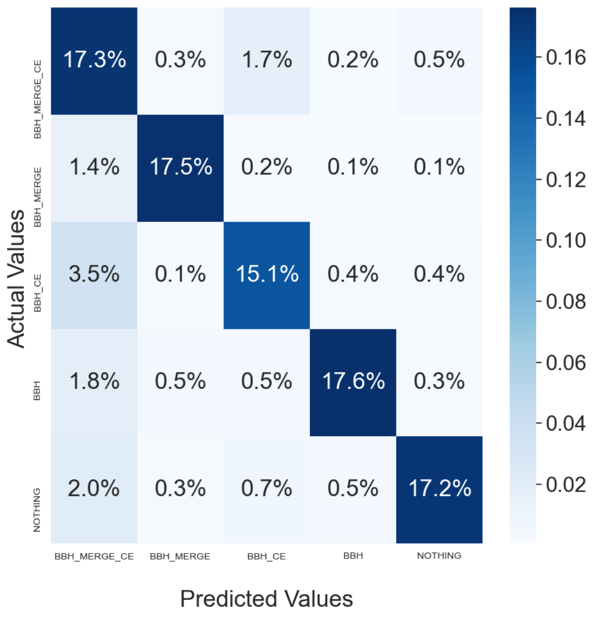

Fig. 3: Confusion matrix for the 5-channel classification.

9.2. 2-Channel Classification

As before, from the hyper-parameters optimization phase we obtained a best model (visible in table 8), leading to an accuracy value when applied to the reduced dataset of:

$$Accuracy_{RS} = 0.715$$

$$Accuracy_{GS} = 0.849$$

Then, the training and further testing gave us the following result:

$$Accuracy = 0.920$$

Hyper-parameter	**Value**
n estimators	375
max features	sqrt
max depth	30
min samples split	7
min samples leaf	2
bootstrap	False
criterion	entropy

Table 8: Hyper-parameters obtained by the grid search.

Also for this task we can see that the accuracy value goes up from using the reduced dataset in the optimization phase to the total equalized dataset. In this case the accuracy value is a lot higher than in the previous task, but this obviously is related to the fact that the classification here is narrower and that this task did not included the common envelope phase as part of the label dataset. Moreover, from the fact that we have only two classes, the numerosity of the less numerous class is higher than in the 5-channel classification task.
As before, the features relative importance was performed (it is provided in figure 4 and table 9). In this case we can easily see that half of the explained variance is given by the Keplerian elements of the orbit (i.e. *a* and *e*), having the semi-major axis accounting for up to slightly over 30% of the explained variance. This is explainable considering the goal of the 2-channel classification: wanting to classify the dataset by the means of only the merging and neglecting the common envelope, it makes sense for the dynamical components to be dominant, especially for the relative distance of the binary components. Regarding the physical parameters of the stars, we can easily see that also in this case the metallicity plays an important role, explaining almost the same variance as the eccentricity. From the confusion ma-

Variable	**% of explained variance**
a	30.91
e	19.29
Z	19.14
M_1	16.05
q	14.61

Table 9: Feature importance table.

trix we can see that now we do not have the systematic offset that we had with the multi-channel classification, nevertheless we have more misclassification from the non-merging binaries to the merging binaries.

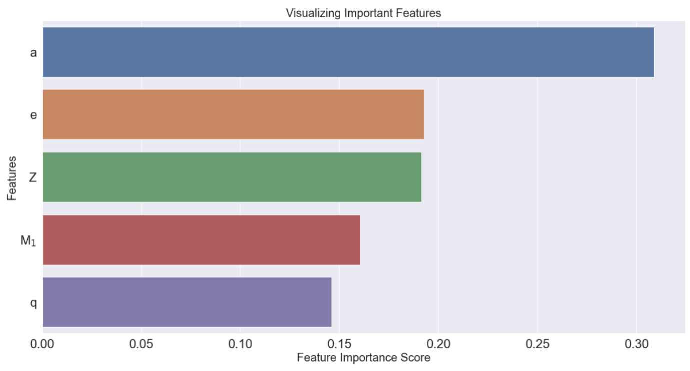

Fig. 4: Histogram of the features relative importance.

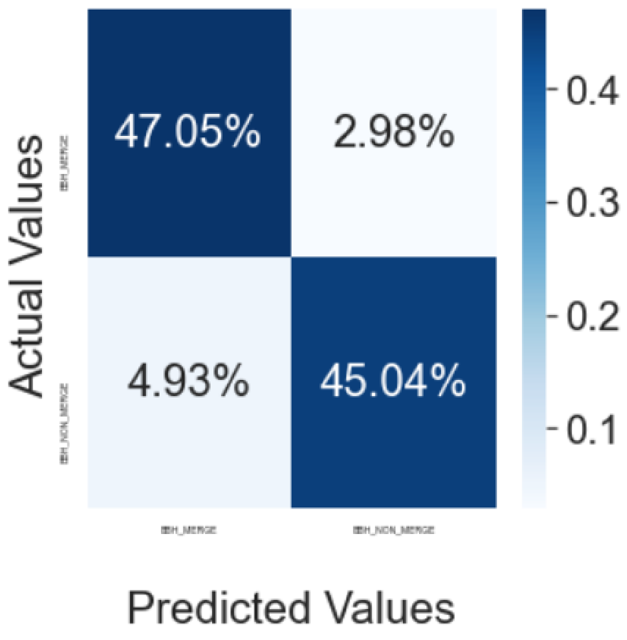

Fig. 5: Confusion matrix for the 2-channel classification.