Machine Learning Hyper-parameter Optimisation using Differential Evolutionary Algorithms

Tianyao Jiang

A thesis submitted for the degree of
Bachelor of Advanced Computing (Honours)
The Australian National University

June 2021

Except where otherwise indicated, this thesis is my own original work.

Tianyao Jiang
8 June 2021

To my parents Feng Jiang and Xiangrong Zhou for their love, courage and dedication.

Acknowledgments

For a long time, I have dreamed of becoming a computer scientist and obtaining the insights into the unsolved problems of the machine learning field. As the deadline of this thesis approaches, I realise that the dream became a reality in this academic year along this research project, and that I have to thank those people who have been hugely supportive.

First, to my supervisor Amanda Barnard, thank you for always guiding and assisting me in this research project when challenges occur, and for reading and advising on the chapters of my thesis. I have learnt so much from you and without your help, my future academic and career plans cannot come true.

Second, to everyone at the College of Engineering and Computer Science. Thank you, Robin, Evan, Justin and Song, for always being happy to discuss insightful ideas and answer my questions, and Harry for being the guider in my first year. Thank you

To my colleagues Haonan, Haiqi, Chujie and Tommy who joining Amanda's group in Semester2 2020. Thank you for sharing the exciting discoveries and new results in weekly group meetings.

To all of my friends in Australia, Naiqian, Bing, Chong, Dafeng, Wenwen, Edward, Alex, Roc, Hai, Gary, Ginobilly, Jun, Harper, Huaidian, Jack, Aozhou, James, Joseph, K, Tad, Nai, Yukang, Max, Wenwen, Neil. Thank all of you for beautiful memory.

To Dai, for always being the listener since high school days. Without your convincement, I may not have the thought to study in Australia as an international student. The epidemic has taken away the time we should have been together, but the good days in Canberra are unforgettable.

To "Dagou" Naisheng, who has meant I have always had a friend in Canberra, for walks, talks, and everything we have experienced during the university years. We overcome all the challenges from the study, life, and "Summoner's Rift" together throughout the past four years, which are fond memory to me. I wish you could fulfill every academic and career plan.

A special thanks to Xinyu. We have accompanied each other for most of the college time, which is also the most precious period of time in both of our life. Thank you for coming into my life and giving me unforgettable memory and "FeiFei" planet, and I wish all the happiness your heart can hold.

Finally, to my family. Thank you for your love and unconditional support over 23 years. Thank you, mom and dad, for always encouraging me to make my dreams in reality.

Abstract

Every machine learning model has hyper-parameters, even non-parametric models. The performance of a machine learning model will be significantly impacted by the setting of its hyper-parameters. One way to tune them is by a manual search, which is time-consuming (Putatunda and Rama [2019]). There are some alternatives, including grid search (Lerman [1980]) and random search (Bergstra and Bengio [2012]), which can perform an automatic search. However, it has been proven that due to the exponential time complexity of the exhaustive search, grid search is too time-consuming to be employed in most cases, and that random search has high variance and no intelligence in selecting the combination of hyper-parameters for the next iteration. A promising alternative is the Differential Evolution: simulating the biological evolution process to search in a designated space. It is proposed that this algorithm will not be significantly impacted by the dimension of a search space and has a guide for selecting the combination of hyper-parameters for the iteration. This thesis develops a novel algorithm backboned by Self-adapting Differential Evolution (SaDE) (Qin and Suganthan [2005]) which is a variant of Differential Evolution (DE) (Storn and Price [1997]) and constructs a python package, EvolutionarySearchCV, with the aim of outperforming two established packages available in sklearn (RandomisedSearchCV and GridSearchCV). In order to improve the performance of the original SaDE further, we add a powerful concept referred to as "attention", which is widely used in the deep learning area. With the exception of the main functioning algorithm SaDE+ embedded in the new package, we also implement and design some functional mechanisms, including multi-processing and early stopping which can improve its time efficiency and practicality. We design collections of tests for our new model to evaluate the performance and compare it with RandomisedSearchCV and GridSearchCV. By performing the tests on two datasets Ag_ordered_dataset and Pt_nanoparticle_dataset, we find that our model can outperform RandomisedSearchCV in the aspect of final accuracy and GridSearchCV in the aspect of time consumption. However, our package is not as computationally efficient as RandomisedSearchCV, particularly for large datasets of high dimension. Thus, in this thesis, we will also evaluate the limitations of our package and propose future approaches for improving it further in terms of time efficiency and the quality of the returned combination of hyper-parameters.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, background knowledge about the machine learning field will be given in Section 1.1. In Section 1.2, we will provide a broad view of the problem that we aim to solve in this thesis and the reason why it is vital in the machine learning area. In Section 1.3, we will give two currently existing solutions to this problem. In Section 1.4, a new approach to solving this problem will be present in Section 1.2, and we will discuss why it may be more suitable to solving hyper-parameter optimisation problems than the former ones.

1.1 A Popular Field: Machine Learning

1.1.1 A Recap of Basic Concepts

Machine Learning (ML) refers to the research of computer algorithms that can automatically improve by accumulating experience from given data (Mitchell et al. [1997]). ML algorithms are usually employed to perform prediction (predicting numerical labels) and classification (predicting categorical labels) tasks. Hence, they are currently popular in a wide range of fields, such as medicine, Computer Vision and Email Filtering since these tasks are usually unsolvable to conventional algorithms (Hu et al. [2020]).

1.1.2 ML Model Parameter and Hyper-parameter

ML models have two types of parameters: the normal ones and hyper-parameters, which are difficult to be discriminated. This subsection will briefly discuss the difference between them as the first taste of machine learning hyper-parameters.

A model parameter is an internal variable that controls the configuration of the model, and there is a good metaphor for it: the machine learning model can be regarded as the hypothesis, while its model parameters are the tailoring of it to a specific dataset (Brownlee [2019]). For instance, the weights of neurons in an artificial neural network and the variables and thresholds used to split each node in Random Forest are model parameters. Moreover, all the model parameters can be learned from the given dataset in the training process.

A model hyper-parameter is an external variable that cannot be estimated from data or learned in the training process. Comparing to parameters, model hyper-parameters have three features: 1. they need to be specified before training. 2. They are set in order to help the model learn the parameters. 3. They need to be tuned according to the given predictive problem (Brownlee [2019]). Unlike model parameters, there is no analytical formula that can automatically return appropriate values for hyper-parameters, and researchers usually tune them by expert experience or some rules of thumb (Kuhn et al. [2013]).

1.2 The General Problem

Every machine learning model, including parametric and non-parametric models, has hyper-parameters. More specifically, parametric models, such as Support Vector Machine (SVM), have the kernel type and regularisation parameter to be tuned, and non-parametric models such as k-Nearest Neighbours (kNN) have the number of neighbours to be specified before training. Therefore, hyper-parameters that can determine the effectiveness of the training process and thus affect the prediction and generalisation ability of a model should be specified before learning, while normal parameters can be learned in the training process.

In the machine learning field, hyper-parameter optimisation or tuning refers to the problem of choosing an optimal combination of hyper-parameters for a machine learning model. In other words, it aims to find the optimal combination of hyper-parameters in the search space, which can minimise the loss function on a given dataset (Claesen and De Moor [2015]). As we can see from the definition, a machine learning model, search space, and dataset are essential components required to perform hyper-parameter optimisation. We can also see that hyper-parameter optimisation is an essential part of building a machine learning model because its performance heavily depends on proper hyper-parameter optimisation (Putatunda and Rama [2019]).

According to the "no free lunch" theorem (Igel [2014]), the amount of computational capacity needed for all optimisation problems is the same, and there is no shortcut. In the hyper-parameter optimisation setting, the search space is dependent on the number of hyper-parameters and possible values that can be passed to them, which is high-dimensional and large. More specifically, the number of hyper-parameters needed to be optimised determines the dimensionality of the search space (if two hyper-parameters are waiting to be tuned, the search space is 2-dimensional), and the number of possible values for each hyper-parameter controls the range of search space. This feature of hyper-parameter optimisation problem causes the curse of dimensionality, which in turn makes the search extremely time-consuming. In practice, researchers usually rely on expert experience to choose and tune a small number of relatively important parameters and narrow down the search space (Yu and Zhu [2020]). However, it is nearly impossible for them to obtain the optimal combination of hyper-parameters relying only on the experience, given a large and high-dimensional search space. This is the reason why we need automatic algorithms to help us complete the search.

1.3 Two Existing Solutions

There are two existing search algorithms that can perform an automatic search for hyper-parameters. In this section, we will briefly introduce three possible automatic solutions to this problem.

1.3.1 Grid Search

Grid search is the most straightforward solution to hyper-parameter tuning, which performs an exhaustive search on all the possible hyper-parameter combinations and chooses the best one to return (Hsu et al. [2003]). By using it, the most accurate combination of hyper-parameter will be returned if the computational resources are sufficient.

Grid Search has two main advantages: 1. it is guaranteed that the optimal combination will be returned if it is in the search space. 2. every possible combination can be executed independently, so this algorithm can run in parallel if there is enough computing capacity.

However, it is only applicable when the total number of hyper-parameters waiting to be tuned and the search space are small due to the curse of dimensionality (Yu and Zhu [2020]). To be specific, when the number of hyper-parameters required to be tuned increases, the time complexity of this algorithm will also exponentially increase since the complexity of Grid Search is $O(n^m)$, where m is the number of hyper-parameters, and n is the number of possible values for the parameter which has the largest number of possible values. Therefore, this algorithm can only be used when the total number of hyper-parameters required to be tuned is less than or equal to 3, or the user can narrow the search space down beforehand (Yu and Zhu [2020]). Although it can only be employed to complete the search task in a small number of cases, it is still the most widely used search algorithm for hyper-parameter optimisation due to its mathematical simplicity (Bergstra and Bengio [2012]).

1.3.2 Random Search

The use of Random Search for hyper-parameter tuning is first proposed in (Bergstra and Bengio [2012]). Random Search will randomly select from all the possible combinations of hyper-parameter instead of exhaustively testing every possible combination.

Hence, Random Search will terminate if the desired accuracy is reached, or the predetermined combinations are all tested. With this feature, Random Search is more suitable for complex optimisation problems given limited computational resources and a good alternative to Grid Search. Comparing to Grid Search, Random Search has been proven to generate better results because of one major benefit: computation budget can be allocated according to the distribution of the search space, while Grid Search can only evenly assign budget to all the combinations. Thus, Random Search is more likely to outperform Grid Search when the distribution of the search space is not uniform. In (Bergstra and Bengio [2012]), Random Search has been proved that it is empirically and theoretically more efficient for parameter optimisation than Grid Search.

However, Random Search also has apparent drawbacks. Firstly, it cannot guarantee that all the search space is covered even though we allocate the same number of computational resources as Grid Search (Barros [2014]). Another drawback is that this algorithm does not employ any intelligence to choose the combination for the next iteration. Some variants of Random Search, such as Fixed Step Size Random Search (FSSRS) (Rastrigin [1963]), Optimum Step Size Random Search (OSSRS) (Schumer and Steiglitz [1968]) and Optimized Relative Step Size Random Search (ORSSRS)(Schrack and Choit [1976]), have improved the sampling process of the original Random Search. Nevertheless, due to the high variance in the selection process, it is suggested that only using Random Search to narrow down the search space before the intensive search, and then use a guided algorithm so as to obtain better results Ng [2017].

1.4 A New Hope: Evolutionary Algorithms

Due to Grid Search's huge computational demand, this algorithm is apparently unsuitable for complex ML models such as deep neural networks (DNN) and random forest with a large number of hyper-parameters. Moreover, even though the Random Search can significantly reduce the computational time, the selection of parameters for each iteration is completely random, leading to high variance. Hence, we need an algorithm that can achieve comparative performance as Random Search and also has a direction in the selection process.

The Evolutionary Algorithm (EA) was first introduced in 1954 (Barricelli et al. [1954]), and it is a metaheuristic optimisation algorithm that relies on a heuristic to perform comparisons between candidate solutions and therefore hopefully finds the best one among them.

The EA is inspired by Darwin's Evolution Theory and Natural Selection (Mitchell [1998]), which usually consists of three main processes mutation, crossover, and selection (Bäck et al. [1997]). Broadly speaking, individuals of the population in an evolutionary algorithm are the candidate solutions, and there is a fitness function that can evaluate each individual and determine whether it can survive to the next generation or not. Moreover, each individual will perform crossover and mutation before being selected according to its performance on the fitness function. As we can see, the fitness function is the heuristic that plays a vital role in evolutionary algorithms. When using evolutionary algorithms to perform hyper-parameter optimisation, the fitness function would be a user-specified loss function, which can evaluate the overall performance of a machine learning model.

The effectiveness and efficiency of EA have been successfully demonstrated in many applications, including pattern recognization (Storn and Price [1997]), communication (Ilonen et al. [2003]), and mechanical engineering (Storn [1996]). There are two reasons why the evolutionary algorithm is a new hope to this optimisation problem. Firstly, it is a metaheuristic optimisation algorithm. According to (Balamurugan et al. [2015], Bianchi et al. [2009]), a metaheuristic algorithm can a offer sufficiently good solution to an optimisation problem, especially when there are limited computational resources. Therefore, compared to Grid Search, EA is more suitable for some problems with large search spaces. Another reason is that although EA and Random Search are both stochastic optimisation of which the solution depends on the set of random variables generated, the evolution mechanism will provide directions for the searching (Bianchi et al. [2009]).

The evolutionary algorithm has a number of variants, and two of them are widely used in optimisation problems. Genetic Algorithm (GA) is the most popular type of evolutionary algorithm, which was first proposed in (Holland [1992]). Another variant is Differential Evolution (DE), proposed in (Storn and Price [1997]). According to (dos Santos Amorim et al. [2012]), DE has been proven to outperform GA in terms of the final scores in numerical optimisation problems. Moreover, it is shown that DE has a faster convergence speed than GA when approaching the region of reasonable solutions.

1.5 Thesis Outline

The primary purpose of this thesis is to discuss our model, EvolutionarySearchCV, backboned by the novel differential evolution algorithm, SaDE+, which targets at solving hyper-parameter optimisation problems. Broadly speaking, in Chapter 2 and 3, we will mainly focus on the background theory and useful methodology involved in our project, which are also necessary to build and test our new model. Chapter 4 contains the detailed implementation and some helpful designs which can accelerate the package or provide the potential users with convenience. Chapter 5 focuses on the experimental methodology. Chapter 6 then gives the experiment results for the tests outlined in Chapter 5. Chapter 7 will discuss the conclusion for this thesis and propose meaningful plans for future work.

More specifically, in Chapter 2, we will introduce the motivation of our project, two established packages for solving the problem, and the reason why there is an alternative. Firstly, we will discuss why hyper-parameter optimisation is important in the machine learning field. Secondly, two well-known optimisation algorithms: Grid Search and Random Search, which aim to solve this problem, will be recapped. Moreover, an alternative to this problem, which is the DE, will be introduced. In Chapter 3, we will review the powerful variant of the original DE; Self-adaptive Differential Evolution (SaDE) which is the main backbone of our model. We will also discuss the widely used machine learning model, Random Forest, which will be employed for completing our tests of the new model. Furthermore, the Ag_orderer_dataset and Pt_nanoparticle_dataset which will be used in this thesis, will also be discussed in the chapter. In Chapter 4, the implementation of our model, especially SaDE+, and other designs, including Early Stopping and Multi-processing, will be provided. In Chapter 5, the experimental methods for EvolutionarySearchCV will be provided, including 1. tests on different population sizes. 2. tests on multi-processing. 3. tests on the performance of the Early Stopping mechanism. 4. Comparison between established packages and EvolutionarySearchCV in terms of time efficiency and the final accuracy achieved by the returned hyper-parameter combination. In Chapter 6, the experiment results will be shown and explained. Chapter 7 will summarise our thesis and finish it with some remarks for further research, including adaptive population size, more reliable early stopping mechanism, and combining Global Search and Local Search to improve time efficiency further.

Chapter 2

Background and Related Work

In this chapter, the motivation of our research project will be introduced by discussing the importance and complexity of hyper-parameter tuning in Section 2.1 and Section 2.2, respectively. We will also look into two established models: GridSearchCV and RandomisedSearchCV, by discussing how they function and their scaling problems in Section 2.3. The background knowledge about the original DE proposed in (Storn and Price [1997]) will be provided in Section 2.4

2.1 The Importance of Hyper-parameter Optimisation

Even though feature engineering and outlier removal are performed, the hyper-parameters still await to be tuned according to the dataset to assure the acceptable accuracy. In other words, the performance of almost every machine learning method will highly depend on the setting of hyper-parameters (Hutter et al. [2014]).

We consider a support vector classifier (SVC) in the scikit-learn library as an example, which has four hyper-parameters which need to be set before training: 1. *C*: regularisation parameter. 2. *Kernel*: the type of kernels: linear, polynomial, sigmoid or any callable function. 3. *Degree*: a custom degree for the polynomial kernel. 4. *Gamma*: coefficient for polynomial and sigmoid kernel. Tuning any one of these hyper-parameters can improve or reduce the fitting and generalisation ability of a SVC. As we can see from Figure 2.1 and Figure 2.2, the change of kernel type can hugely improve the fitting ability. To be specific, when a linear kernel is employed, as shown in Figure 2.1, the classification result is unsatisfactory. In fact, it is impossible to separate these two classes by a linear function. After changing the kernel to a non-linear one, the classification result is improved. Thus, hyper-parameter optimisation is an essential process in building a machine learning model.

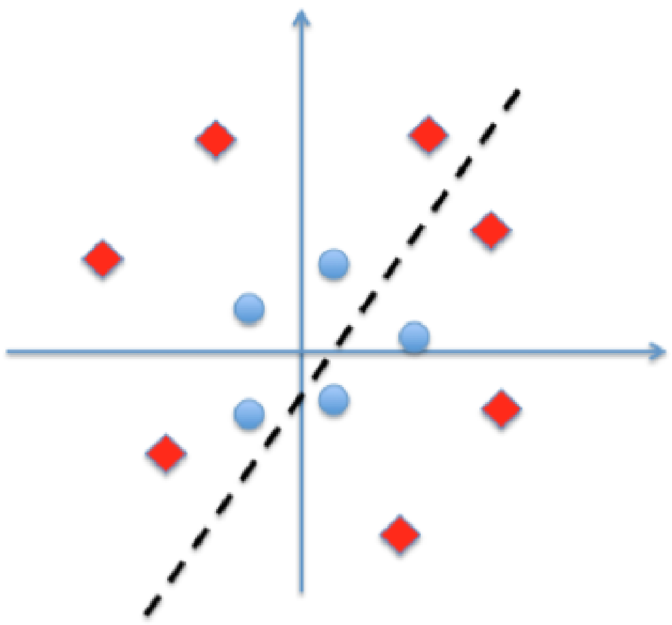


Figure 2.1: The classification result generated by a SVC with a linear kernel (Bishop [2006])

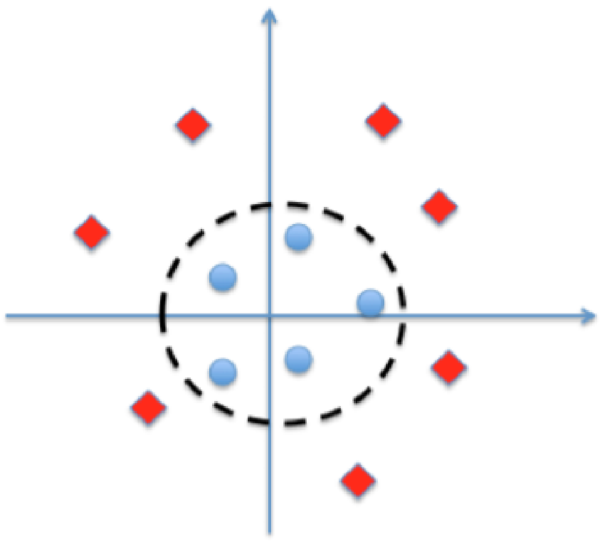


Figure 2.2: The classification result generated by a SVC with a non-linear (polynomial) kernel (Bishop [2006])

2.2 Complexity of Hyper-parameter Optimisation

The complexity of hyper-parameter optimisation is mainly from the range of search space. As discussed in Section 1.2, the search space is determined by the number of hyper-parameters and possible values that can be passed to them. However, it is difficult to quantitatively determine what parameters are important to the final accuracy (Yu and Zhu [2020]). This section will use Artificial Neural (ANN) Network as an example to illustrate the complexity of hyper-parameter optimisation.

2.2.1 Artificial Neural Network (ANN)

The artificial neural network is a powerful machine learning model which simulates animal brains. The first functional neural network was published by Ivakhnenko and Lapa (Ivakhnenko and Lapa [1967]), and then the use of backpropagation (Rumelhart et al. [1986]) enables practical training of multi-layer networks.

2.2.2 Hyper-parameter for ANN

For neural networks, there are a set of hyper-parameters we need to specify before training. Firstly, an optimiser or solver is for weight optimisation. There are some popular choices for this hyper-parameter. Stochastic Gradient Decent (SGD) is the first optimiser, which is proposed in 1951 (Robbins and Monro [1951]). Currently, this optimiser is usually used with momentum (Qian [1999]). There are also some alternatives for SGD, including Broyden–Fletcher–Goldfarb–Shanno (BFGS) (Avriel [2003]), Adaptive Gradient Algorithm (AdaGrad) (Duchi et al. [2011]), Root Mean Square Propagation (RMSprop) (Hinton et al. [2012]) and Adam (Kingma and Ba [2014]).

Secondly, activation functions usually refer to the functions which can introduce non-linear properties to their input, and it is also a hyper-parameter for neural networks. Activation functions are the key to the high accuracy that one neural network can achieve, especially when used in some complex problems such as Natural Language Processing (NLP) and Computer Vision (CV). Without a non-linear activation function employed, a neural network will function as a normal linear regressor or classifier. It is worth mentioning; activation functions should be differentiable since this is the key to performing gradient descent and backpropagation in training. According to (Yu and Zhu [2020]), in practice, some widely used activation functions include: sigmoid, hyperbolic tangent (tanh) (Osborn [1902]), rectified linear units (ReLU) (Nair and Hinton [2010]) and Swish (Prajit et al. [2017]).

The learning rate determines the length of the step for gradient descent, which is a positive scalar (Goodfellow et al. [2016]). In practice, this hyper-parameter is important to the final accuracy of a neural network (Bengio [2012]). More specifically, a small learning rate can lead to a prolonged training process and make the gradient descent hard to get out of a local minima, but a large one may cause the gradient descent to diverge (Buduma and Locascio [2017]). A trade-off is required between the convergence rate and overshooting. Theoretically, there is an infinite number of choices for learning rate. For most cases, it can vary from 0.1 to 0.001.

Furthermore, there are also some hyper-parameters that can control the model design. In neural networks, the model design refers to the structure of a neural network. The most typical hyper-parameters in neural networks, which can control the structure, are the number of hidden layers and the number of hidden neurons in each hidden layer. These two hyper-parameters can determine the function complexity the neural network can model (Hinton et al. [2006]). To be specific, if the number of hidden layers and the number of hidden neurons increase, the complexity of the function it can model will also increase. Suppose the model complexity is too high since it has a significant number of hidden layers and neurons. In that case, it will lead to over-fitting problems, which will affect the generalisability of a model to unseen data. However, tuning this hyper-parameter is also problematic since we can have an infinite number of combinations for the hidden neurons if there is more than one hidden layer. Even though there is only one hidden layer, the possible number of hidden neurons in that layer is infinity. Considering DNNs with greater than three hidden layers, tuning these parameters will become more impossible by expert experience or manual selection.

As we can see, finding a good combination of hyper-parameters for a machine learning model is always a bottleneck for researchers since every model has a number of hyper-parameters to be tuned, which in turn can take a wide range of values. It is impossible to manually traverse all the possible values for all the hyper-parameters when their numbers are significant. Therefore, hyper-parameter optimisation is a time-consuming part of constructing a machine learning model, and this is why having a powerful package to do this tedious task is meaningful.

2.3 Two established models: GridSearchCV and Randomised-SearchCV

In order to obtain optimal combinations of hyper-parameters automatically instead of relying on experience and tuning by hand, people now usually rely on two established packages RandomisedSearchCV and GridSearchCV, which are backboned by Grid search and random search algorithm, respectively. In this section, we will discuss these two search algorithms in detail.

2.3.1 The Meaning of "CV": Cross Validation

Both GridSearchCV and RandomisedSearchCV have "CV" in their package titles, which is the acronym of "cross validation". Cross validation is a model validation technique, which can evaluate the generalisation ability of a newly generated model (Stone

[1974]). In the machine learning field, it is mainly used in supervised learning, which aims to evaluate how accurate a predictive model (a classifier or regressor) can be. To be specific, in prediction or classification problems, the given dataset will firstly be divided into two parts: the training set and the testing set. A machine learning model will be trained on the known data (the training set) and tested on unseen data (the testing set). The aim of employing cross validation is to evaluate the generalisability of a machine learning model and test whether it has over-fitting problems and selection bias (Cawley and Talbot [2010]) since over-fitting models can usually have a relatively good performance on the training set but achieve poor results on the unseen data due to the poor generalisation ability.

In practice, in order to reduce selection bias, multiple rounds of cross validation (also known as K-fold cross validation, where K is the number of rounds) will be performed using mutually different partitions. K-fold cross validation will first separate the given dataset into K parts of equal size if possible. Then $K - 1$ of the groups will be used as training data for the model, and the remaining group will be used as testing data to evaluate its generalisation ability (Bishop [2006]). This procedure will be performed K times to ensure that all the data points have been used as testing data at least once, and the results of K cross-validations will be combined (usually averaged) to output the overall estimate of the model's predictive performance (Seni and Elder [2010]). Figure 2.3 illustrates the 4-fold cross validation, and red blocks indicate the held-out group (testing data) for each round.

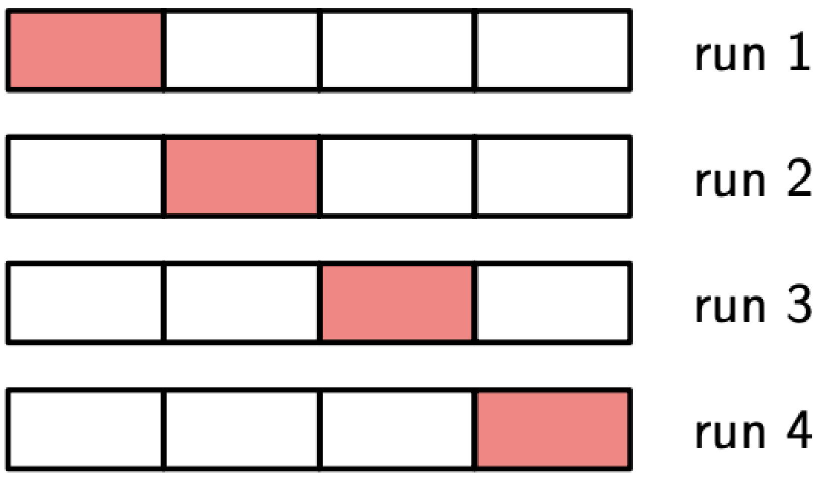


Figure 2.3: 4-fold Cross Validation, where red blocks are the testing set for each run (Bishop [2006])

2.3.2 GridSearchCV

GridSearchCV is backboned by Grid Search, which requires three essential parameters, including *estimator*, *param* and *scoring*, to be specified and a dataset, before execution. The *Estimator* is the machine learning model we choose to use. Moreover, *param* is the hyper-parameter search space, and *scoring* is the metric we use to evaluate the current performance of the machine learning model when using this combination of hyper-parameters. We use a concrete example here to illustrate the way that GridSearchCV works. Given a machine learning model ANN, a 2-dimensional search space: { activation function: Relu, Sigmoid, tanh. Optimiser: Adam, SGD } and a metric: classification accuracy; firstly, GridSearchCV will generate all the possible combinations of the input hyper-parameters as shown below. In this case, there are six combinations in total. Then it will train the ANN using every combination and test it using the metric classification accuracy. Finally, the combination which can achieve the best performance will be returned by GridSearchCV.

$$\{Relu, SGD\}, \{Relu, Adam\}, \{Sigmoid, SGD\}$$
$$\{Sigmoid, Adam\}, \{tanh, SGD\}, \{tanh, Adam\}$$

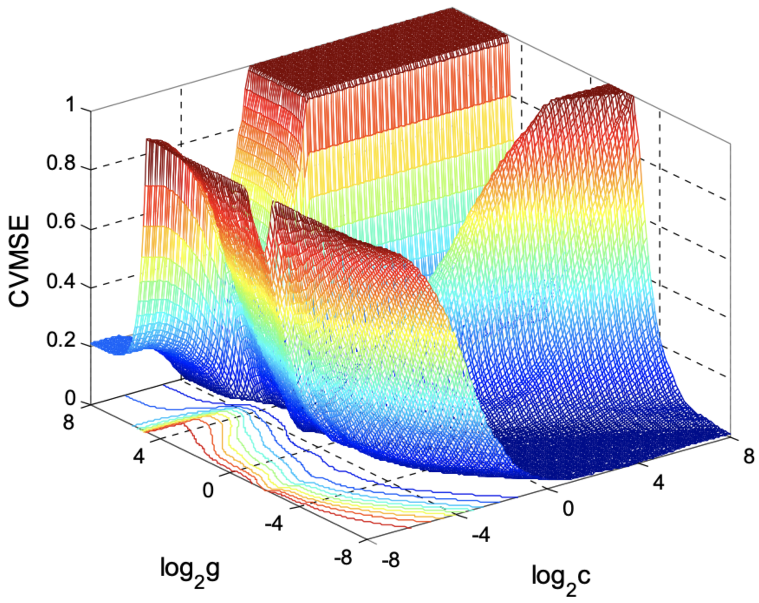


Figure 2.4: CVMSE values for different combinations of hyperparameters. CVMSE is the result of cross validation using the metric MSE. *g* and *c* are two hyper-parameters of SVC (Sun et al. [2021])

As we can see from Figure 2.4, the CVMSE values (calculated by averaging all the MSE values returned by each fold of cross validation) of a large number of hyper-parameter combinations in the parameter grid are high, while only in some small parameter ranges, the values are low. Therefore, Figure 2.4 illustrates that a large amount of computational resources are used by the searching on invalid parameter ranges Sun et al. [2021]).

2.3.3 RandomisedSearchCV

RandomisedSearchCV requires four parameters, including *estimator*, *param*, *scoring* and *n_iter*, and a dataset to be run. As we can see, it has one extra parameter comparing to GridSearchCV, which is *n_iter*. This parameter is the key to the control of time consumption. To be specific, if *n_iter* is set 3, the model will uniformly sample from the parameter grid three times to generate three combinations of hyper-parameters. Then these three combinations of hyper-parameters will be evaluated by the specific metric, and the model will return the best combination with the smallest loss.

By introducing *n_iter* and the mechanism of sampling from the parameter grid, it is guaranteed that RandomisedSearchCV can outperform GridSearchCV in terms of time complexity. However, due to the high variance in the selection process, it is suggested that only using Random Search to firstly narrow down the search space, and then use a guided algorithm so as to obtain better results Ng [2017].

2.4 A Better Choice: Differential Evolution

Differential Evolution (DE) is a type of evolutionary algorithm, and the original DE was introduced in (Storn and Price [1997]), and it is consists of three main processes, including mutation, crossover and selection. Figure 2.5 illustrates the main flow of DE. Moreover, DE has three hyper-parameters, namely NP, F and CR, which will be discussed in detail in Subsection 2.4.1.

2.4.1 Critical Control Parameters

The first control parameter is NP which specifies the population size. Once this parameter is set, the population for the first generation G can also be initialised as below, where $x_{i,G}$ is the i_{th} individual in the population. In the hyper-parameter optimisation setting, each individual (vector) of the current population encodes one hyper-parameter combination.

$$Population = \{x_{i,G}, i = 1, 2, 3...NP\}$$

NP is a hyper-parameter that can be specified beforehand by users, but it cannot be modified in the process of evolution. This is why users should specify NP according

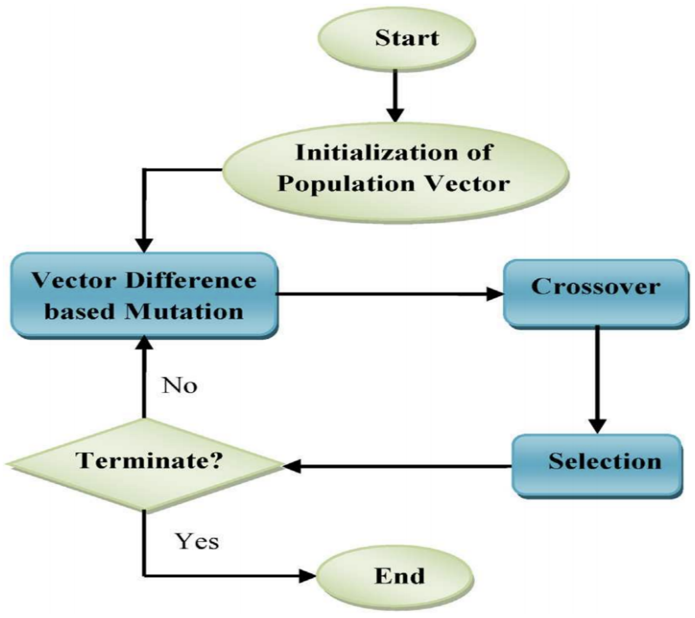

Start
Initialization of Population Vector
Vector Difference based Mutation
Crossover
Selection
Terminate?
No
Yes
End

Figure 2.5: The flow of the differential evolution, including initialisation, mutation, crossover and selection (Kuila and Jana [2014])

to their needs. To be specific, a large population can provide better performance at the end of the algorithm since we can test more combinations of hyper-parameters in each iteration. However, the computation time will also be increased when the size of the population grows. Thus, there is a trade-off between performance and efficiency on NP.

F is the hyper-parameter controlling the importance of differential variation δ, which will be used to generate mutant vectors in mutation (the detailed explanation of mutation will be provided in Subsection 2.4.2). Mutant vector x_{mutant} is obtained by three randomly selected vectors x_p, x_r and x_q as shown below:

$$\delta = x_r - x_q$$
$$x_{mutant} = x_p + F \cdot \delta$$

As we can see, F determines the importance of differential variation δ. Specifically, when F increases, δ has more weight in calculating $v_{i,G+1}$, and vice versa. It is worth mentioning, F is also relating to the convergence speed of the algorithm (Qin and Suganthan [2005]).

The last control parameter is CR which specifies the degree of diversity in crossover (the detailed explanation of mutation will be provided in Subsection 2.4.3). In the original DE, CR is a constant of which the range is $[0, 1]$. This hyper-parameter controls the diversity of our population, which means that it is closely related to problem complexity, and it is the most crucial control parameter in the original DE.

2.4.2 Mutation

According to the original paper of DE (Storn and Price [1997]), after the population is initialised, the mutation will be performed on every individual inside of it. For each individual $x_{i,G}$, in generation G, where $i \in 1, 2, 3 \ldots NP$, we will randomly choose three other individuals namely, $x_{r_1,G}$, $x_{r_2,G}$ and $x_{r_3,G}$ to generate a mutant vector $v_{i,G+1}$ as shown below:

$$v_{i,G+1} = x_{r_1,G} + F \cdot (x_{r_2,G} - x_{r_3,G})$$

As we can see from Figure 2.6, this mutation process will enable us to explore different regions of the search space (Das et al. [2009]), which may also direct the evolution to the local or global minima. This is why F also controls the speed of convergence.

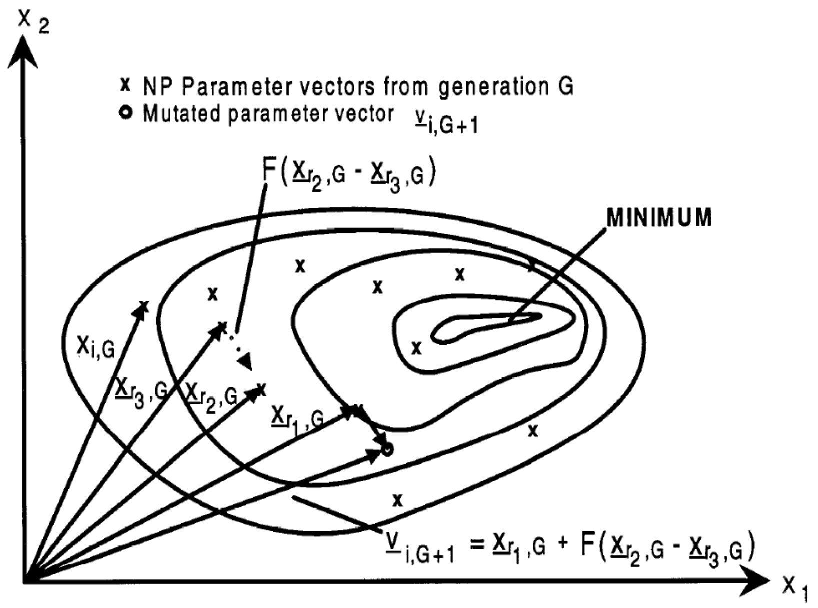

X 2
x NP Parameter vectors from generation G
o Mutated parameter vector v_{i,G+1}
F(x_{r2,G} - x_{r3,G})
MINIMUM
x_{i,G}
x_{r3,G}
x_{r2,G}
x_{r1,G}
v_{i,G+1} = x_{r1,G} + F(x_{r2,G} - x_{r3,G})
X 1

Figure 2.6: Mutation process when using the learning strategy "DE/rand/1/bin" (Storn and Price [1997])

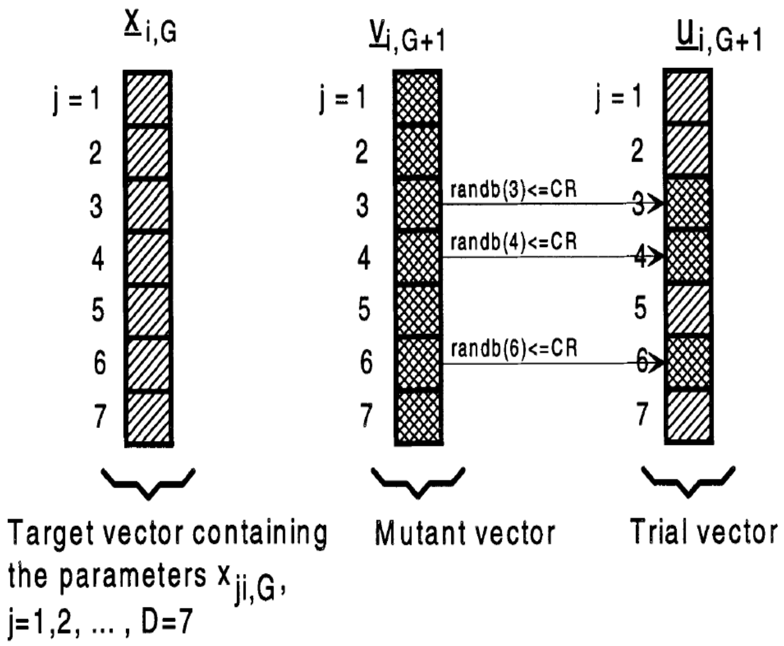


Figure 2.7: Crossover process of DE when using the learning strategy "DE/rand/1/bin" (Storn and Price [1997])

2.4.3 Crossover

Crossover is also an essential component of DE, which can improve the diversity of the population. As we can see from Figure 2.7, in the crossover process, three elements ($x_{i3,G}$, $x_{i4,G}$ and $x_{i6,G}$) of the target vector $x_{i,G}$ are replaced by the elements of the mutant vector $v_{i,G+1}$ of the corresponding positions. To be specific, if $randb(j) \leq CR$ ($randb(j)$ is a uniform random generator of which the output range is $[0, 1]$), where j is the index of the element of mutant vector $v_{i,G+1}$, this element $v_{ij,G+1}$ will be used to replace the element of $x_{i,G}$ in the corresponding position, which is $x_{ij,G}$. Otherwise, $x_{ij,G}$ will be kept. After traversing all the elements of $v_{i,G+1}$, a new trial vector will be generated.

In figure 2.7, the dimension of vectors in the population is seven, which meas that the range of j is from one to seven. In the hyper-parameter optimisation setting, this means that there are seven different hyper-parameters encoded in a single vector.

We can see from Figure 2.7 that CR can control the probability of whether the parameter is replaced by the mutant vector.

2.4.4 Selection

In the selection process, the fitness scores of the trial vector $u_{i,G+1}$ and the target vector $x_{i,G}$ will be compared, using the fitness function. In the hyper-parameter optimisation setting, the fitness function would be one of a collection of metrics that can assess the performance of a machine learning model. To be specific, if the combination of hyper-parameters embedded in target vector $x_{i,G}$ can achieve better performance on the evaluation matrix, then $x_{i,G}$ will survive for the next generation G+1. Otherwise, trial vector $u_{i,G+1}$ will replace the target vector to survive.

2.4.5 Learning Strategies

According to (Storn and Price [1997]), there is not only one scheme for DE, and in order to classify different variants, learning strategies follow this notion: $DE/x/y/z$. Specifically, x is the individual chosen to be mutated, which can be "rand" or "best". If "rand" is used, the individual to be mutated will be randomly selected from the population. Otherwise, the individual with the best fitness score will be chosen when x is "best". In the original paper of DE, "rand" is used. y denotes the number of differential variations used to generate the mutant vector. z is the crossover scheme that will be set to "bin" in most cases.

Except for the original learning strategy $DE/rand/1/bin$ stated in (Storn and Price [1997]), there are many alternatives proposed afterwards. (Das et al. [2009]) points out four widely used learning strategies as shown below, where indices r_1, r_2, r_3, r_4 and r_5 are indices which are randomly chosen from the range $[1, NP]$. These indices are mutually different and also different from the current target vector's index i. $x_{best,G}$ refers to the individual which can achieve the best fitness score in the current population at generation G.

$$DE/best/1/bin : v_{i,G} = x_{best,G} + F \cdot (x_{r_1,G} - x_{r_2,G})$$

$$DE/current\ to\ best/1/bin : v_{i,G} = x_{i,G} + F \cdot (x_{best,G} - x_{i,G}) + F \cdot (x_{r_1,G} - x_{r_2,G})$$

$$DE/best/2/bin : v_(i,G) = x_{best,G} + F \cdot (x_{r_1,G} - x_{r_2,G}) + F \cdot (x_{r_3,G} - x_{r_4,G})$$

$$DE/rand/2/bin : v_{i,G} = x_{r_1,G} + F \cdot (x_{r_2,G} - x_{r_3,G}) + F \cdot (x_{r_4,G} - x_{r_5,G})$$

In this thesis, we will mainly focus on "DE/current to best/1/bin" and "DE/best/2/bin" because the former shows good convergence speed, while the latter can help increase the diversity of the population (Qin and Suganthan [2005]). Therefore, these two strategies can complement each other in the evolution process.

Chapter 3

Methodology

In this chapter, we will introduce the algorithm our thesis mainly relies on, which is Self-adapting Differential Evolution in Section 3.1, and the machine learning model we employ to test the performance of our model in Section 3.2. In Section 3.3, we will introduce the datasets on which we perform the experiments.

3.1 Self-adapting Differential Evolution (SaDE)

As discussed in Chapter 2, DE would be a good choice for solving hyper-parameter optimisation problems, compared to Random Search and Grid Search. Nevertheless, the original DE also has two drawbacks. A number of literatures (Rogalsky et al. [2000], Gämperle et al. [2002]) have pointed out that the hyper-parameters of the original DE, which are required to be specified before searching, including the strategy choice and parameter setting, can hugely affect the performance of it. Even though there are some rules of thumb for choosing proper values for these control parameters and learning strategies (Liu and Lampinen [2002], Price et al. [2006]), this feature of the original DE will introduce extra hyper-parameters to the search, which makes the problem even more complex as researchers can only set them by their experience or by a large number of tests, which is time-consuming.

Self-adapting Differential Evolution (Qin and Suganthan [2005]) provides a solution to this challenge, which allows to adapt learning strategy and control parameters during evolution. The evolution process of this algorithm is similar to the original DE, but it can adapt the learning strategies and two out of three control parameters: CR and F. The remaining control parameter will be a user-specified variable that can control the population complexity according to the complexity of the problem. Let us first introduce these powerful self-adapting mechanisms in SaDE.

3.1.1 Learning Strategy Adaptation

As we discussed in Chapter 2, there are five widely used learning strategies in DE, and each of them has its own advantages. In SaDE, a set of learning strategies will be chosen as candidates and apply one of them to the current population probabilistically (Qin and Suganthan [2005]). The proposed algorithm (Qin and Suganthan [2005]) chooses two different strategies, "rand/1/bin" and "current to best/2/bin" as the candidates, and switches from one to the other in different learning periods (a specified number of generations). The reason why these two learning strategies are selected as the candidates is that "rand/1/bin" can improve the diversity of the population to enlarge the searching range while "current to best/2/bin" can improve the convergence speed (Qin and Suganthan [2005]). Therefore, these two strategies can complement each other in the evolution process.

More specifically, according to (Qin and Suganthan [2005]), two probabilities p_1 and p_2 are both initialised as 0.5 first, which means that each strategy has the same probability of being applied. Moreover, a vector of size NP will be generated, and each element of this vector is created according to the uniform distribution of which the range is $[0, 1]$. If the j_{th} element of this vector is larger than p_1, then "current to best/2/bin" will be applied to the j_{th} individual of the current population. Otherwise, another strategy, "rand/1/bin", will be applied (Qin and Suganthan [2005]). After one generation, we will use ns_1 and ns_2 to count and store the number of trial vectors which successfully survive and are also generated by the strategy "rand/1/bin" and "current to best/2/bin", respectively. Furthermore, nf_1 and nf_2 will be used to count and record the numbers of trial vectors which are discarded in the generation before and also employ "rand/l/bin" and "current to best/2/bin" as strategies, respectively (Qin and Suganthan [2005]). After a specified number of iterations (50 in the original paper), p_1 and p_2 will be updated according to these four values:

$$p_1 = \frac{ns_1 \cdot (ns_2 + nf_2)}{ns_2 \cdot (ns_1 + nf_1) + ns_1 \cdot (ns_2 + nf_2)}$$

$$p_2 = 1 - p_1$$

After a specified number of iterations (one learning period), p_1 and p_2 will be updated by the above calculation from (Qin and Suganthan [2005]). p_1 is updated by the value of the success rate of those trial vectors generated by strategy "rand/1/bin" on the sum of itself and the success rate of other trial vectors generated by strategy "current to best/2/bin" during one learning period(Qin and Suganthan [2005]). Therefore, the probabilities of applying "rand/1/bin" and "current to best/2/bin", p_1 and p_2, can be updated during evolution, and the most suitable learning strategy will be applied at different stages of searching by employing this adaptation mechanism.

3.1.2 Control Parameter Adaptation

For the control parameter adaptation mechanism of SaDE, two out of three critical parameters, CR and F, will be adaptively changed in the evolution process instead of passing fixed values to them before searching. For another control parameter of DE, the population size NP still requires to be specified before evolution according to the problem complexity.

According to (Qin and Suganthan [2005]), CR is more related to the problem's property and complexity, while F is more relevant to controlling the convergence speed. Thus, the choice of F has larger flexibility. In the original paper, for different individuals in the current population, F will be sampled from the normal distribution of mean 0.5 and standard deviation 0.3, of which the range $(0, 2]$. For CR, SaDE will accumulate the previous learning experience since choosing a proper CR value may lead to good performance under a large set of learning strategies, but poor performance under any learning strategies may be caused by the wrong choice of CR. Moreover, the range in which the good value for CR may occur is small (Qin and Suganthan [2005]). In the original paper, CR values are initialised for every individual according to the normal distribution with mean CRm and standard deviation 0.1. After each generation, the CR of the individuals which can successfully survive in the generation before will be recorded. Then, after a learning period (a specified number of generations), CRm will be re-calculated according to the recorded list of CR values, and CR values for different individuals will be re-drawn from the new distribution learned from early generations.

By introducing these two adaptation schemes into the original DE, the SaDE does not require users to provide CR, F and learning strategies to run. Instead, it will only need NP which is a hyper-parameter easy to be tuned. This is because higher NP guarantees a higher probability of finding the global minima, which means that users can tune it according to the problem complexity or available computation resources.

3.2 Random Forest

In this section, we will introduce Random Forest since it is the main machine learning algorithm we employ to perform the experiments on our model in Chapter 6.

3.2.1 Overview of the Model

Random Forest is a machine learning model, and it is composed of more than one decision trees, that outputs by averaging or calculating the mode of the predictions of the individual trees (Ho [1995], Ho [1998]). Random Forest is an ensemble learning method, which refers to statistical methods that use multiple learning algorithms to achieve better predictive performance (Opitz and Maclin [1999]). By grouping multiple learning algorithms, it is more likely to achieve better performance than using the same learning algorithm alone (Polikar [2006], Rokach [2010]). In Random Forest, multiple decision trees are grouped.

The reason why we employ this machine learning model is that Random Forest is becoming increasingly popular in different fields such as biology and chemistry

(Marchese Robinson et al. [2017], Strobl et al. [2008]) because it is highly data adaptive. Furthermore, the "grouping property" of Random Forest also enables Random Forest to be widely used for genomic data analysis and bioinformatics research (Chen and Ishwaran [2012]). To be specific, individual decision trees can explain the correlations between the features in a dataset, and Random forest can merge the ideas from the trees to deal with interaction among variables (Chen and Ishwaran [2012]).

3.2.2 Decision Tree Construction

In this subsection, we will briefly introduce the construction of decision trees, which is a recursion process. A decision tree is composed of three main parts: internal nodes, branches and leaf nodes. Firstly, the internal nodes (including the topmost node) represent an evaluation on an attribute of a dataset. The evaluations on internal nodes can have multiple outcomes, and each of them will be represented as a branch to another internal node or leaf node. The leaf nodes represent a categorical class for a classification tree, or a numerical number for a regression tree.

The attributes and the splitting points for each attribute are usually selected by Information Gain and Gini Ratio (Quinlan [1986]).

3.3 An Overview of the Datesets

Training and testing machine learning models with a specific combination of hyper-parameters require datasets. In this thesis, we will employ two datasets Ag_ordered_dataset and Pt_nanoparticle_dataset, and perform our experiment on them to test the model. We will briefly introduce these two datasets and the machine learning problems we devised for them in this section.

3.3.1 Ag_ordered_dataset

Ag_ordered_data is a dataset of silver nanoparticle final configurations, which is constructed for data-driven research. A detailed description of it can be found in (Barnard et al. [2017]).

There are 425 instances and 85 features and labels in this dataset. It only has one categorical label: the nanoparticle shape, so we devise a classification problem for this label. The label "Shape" has 23 unique values, which means that the machine learning task for this dataset will be a difficult 23-dimensional classification problem. As we can see from Figure 3.1, the number of data points belonging to each class is shown, indicating there is a significant imbalance. More specifically, more than 45 percent of instances are of shape "Decahedron", and there is only one instance each belonging to the class "Modified_rhombic_dodecahedro", "Truncated_trapezohedron", "Modified_cube" and "Truncated_hexoctahedron".

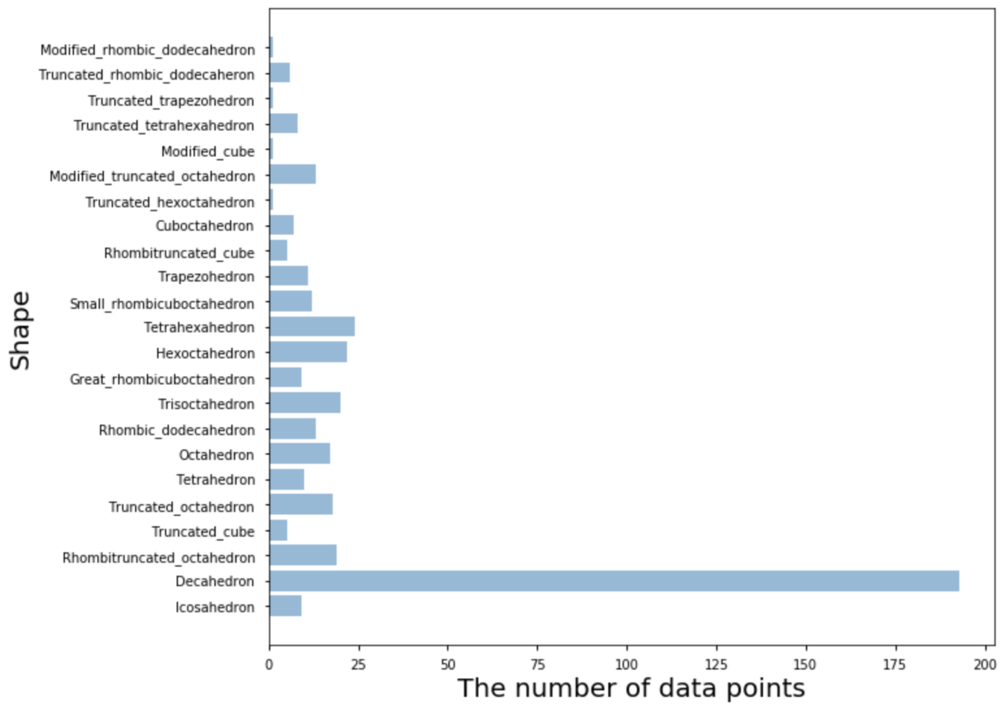


Figure 3.1: The distribution of the label "Shape" of Ag_ordered_data

3.3.2 Pt_nanoparticle_datsaset

Pt_nanoparticle_datsaset is a dataset of platinum nanoparticles, in which each nanoparticle is characterised by topological features, including surface curvature, lattice structure, size and order parameters (Barnard et al. [2018]).

Pt_nanoparticle_datsaset is larger than Ag_ordered_dataset, and it is composed of 1300 instances, 182 features and 5 numerical labels. For this dataset, we devised a regression problem using the label "Surf_defect_mol" which indicates the concentration of Surface Defects (Barnard et al. [2018]). As we can see from Figure 3.2, the range of this label is from 0 to 8.4866, and the label is significantly imbalanced. To be specific, the values of Surf_defect_mol for most of the data points are smaller than or equal to 2, as we can see from Figure 3.4, and there are only 42 data points of which Surf_defect_mol values are larger than 2 (see Figure 3.3).

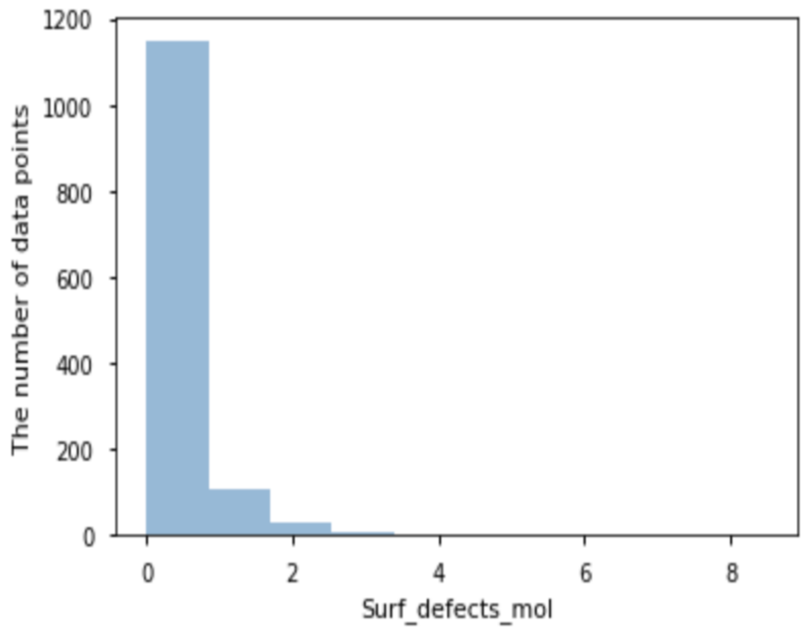


Figure 3.2: The distribution of the label Surf_defect_mol of Pt_nanoparticle_datsaset

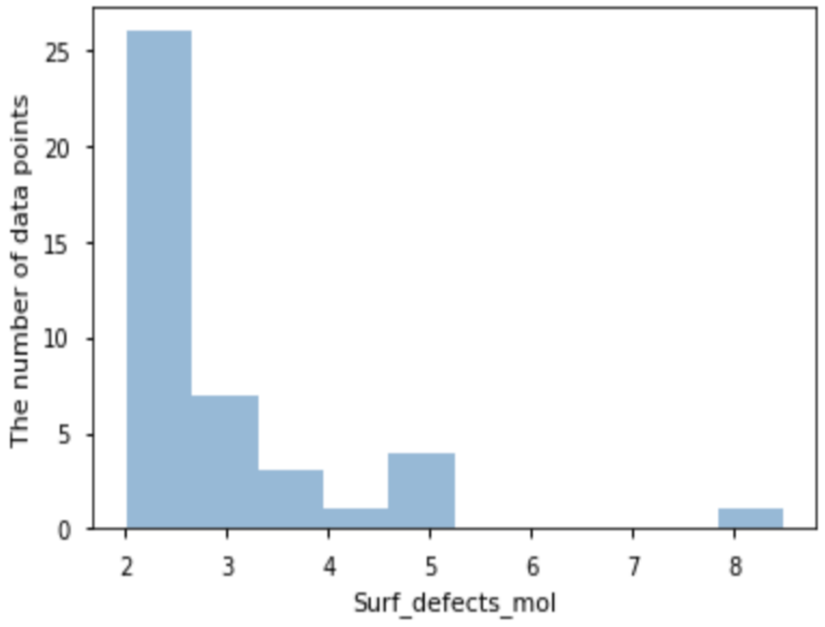


Figure 3.3: The distribution of the label Surf_defects_mol of Pt_nanoparticle_datsaset for the values larger than 2

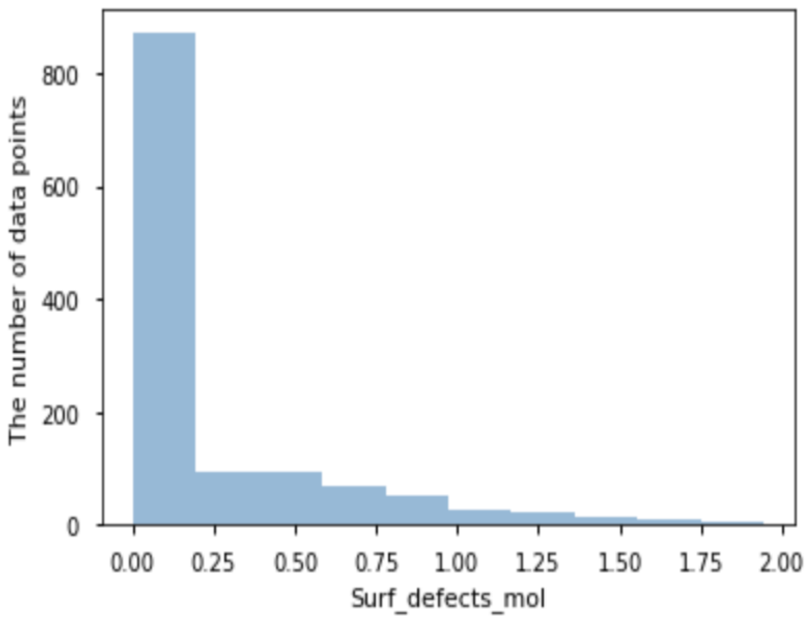


Figure 3.4: The distribution of the label Surf_defects_mol of Pt_nanoparticle_datsaset for the values smaller than or equal to 2

Chapter 4

Design and Implementation

In this chapter, we will discuss our novel differential evolution algorithm SaDE+ and its implementation. Moreover, some novel designs and features in our new package, EvolutionarySearchCV, will also be introduced.

Firstly, we will discuss the main algorithm embedded in EvolutionarySearchCV, SaDE+ which is backboned by the original SaDE proposed by Qin and Suganthan (Qin and Suganthan [2005]). The modification on the original algorithm and the reason why the added simple mechanism can potentially improve the performance of the original algorithm will be discussed. Moreover, Some novel designs for accelerating the package and providing the potential users with convenience, including early stopping, multi-processing and the unlimited choice of metrics, will also be discussed in detail.

4.1 SaDE+

In this section, we will discuss the main algorithm working inside our package, SaDE+, and the improvement of it compared to its "precursor", the original SaDE. Broadly speaking, our SaDE+ follows the original SaDE's initialisation, mutation and crossover processes and makes an improvement on the recalculation of CRm, which will be shown in Subsection 4.1.3.

4.1.1 Unfair Calculation of CRm

We choose to employ the original SaDE (Qin and Suganthan [2005]) as the backbone to construct our new differential evolution algorithm as it can adapt the learning strategy and two of three important control parameters CR and F in the processing of evolution. This self-adapting mechanism can improve the performance comparing to the original DE and provide the users with convenience as they are not required to specify the mutation strategy and two of three key control parameters CR and F, before evolution, which in turn improves the performance. This is because, in the mutation strategy, CR

and F are hyper-parameters to SaDE, and as we discussed in Chapter 2, the performance of a model will be highly dependent on the setting of hyper-parameters. Thus, this intelligent self-adapting mechanism can improve the performance comparing to the original DE.

In the original SaDE, the calculation of the most crucial parameter CR, which can directly affect the performance of the algorithm, is not perfect; or more specifically, not fair. In the original calculation, the CR values for vectors in one generation are random numbers sampled from the normal distribution with mean CRm and standard deviation 0.1. The way of obtaining the mean CRm for one generation is to average the recorded CR values. Furthermore, the criterion for a CR value qualified to be recorded is that the associated trial vector can successfully enter the next generation. This mechanism follows the concept of evolution in which better individuals can survive and hence obtain the rights to mutate and crossover since the individuals which successfully enter the next generation are more likely to have better CR values. Therefore, by recording these values associated with better individuals to calculate the new CRm so as to create a new distribution to be sampled from, this mechanism can accumulate the experience from previous generations, which in turn lets this important parameter adapts in the process of evolution.

However, since CRm is calculated by simply averaging all the recorded CR values, it is not fair to every individual, especially those whose better fitness scores are much higher than others. To be specific, if there is one individual which can achieve obviously higher performance than others, it should contribute more to the new CRm. This is why we employ the most popular concept attention (Bahdanau et al. [2014]) in machine learning, especially the deep learning field and choose to embed it in our novel DE algorithm SaDE+. This attention mechanism will mitigate the problem discussed before.

4.1.2 Attention Mechanism

In this subsection, we will discuss the attention mechanism which is employed in our novel algorithm.

Attention mechanism was first proposed in (Bahdanau et al. [2014]). This has a considerable impact on especially the Neural Language Processing (NLP) area where Recurrent Neural Network (RNN) and Long Short-term Memory (LSTM) are widely used. In this subsection, we will discuss this simple but elegant idea.

In NLP tasks, including machine translation, speech recognition and writing recognition, long term memory of a neural network is required. When we require the neural network to perform machine translation tasks, it needs to memorise the words in between. Even though LSTMs (Hochreiter and Schmidhuber [1997]) use a number of cells to help memorise, the performance is not acceptable.

In these NLP tasks, people usually employ the encoder-decoder mechanism proposed in (Cho et al. [2014]). The encoder will firstly summarise the sentence and generate a vector in the hidden layer, and the decoder will try to recover it. The encoder and decoder here are both RNNs or LSTMs. However, as the length of the input sentence increases, the performance of the encoder-decoder network rapidly degrades (Cho et al. [2014]), and this is why an attention mechanism (as shown below) becomes the new hope for deep learning applications.

This attention mechanism employs a softmax function to calculate and assign a weight which is also the relative importance to each input word, and thus take account of all of them. To be specific, firstly, the scores e_{ij} can be obtained by the function a which is used to measure the alignment between input at j and output at i (Bahdanau et al. [2014]).

$$e_{ij} = a(s_{i-1}, h_j)$$

Secondly, the scores will be input into a softmax function which will turn them into weights.

$$a_{ij} = \frac{exp(e_{ij})}{\sum_{k=1} exp(e_{ik})}$$

Moreover, these weights will be used to calculate the new output c_i of the encoder as shown below.

$$c_i = \sum_{i=1} a_{ij} \times h_j$$

Thus, by using this mechanism, the important words will be assigned relatively high importance, and they are more likely to occur in the next prediction.

4.1.3 Attention Embedded in SaDE+

In SaDE+, we employ the attention concept and improve the calculation of CR to ensure it is fair. In other words, better individuals will have the rights to have more weight on the calculation of CRm.

Firstly, a list called result list is claimed to record all the scores of those individuals which successfully enter the next generation. This is because CR is a control parameter in the process of crossover, and when the new vector can outperform the target vector, it shows that the CR value is acceptable. Moreover, we will use the softmax function (discussed in Subsection 4.1.2) to calculate the weight for each successful individual. As we can see below, w_i is the weight for individual i, and $f(x)$ is the fitness function or metric which can evaluate the individual and return a score.

$$w_i = softmax(f(x_i))$$

Furthermore, we will use the weighted sum of these individuals' CR values to calculate the new mean, CRm, to ensure that it is normalised.

```
if k % learning_period == 0 and k != 0:

    if len(cr_list) != 0 and len(fitness_score_list) != 0:
        weights = softmax(fitness_score_list)
        weighted_cr_list = weights * np.array(cr_list)
        crm = sum(weighted_cr_list)
```

Figure 4.1: The implementation of embedding the attention mechanism into the recalculation of CRm

$$CRm = \sum_{i}^{n} w_i \times CR_i$$

Figure 4.1 shows the implementation of embedding the attention mechanism into the recalculation of CRm. Therefore, by calculating and adding the weights to our calculation of CRm, we can ensure that the calculation will pay more attention to the individuals with better performance.

4.2 Multi-processing

Multi-processing is a powerful concept and technique in the computer science field, which can significantly accelerate the execution of some code that can run in parallel. This is why we choose to embed this design in our package to accelerate the execution of our algorithm. At first, multi-processing refers to using more than two central processing units (CPUs) to compute within a single computer system (Rajagopal [1999], Kettner et al. [2011]). Almost every personal computer has a multi-core CPU, where the multi-core CPU is defined as a computer processor with more than one separate processing unit on a single integrated circuit. These separate processing units are usually called cores, which can read and execute program instructions independently (Rouse [2019]). There are many variations for this basic concept, and the definition of multi-processing may vary in different context. In our implementation, we use the technique that can run multiple processes in parallel, and each process runs on a separate processing unit (core).

4.2.1 Difference between Multi-threading and Multi-processing

These two concepts, multi-threading and multi-processing, have similar purposes, but they are totally different in terms of implementation. In this subsection, we will discuss the difference between them and point out which of them is more suitable to be employed in our package.

As discussed before, multi-processing refers to multiple processes running in parallel since there are multiple cores available. Each process on a separate core will be executed simultaneously and share other resources, such as the clock, memory and other peripheral devices like printers.

However, multi-threading refers to a single process running a collection of code segments concurrently, supported by the operating system (Mukherjee et al. [2002]). In other words, multi-threading aims to increase the use of a single process or a single core, and it is a type of concurrency instead of real parallelism.

Thus, we can see the difference between multi-processing and multi-threading, which is that multi-processing has multiple processes in more than two cores while multi-threading only has one process running.

If comparing multi-threading with multi-processing, the disadvantage of the former is that this technique is not an implementation of real parallelism. The advantage of it is that it is less costly since it does not require an additional processor. In most modern systems and architectures, these two techniques are usually employed together because they are complementary (CPUs with multiple multi-threading cores). Nevertheless, in our model, only multi-processing is employed due to Python's Global Interpreter Lock (GIL) discussed in Subsection 4.2.2.

4.2.2 GIL in Python

Python has a Global Interpreter Lock (GIL), which ensures that there is only one thread holding the control of the Python interpreter. In other words, only one thread can be in the state of execution at a specific point of time if using Python to program. Due to this lock, multi-threading is not applicable in a Python program, which will lead to an adverse impact on efficiency.

Python has this GIL to avoid race conditions between two threads trying to modify a variable in the memory simultaneously. To be specific, Python has a Reference Count Variable for every claimed object, which will count the number of references pointing to the same object. Therefore, when two threads are trying to increase or decrease this variable simultaneously, the race conditions occur due to simultaneous reading and writing operations on the memory.

There are two potential solutions to this problem. The first is that this race condition can be avoided and solved by adding a lock to each object. This method may lead to a more severe problem which creates a 'deadlock'. Deadlock is a condition where every process awaits resources that are held by other processes (Chandy et al. [1983]). As shown in Figure 4.2, resource 2 is required for process 1 to proceed further, which is currently occupied by process 2, while process 2 awaits process 1 to release resource 1. Without obtaining both resource 1 and 2, the processes cannot proceed further, and thus a deadlock occurs, making the whole program freeze.

An alternative method is to claim a global lock, GIL, on the interpreter itself, which is required when any code needs to be in the state of execution. This GIL prevents deadlocks but also requires all the Python code to be single-threaded.

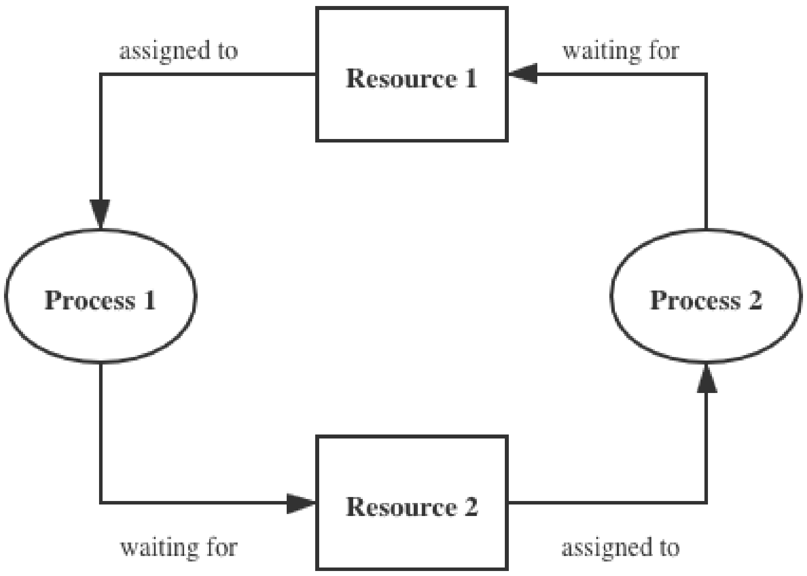


Figure 4.2: A deadlock in which Process 1 requires Resource 2 which is currently held by Process 2, and Resource 1 is held by Process 1, which is essential for Process 2 to run.

4.2.3 Implementation of Multi-processing

In order to side-step the GIL and complete multi-threading, we employ the Python package called Concurrent which is written on the top of another package called multiprocessing. This Python package uses a smart design to side-step the GIL. To be specific, instead of claiming multiple threads in one process, this package will use subprocesses, also called child processes, to simulate multi-threading, and each of them will run in their own Python Interpreter. Therefore, every child process can have its own GIL, which means that the operating system will schedule these child processes on different cores by using this package. Thus, this simulation of multi-threading is more like multi-processing, where multiple child processes can run in parallel.

However, for our algorithm, not every part of it can be executed in parallel. Since in mutation and crossover, the mutant and trial vectors require not only the target vector but also some other individuals, these two processes cannot be in parallel execution. In contrast, in the selection process, we only need the information of the target vector and trial vector, so multi-processing can be applied to this process.

We will show the performance of multi-processing in chapter 6.

4.3 Early Stopping

4.3.1 Why we need Early Stopping

Early stopping usually refers to a technique that can provide regularisation and prevent overfitting in the process of training a model in the machine learning field. Machine learning models such as neural networks which employ an iterative training method (in neural networks, the training method usually employs gradient descent, which is iterative), will have this mechanism. By using this concept, the generated neural networks are less likely to be overly complex so as to mitigate the problem of overfitting.

In our project, early stopping is not for regularisation; instead, it will provide the user an additional choice to stop the algorithm early. By setting the parameter of our EvolutionarySearchCV, the package will stop when the stopping criterion is met instead of when reaching the maximum number of generations.

We make this design since by observing the best individual of each generation, we find that SaDE+ will fall into local minima for several generations. During later generations, the improvement of accuracy is not significant. Thus, we design this feature as a trade-off between performance and time consumption. Undoubtedly, claiming a relatively small value for this parameter will have an adverse impact on the performance of the final model since our search may not converge and find a better combination of hyper-parameters in the next generation. However, by sacrificing the performance, the users can gain extra efficiency as early stopping can reduce the time consumption of executing this package.

4.3.2 Implementation of Early Stopping

The implementation of the early stopping mechanism is not complex, and we mainly rely on a counter to indicate when the algorithm should stop. In our algorithm, this counter is employed to count down when the score of the best individual in this generation is precisely the same as the one of the generation before. For instance, if the counter is set to ten, once there are identical best scores appearing ten times in a row, even though the number of generations is not reached, the algorithm will terminate. Figure 4.3 shows the implementation of the early stopping mechanism in our model.

```
if self.earlystop_counter == 0:
    break

if last_best == best_score:
    if self.earlystop_counter != None:
        self.earlystop_counter = self.earlystop_counter - 1
else:
    self.earlystop_counter = early_stop
last_best = best_score
```

Figure 4.3: The implementation of the early stopping mechanism in SaDE+

4.4 Free to Pass Different Metrics

In this section, we will discuss another interesting and functional design of our package, which is that we also allow users to pass different metrics as a parameter to evaluate the fitness score.

Every machine learning model needs to be evaluated in training, validation and testing processes. Machine learning metrics are the tools that can evaluate and provide a score to the generated machine learning model according to its performance. However, there are a significant number of metrics to evaluate available, and different metrics are suitable for some specific cases or problem settings and may have poor performance in some cases. Therefore, it is vital to make this package suitable for various tasks where different datasets and types of machine learning models are involved, which can provide potential users with convenience and make this package more compatible with various situations and tasks.

Unlike the original DE and SaDE, the fitness function is fixed and defined as a function which takes the individual vector as the input and outputs a numerical variable as the fitness score, our package accepts any callable functions or metrics as the fitness function. Therefore, users can specify and pass a machine learning evaluation metric as a parameter to our package. The implementation of this feature is simple. We do not specify a fixed fitness function in our package and instead, replace it as a parameter to which users can pass callable functions. This is a simple design, but it is important and essential to a package which will be used to tune hyper-parameters for machine learning models.

Chapter 5

Experimental Methodology

In this chapter, we will discuss how we measure and evaluate the performance of the package EvolutionarySearchCV backboned by our novel algorithm SaDE+. There are four main parts of the experiment: 1. Tests on the population of different sizes. 2. Tests on multi-processing. 3. Tests on early stopping. 4. Performance comparison between EvolutionarySearchCV and established packages (GridSearchCV and RandomisedSearchCV). The actual experimental results will be provided and evaluated in the next chapter.

5.1 Hardware Platform

Table 5.2 shows the hardware platform of our testing machine.

Memory	CPU	Operating System
16 GB DDR4 2400MHz	Intel Core i7 6-core 2.2 GHz	MacOS

Table 5.1: The description of the hardware platform of our testing machine

5.2 Metrics

Machine learning metrics are used to evaluate the performance of a trained classifier or predictor. There are a large number of literatures about metrics, such as Receiver Operating Characteristic Curve (ROC), Area Under the Curve (AUC) and Matthews Correlation Coefficient (MCC) (Fawcett [2006], Piryonesi and El-Diraby [2020], Powers [2020], Sammut and Webb [2011], Chicco and Jurman [2020], Chicco et al. [2021], Tharwat [2020]), and different metrics are suitable for different cases or problem settings. For example, in medical and biology-related fields, the metrics such as Sensitivity and Specificity originally introduced in (Yerushalmy [1947]) are preferred, while Precision and Recall are widely used in the computer science area such as NLP tasks (Wang

Actual/Predicted	False	True
False	TN	FP
True	FN	TP

Table 5.2: The simplest confusion matrix for binary classification

and Li [2019]). In the following, we will discuss some widely used metrics and their advantages and disadvantages.

First of all, we will introduce the confusion matrix which is the basis of a large number of metrics. The confusion matrix is often employed to evaluate binary classifiers but is extensible to multi-classification problems. As we can see from the table below, there are four terms, and each of them means that: (1) true positives (TP): the classifier votes for "yes", and the true label is also "yes". (2) true negatives (TN): the classifier votes for "no", and the true label is "no". (3) false positives (FP): the classifier votes for "yes", but the true label is "no". (4) false negatives (FN): the classifier votes for "no", but the true label is "yes".

Table 5.2 shows the hardware platform of our testing machine.

5.2.1 Classification Accuracy

Classification accuracy is the most straightforward metric that can evaluate the overall performance of classification. It is calculated as shown below, which represents the number of data points which are correctly classified divided by the total number of data points in the given dataset:

$$Classification\ Accuracy = \frac{TN + TP}{TN + TP + FP + FN}$$

However, there are several cases where classification accuracy is not a good indicator for the performance of the classification task. One of those would occur when the dataset is not balanced. For example, in a two-class classification problem, the number of data points with the label "True" is 99, and the number of data points with the label "False" is 1. Even though we have a deterministic classifier that will classify all the unseen data points into class 'True', this classifier can even achieve classification accuracy 99% if we employ classification accuracy to evaluate it. However, this classifier actually does nothing. Therefore, we need more metrics to help us mitigate this problem.

5.2.2 Balanced Accuracy

Balanced accuracy (bACC) is a metric that does not suffer from unbalanced dataset problems. Balanced accuracy normalizes TP and TN predictions by dividing them by the number of positive and negative samples, respectively. This will generate TPR and TNR which can help us evaluate how good is a machine learning model even though it is trained using an unbalanced dataset.

$$TPR = \frac{TP}{P} = \frac{TP}{TP + FN}$$
$$TNR = \frac{TN}{N} = \frac{TN}{TN + FP}$$
$$TPR = 1 - TNR$$

5.2.3 Precision and Recall

Precision and recall are specific machine learning metrics. They are defined as the fraction of relevant instances (TP) among the retrieved instances (TP+FP) and the fraction of relevant instances (TP) which were retrieved (TP+FP), respectively. The calculations of them are shown below:

$$Precision = \frac{TP}{TP + FP}$$
$$Recall = \frac{TP}{TP + FN}$$

These two metrics can shed light on how many data points labeled as "yes" are actually "yes" and what percentage of the total data points which are actually "yes" are correctly labeled as "yes". However, they say nothing about other classes. Thus, they cannot be used to evaluate the overall performance of a machine learning model.

5.2.4 F-measure

Since precision and recall are usually discussed together (fix one and check the other), we can use F-measure to combine the calculations, as shown below.

$$F_\beta = (1 + \beta^2) \times \frac{Precision \times Recall}{(\beta^2 \times Precision) + Recall}$$

F1-Score is the most widely used version of F-score of which the calculation is shown below. When we set β to 1, we can have the F1-score:

$$F_1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

By using this F1-Score, we can combine the measurement of precision and recall. Thus, for application which needs both recall and precision, we can employ F1-score instead.

5.2.5 Metrices for Regression: MSE and MAE

For regression models, we usually use the number of errors to measure their performance. Mean Squared Error (MSE) is generated by averaging squared error between actual and predicted values, which is also the most popular metric to evaluate a regressor. The calculation of MSE is shown below, where N is the total number of predictions, and y_i and $y_{predicted_i}$ are the ith actual and predicted value, respectively.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - y_{predicted_i})^2$$

This metric also has some variants, including RMSE and Foveated MSE (Rimac-Drlje et al. [2010]).

However, there is a drawback of MSE: the outliers will have an increasing impact on the evaluation when the squared calculation is applied to the error. Thus, when there are a significant number of outliers in a dataset, the Mean Absolute Error (MAE) is preferred. The calculation of MAE is shown below, where N is the total number of predictions, and y_i and $y_{predicted_i}$ are the ith actual and predicted value, respectively.

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - y_{predicted_i}|$$

By replacing the squared with the absolute calculation, the effect of outliers will be mitigated.

In this section, we discussed some metrics for evaluating classifiers and predictors. In our experiment, for the smaller dataset Ag_ordered_dataset, we choose to use classification accuracy as the fitness function of our model since this metric can show the overall classification ability of a classifier comparing to precision and recall. For Pt_nanoparticle_dataset, we choose to use MAE as the metric to evaluate the predictor. This is because we can mitigate the effect of outliers since the label "Surf_defect_mol" is significantly unbalanced.

5.3 Tests on the Population of Different Sizes

As we can see from Section 3.1, the population size initialised will impact the performance and efficiency of DE. To be specific, if the size of the population initialised is increased, the performance of DE will be improved due to the reason that the added individuals can help explore more search space than before. In other words, the probability that our algorithm can find the global minima will be improved while the population size is increased.

However, while the population size in the initialisation increases, the time consumption of completing the search will also be increased. This is because the added individuals will require more computation capacity to perform mutation, crossover and selection. Worth mentioning, since in hyper-parameter optimisation problem setting the fitness function employed in the selection process is the accuracy the model can achieve by using the returned combination of hyper-parameters encoded in this vector, the machine learning model needs to be re-trained every time there is a new combination of hyper-parameters to generate a new accuracy. This is also why the selection process is time-consuming.

Therefore, there is always a trade-off between performance and efficiency for the parameter, the population size NP. This is why we design this set of tests to check how the performance and efficiency of our package will be improved or affected, respectively, when we increase the value passing to the parameter.

Firstly, the range of population size is decided, which is from the size of 10 to 100. Hence, there will be 10 tests, and in each of them, we will increase the value passing to the parameter by 10 (Appendix A.1). Furthermore, each test's scores and the time consumption will be recorded in an automatically generated table (Appendix A.2), by which two line plots will be constructed to show the trends of time consumption and scores when the population size increases.

The range of the population size varying is not enormous because when the population size is increased to a large number, it requires a tremendous amount of computing capacity. Due to the computation capacity limit of our test machine, we decide to experiment on this range of population sizes. Moreover, this range of population sizes is enough to show the trend of how the time consumption and the performance will be affected or improved.

Worth mentioning, since this set of tests aims to check the effect of changing population size on our package, we should remain all other related test settings as the same for each test except for this parameter. Therefore, we will briefly introduce other settings here. We choose to use Ag_ordered_dataset on which this set of experiments is performed, and the classification problem aiming to classify the label "Shape" as the machine learning task. This is because this dataset is smaller than Pt_nanoparticle_dataset, which will require less computational power in the process of training the model. Moreover, the machine learning model we use here is Random Forest, and we will use all the cores of our test machine to perform these tests. We will turn off the early stopping to prevent this design from reducing the time consumption and affecting the test results. Furthermore, we will use a relatively large search space (shown in next chapter) since by employing it, we can expect that the population of smaller size will have poor performance and bigger population will have better results, which means that the populations will be discriminated against on the generated graph and make it more informative.

5.4 Tests on Multi-processing

Multi-processing refers to the execution of multiple concurrent processes, and each of them runs on a separate CPU. Since the selection process can be executed in parallel in each generation, we can employ this technique in our EvolutionarySearchCV to improve its efficiency.

Due to the parallelism of the selection process, the efficiency of our package should be increased while the number of cores employed to execute grows. Thus, we design this set of tests to show how this feature can improve the efficiency of our model.

Firstly, the range of the number of cores is specified, which is 1 to 10. Apart from these ten tests, we also add "-1" as a value passed to this parameter. This is because, by design, when the parameter takes "-1" as an input, it will automatically use all the cores that the machine has. Therefore, there are 11 tests in total (test cases are given in Appendix A.3), and the test results will be recorded in an automatically generated table (Appendix A.4). Since our test machine only has six cores, it cannot use more than six cores to run our package. Thus, we wish to see that before the number of cores reaches 6, the time consumption will decrease while the number of cores used increases, and after the number of cores reaches, the remaining tests (the number of cores equals 7, 8, 9, 10, -1) will give us similar results.

Furthermore, since this set of tests is designed to check the capacity of the multi-processing feature embedded in our package, we will remain other conditions as the same for each of the 11 tests. Firstly, the search space is the same as mentioned before. Moreover, the dataset and the machine learning problem are Ag_ordered_dataset and classification problem about label "shape", respectively and the population size is set to 30.

5.5 Tests on Early Stopping

Early stopping is also a technique we embed in our EvolutionarySearchCV package for users to trade-off between time efficiency and performance. Without this early stopping mechanism, our SaDE+ will check whether or not to terminate according to the maximum number of iterations, which is also a hyper-parameter of our package. When users set the value for early stopping, our package may stop early before the maximum number of iterations is reached, which will definitely improve the time efficiency but may also have an adverse impact on the results of our search. This is why we design this set of tests to check this mechanism, and summarise the test results to guide the potential users on how to use this technique and which value is suitable for their needs.

Firstly, the range of the values for this parameter is decided, which is from 2 to 30. In each test, we will increase this value by 2 and one additional test would be *None*, so there are 16 tests in total. This is because if we test by increasing this value one by one, the change of time consumption and performance will not be noticeable. It is worth mentioning, our design of early stopping mechanism will be impacted by the change of populations, which means that the test results are significantly affected by randomicity.

Furthermore, since this set of tests is designed to check the capacity of the early stopping mechanism embedded in our package, we will remain other conditions as the same for each of the 11 tests. Firstly, the search space is the same as mentioned before. Moreover, the dataset and machine learning problem are Pt_nanoparticle_dataset and regression problem about label $Sulf\_defects\_mol$, respectively and the population size is set to 30.

5.6 Performance comparison between packages

Here comes the most important collection of tests aiming to compare our new EvolutionarySearchCV with the two established packages – RandomisedSearchCV and GridSearchCV. This is also the main purpose of our research project, creating a new package that outperforms the established ones.

For the comparison, we mainly focus on two aspects of the performance of packages. Firstly, we will evaluate and compare the time consumption since it is an important indicator of the ability of a package. If a package is time-consuming such as GridSearchCV, people without the help of supercomputers with powerful computational capacities would rarely choose to use it in practice. This is also why Grid Search is not as popular as Random Search in most cases. Another aspect is the accuracy the combination of hyper-parameters found by packages can achieve after it is passed to the machine learning model. As mentioned before, the accuracy is always the primary purpose of tuning hyper-parameters. Thus, the final accuracy it can achieve is also an important criterion to compare different packages.

Since both RandomisedSearchCV and our EvolutionarySearchCV are not deterministic, in other words, the results returned by them will be affected by randomness, we will use different datasets, machine learning problems as well as models and search spaces to experiment with them.

5.6.1 Different Datasets and ML Problems

Firstly, in order to reduce the effect of randomness, we will experiment these packages on different datasets for which there are different classification or regression problems designed. As we discussed in chapter 3 (methodology), the Ag_ordered_dataset and Pt_nonparticle_dataset will be used. For these two datasets, we devised a classification problem aiming at classifying the label *shape* and a regression problem aiming at predicting the label $Surf\_defects\_mol$, respectively.

Since for different datasets, the machine learning models require different hyper-parameter settings to achieve the optimal performance, by performing this set of tests, we can observe the change of performance when these packages working on different datasets.

Secondly, we will also experiment using two different machine learning models – Random Forest and ANN to check the compatibility of our package.

5.6.2 Different Search Spaces

There are 10 different search spaces for this set of tests, and the detailed settings for them are shown in Chapter 6. For each test, the search space is enlarged compared to the one before to ensure that we can observe the change of scores. To be specific, as the search space grows, the packages are more likely to find the quality combination of hyper-parameters since there are more options for the value of one parameter. Moreover, we can also check how the time efficiency of these packages will be affected when search space is growing.

Furthermore, since this set of tests is designed to check the capacity of the early stopping mechanism embedded in our package, we will remain other conditions as the same for each of the10 tests. Firstly, the population size is set to 10, and the early stopping mechanism is turned off to pursue the highest quality of output combinations.

Chapter 6

Experimental Results

In this chapter, we will show the experiment results for the collections of tests outlined in Chapter 5. Moreover, the evaluations of the results will be given. There are four main experiments which are 1. tests for the population of different sizes in Section 6.1. 2. tests for multi-processing in Section 6.2. 3. tests for the early stopping mechanism in Section 6.3. 4. Performance comparison between EvolutionarySearchCV and two established packages, GridSearchCV and RandomisedSearchCV, in Section 6.4.

6.1 Experimental Results for Different Population Sizes

In this section, the experimental results for the tests on the populations of different sizes will be provided. As we discussed in Section 5.3, this set of tests will check the change of performance (score) and time efficiency when the population size is increased, and we will test 10 different population sizes ranging from 10 to 100 (Appendix A.1). For the setting of the search space, we choose to use a large search space here, as shown in Table 6.1.

Max_depth	$Min\_samples\_split$	$Min\_samples\_leaf$	$Max\_feature$
1 - 30	2 - 30	1 - 30	"auto", "sqrt", "log2", "None"

Table 6.1: The search space for the experiment on different population sizes from 10 to 100

Moreover, we use the Ag_ordered_dataset and Random Forest Classifier with the random state set to 1 as the dataset and machine learning model, respectively. The max iteration is set to 50. The detailed setting of these related conditions is shown in Table 6.2.

Firstly, as shown in Figure 6.1, when the population size increases, the time consumption also grows. Specifically, when the population size is 10, the time consumption is 22.05, and this number reaches 161.18, while the population size is increased to 100. This corresponds to the nature of SaDE+ since when the population size grows,

Max Iteration	Dataset	ML model	*Early_stopping*
50	Ag_ordered_dataset	Random Forest	None

Table 6.2: The setting of other parameters of EvolutionarySearchCV for the experiment on different population size from 10 to 100

more calculations for the increased number of individuals are required. To be specific, in each iteration, every individual needs to perform mutation, crossover and selection, and all of them require additional computational capacity to be complete, especially the selection process in which a new Random Forest Classifier will be trained, and it will output a final accuracy achieved by the hyper-parameter combination encoded in this individual as the fitness score. Thus, the increase in population size will have an adverse impact on the time efficiency of EvolutionarySearchCV.

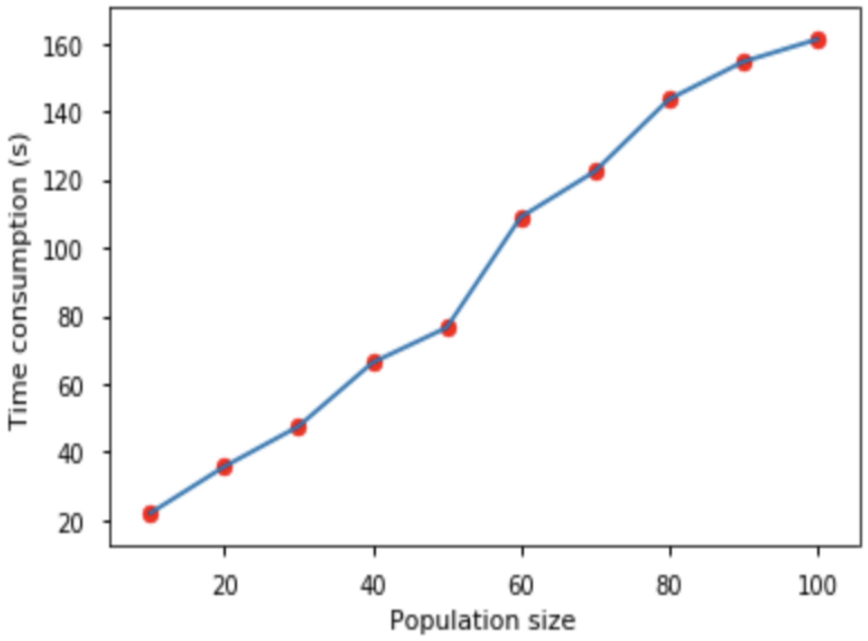


Figure 6.1: The change of time consumption of EvolutionarySearchCV when the population size is increased from 10 to 100

Secondly, Figure 6.2 below shows the change of the final accuracy a random forest classifier can achieve by using the combination of hyper-parameters returned by EvolutionarySeachCV when the population size increases. As we can see, the accuracy change is not significant, and most of them are precisely 0.7209.

Typically, the population size should be tuned according to the complexity of targeting problems. Specifically, when aiming to solve complex problems where there is

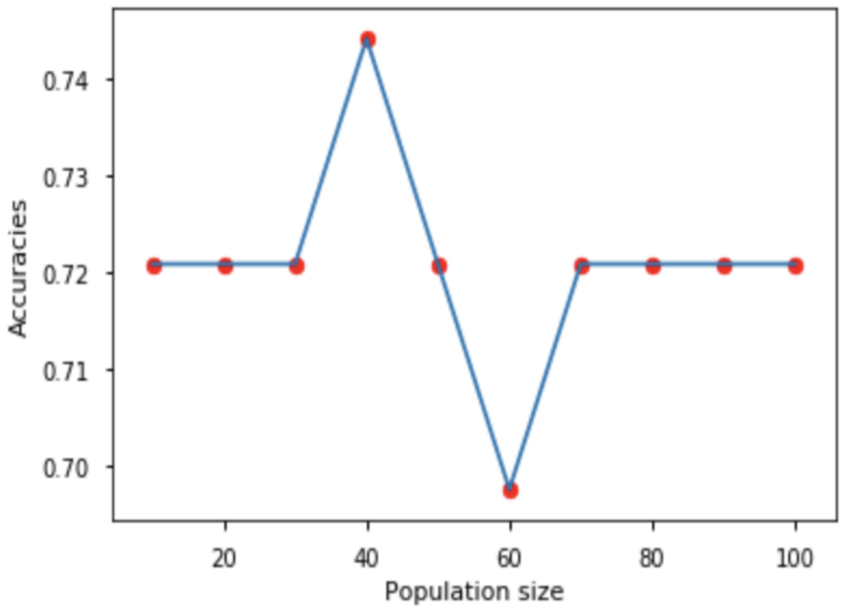


Figure 6.2: The change of final accuracy achieved by the combination of hyper-parameters returned by EvolutionarySearchCV when the population size is increased from 10 to 100

a large dataset and search space, a large population is required since the large number of individuals can discover more areas in the search space so as to find more quality combinations. However, in this set of tests, the problem complexity is not changed as the search space and dataset are not changed, so the increase in population size will not improve the performance.

It is worth mentioning, as we can see, in the test in which the population size is set to 40 and 60, the accuracies are changed. This is attributed to the fact that our EvolutionarySearchCV is not deterministic, and the final result returned by it will be affected by randomness. Hence, the final accuracies will fluctuate a bit.

To sum up, since our SaDE+ still has a hyper-parameter, the population size NP, there should be a trade-off for the value passed to this parameter. If users need an efficient search and do not require a high-quality hyper-parameter combination, they can choose to initialise a small population. In contrast, if users require high performance and to find the nearly optimal combination of hyper-parameters, they should pass a relatively large value for this parameter. Moreover, due to the limit of problem complexity, the change of accuracies when the population size increases is not reflected in this set of tests.

6.2 Experimental Results for the Multi-processing

In this section, the experimental results for Multi-processing will be provided. For this experiment, we mainly focus on the relationship between the number of cores we use and the time consumption of our algorithm which can indicate whether our design of parallel selection process is successful or not. More specifically, this collection of tests can prove whether or not the parallelism of the selection process can improve the efficiency of our model.

For the testing range of the number of cores, we will specify 11 different numbers from 1 to 10 and "-1", as test cases (Appendix A.3). "-1" in our package means using all the cores the machine has. Therefore, theoretically, the last six tests (6, 7, 8, 9, 10, None) should generate similar results due to the reason that our testing machine only has six cores, as described in Section 5.1. For the search space setting, we choose to use a small search space here, as shown in Table 6.3 due to computation capacity limits.

Max_depth	*Min_samples_split*	*Min_samples_leaf*	*Max_feature*
1 - 10	2 - 10	1 - 10	"auto", "sqrt", "log2", "None"

Table 6.3: The search space (parameter grid) for the experiment on multiprocessing, where the number of processes is increased from 1 to "-1" (maximum)

The setting of other conditions is given as shown in Table 6.4.

Max Iteration	Dataset	ML model	*Early_stopping*	Population Size
50	Ag_ordered_dataset	Random Forest	None	30

Table 6.4: The setting of other parameters of EvolutionarySearchCV for the experiment on multiprocessing, where the number of processes is increased from 1 to "-1" (maximum)

As we can see from Figure 6.3, the time consumption decreases as the number of cores used increases, following our prediction stated in Chapter 5. To be specific, before the number of processes reaches 6 which is the maximum number of cores available on our testing machine, the time consumption will drop while the number of processes increases. Furthermore, after it reaches the maximum number of cores our machine has, the time consumption remains almost the same since, in all of these tests, there are exactly six cores employed, which means that theoretically, they are duplicates. It is worth mentioning; there are some fluctuations between 6 and the maximum. This is attributed to the overheads of our algorithm since not every calculation in the mutation, crossover and selection processes takes the same amount of time.

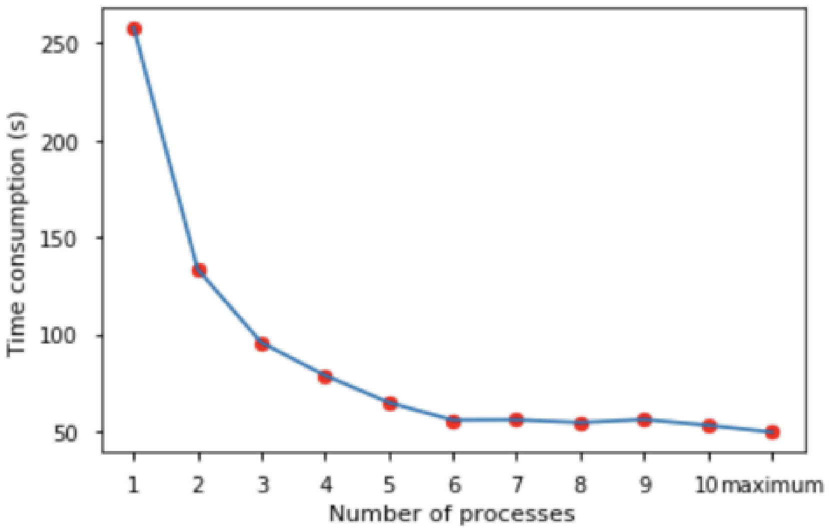


Figure 6.3: The change of time consumption of EvolutionarySearchCV when the number of processes is increased from 1 to "-1"

To sum up, this set of tests indicates that our implementation of multi-processing is successful since the time efficiency is improved, while more cores are being employed.

6.3 Experimental Results for Early Stopping

In this section, we will show the experimental results for Early Stopping. As discussed in Chapter 5, we design 16 tests in total, including 15 tests generated by increasing the value passed to the parameter, *early_stop*, by 2 in each of them from 2 to 30 and one additional test in which *None* is passed (Appendix A.6). In our implementation, when *None* is passed, the early stop mechanism will be turned off. This experiment mainly focuses on how the time consumption will be reduced while the early stopping increases and whether or not this technique will have an adverse impact on the final scores.

For the search space, we use a large search space here, as shown in Table 6.5, and the setting of other conditions is provided as shown in Table 6.6.

Max_depth	*Min_samples_split*	*Min_samples_leaf*	*Max_feature*	*Max_leaf_nodes*
1 - 30	2 - 30	1 - 30	"auto", "sqrt", "log2", "None"	2 - 30

Table 6.5: The search space (parameter grid) for the experiment on Early stopping when the parameter *early_stop* is increased from 2 to 30

Max Iteration	Dataset	ML model	Population Size
50	Pt_nanoparticle_dataset	Random Forest	30

Table 6.6: The setting of other parameters of EvolutionarySearchCV for the experiment on Early stopping when the parameter *early_stop* is increased from 2 to 30

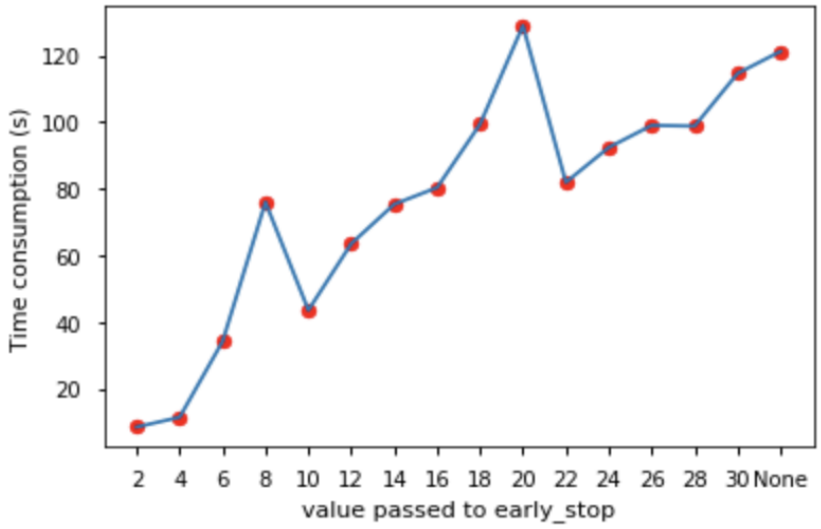


Figure 6.4: The change of time consumption of EvolutionarySearchCV when the value passed to the parameter *early_stop* is increased from 2 to 30

As we can see from Figure 6.4, the amount of time consumption increases while the value passed to *early_stop* grows. When the early stopping mechanism is turned off, which is the last test, the time consumption is 121.18 seconds. As we can see, except for the test where is parameter is set to 20, the early stopping mechanism is triggered in every remaining test and contributes to the time efficiency. Moreover, this also indicates that our early stopping mechanism is affected by randomness since 20 is not the largest number in this set of tests, and higher numbers such as 26, 28 and 30, are more likely to side step this mechanism.

Secondly, Figure 6.5 aims to check whether this early stopping mechanism will have an adverse impact on the final quality of the returned combination of hyper-parameters. As we can see from Figure 6.5, the overall performance is not affected by this mechanism since the final regression scores are all around 0.675. It is attributed to the fact that for this set of tests, the population size 30 is too large for this problem. In other words, the problem is not complex, and when the population size is initialised as 30, the SaDE+ plus can find the local minima immediately. Therefore, even though the *early_stop* is set to 2, which means that the algorithm will terminate in a very early

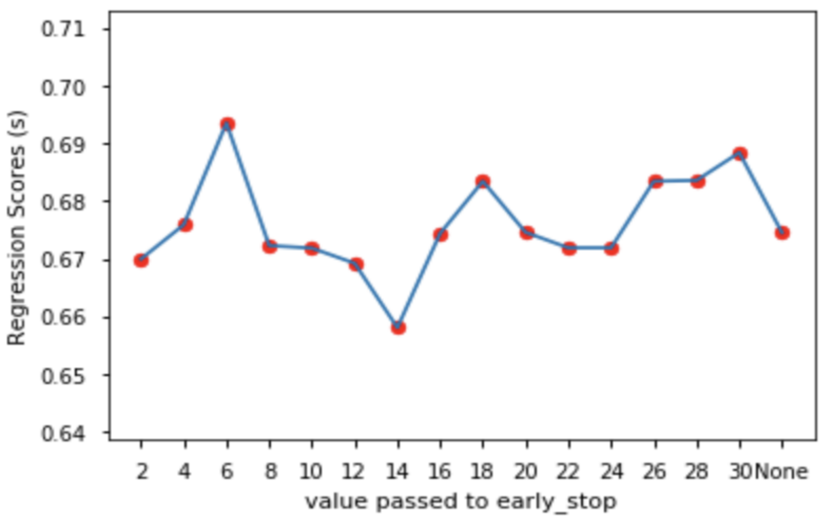


Figure 6.5: The change of the final accuracy achieved by the combination of hyper-parameters returned by EvolutionarySearchCV when the parameter *early_stop* is increased from 2 to 30

stage, the returned result is acceptable.

6.4 Comparison between EvolutionarySearchCV and Established packages

For this set of tests, as outlined in Chapter 5, we will use 10 different search spaces (Appendix A.2), with increasing ranges. It is worth mentioning, due to the limit of computational capacity, GridSearchCV cannot be run on all of the 10 search spaces. Here we use three slowly increasing search spaces to test GridSearchCV so as to show scaling of the time consumption:

As we can see from Figure 6.6, the time consumption dramatically grows as the search space expands. When the search space is set as shown below, the search space 3, the time consumption reaches 2044.8 seconds. However, the search space 3 shown below in this set of tests, is even smaller than the test case 2 in our experiment plan for the comparison (Appendix A.2). Therefore, in this section, we will show the comparison between our model and RandomisedSearchCV in the following experiment.

$$'max\_depth' : [1, 2, 3, 4, 5]$$

$$'min\_samples\_split' : [2, 3, 4, 5]$$

$$'max\_leaf\_nodes' : [2, 3, 4, 5]$$

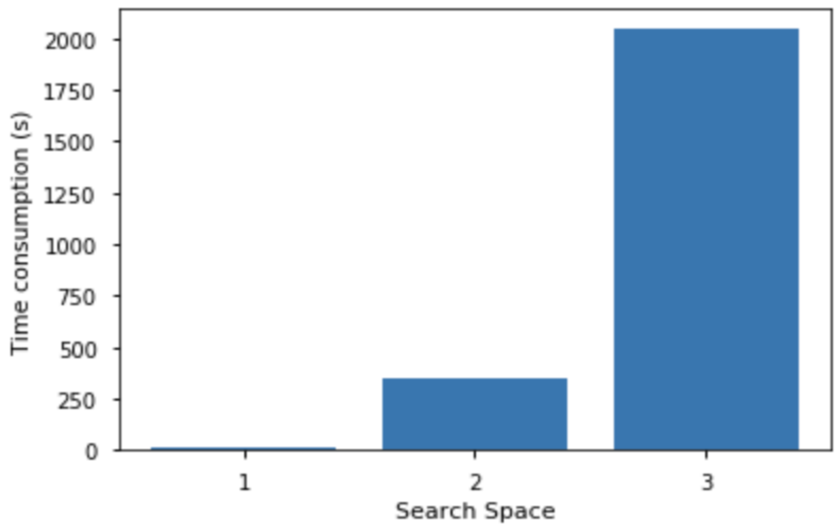


Figure 6.6: The change of time consumption when experimenting on 5-dimensional search spaces with growing sizes

Max Iteration	Dataset	ML model	*Early_stop*
50	Ag_ordered_dataset	Random Forest	None

Table 6.7: The setting of other parameters of EvolutionarySearchCV for the comparison experiment on Ag_ordered_dataset

$$'min\_samples\_leaf' : [1, 2, 3, 4, 5]$$

$$'max\_features' : ('auto', 'sqrt', 'log2', None)$$

Firstly, the experiment will be performed on Ag_ordered_dataset. The setting of other conditions is shown in Table 6.7.

As we can see from Figure 6.7, RandomisedSearchCV is less time-consuming than our package, even though the smallest population size is set to 10. For both of them, the enlarged search space has a negligible effect on the time consumption. Moreover, while RandomisedSearchCV is more efficient, the time consumption of Evolutionary-SearchCV is acceptable since it does not change when the search space is enlarged.

In Figure 6.8, we can see that in most test cases, the final accuracies our model can achieve are higher than the ones of RandomisedSearchCV, which means that in terms of the quality of the combination of hyper-parameters returned, Evolutionary-SearchCV outperforms RandomisedSearchCV as well as being orders of magnitude faster than GridSearchCV.

Secondly, we will experiment on Pt_nanoparticle_datset. The new set of search

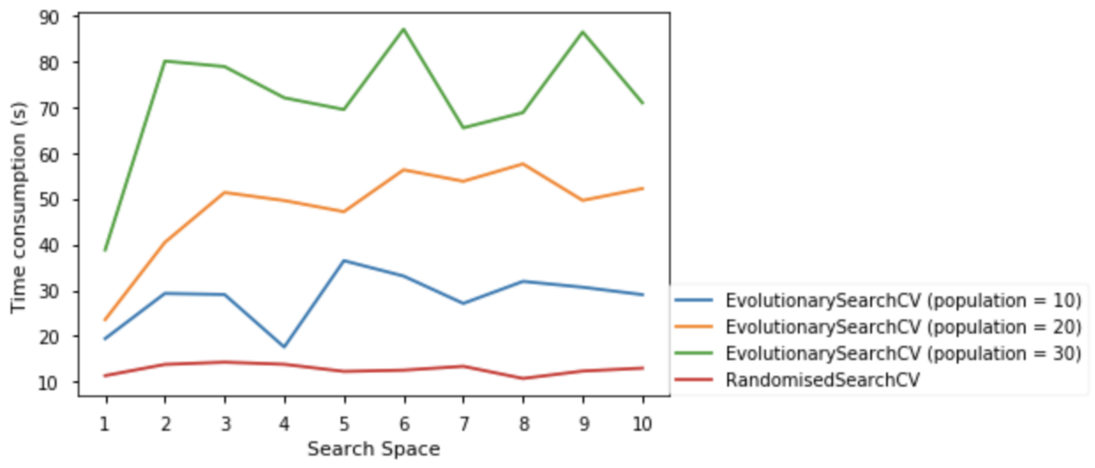


Figure 6.7: The comparison of time efficiency between EvolutionarySearchCV and RandomisedSearchCV on Ag_ordered_dataset. The green, yellow and blue lines are the change of time consumption of EvolutionarySearch when the population size is set to 10, 20 and 30, respectively. The red line is the change of time consumption of RandomisedSearchCV

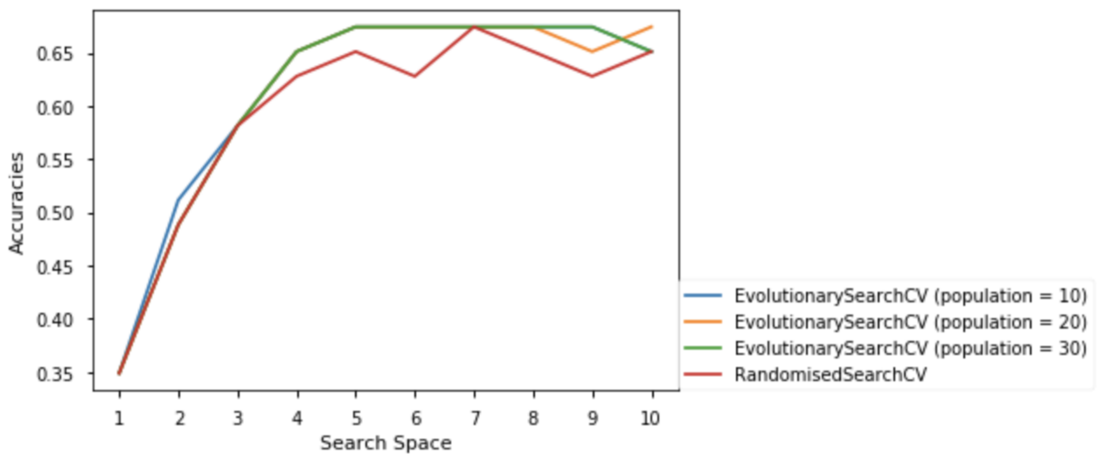


Figure 6.8: The comparison of the final accuracy between EvolutionarySearchCV and RandomisedSearchCV on Ag_ordered_dataset. The green, yellow and blue lines are the change of final accuracy achieved by the hyper-parameter combination returned by when the its population size is set to 10, 20 and 30, respectively. The red line is the change of the final accuracy achieved by the hyper-parameter combination returned by RandomisedSearchCV

Max Iteration	Dataset	ML model	*Early_stop*
50	Pt_nanoparticle_dataset	Random Forest	None

Table 6.8: The description of the hardware platform of our testing machine

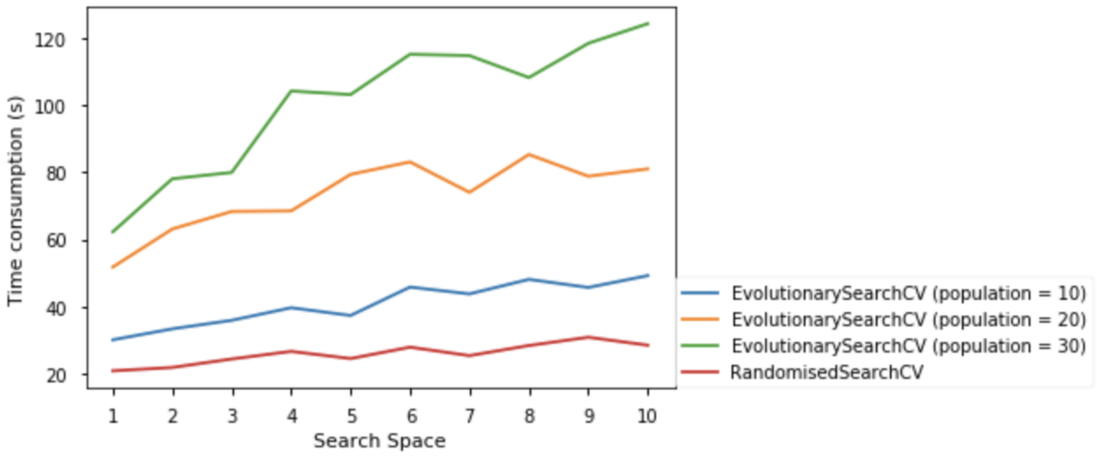


Figure 6.9: The comparison of time efficiency between EvolutionarySearchCV and RandomisedSearchCV on Pt_nanoparticle_dataset. The green, yellow and blue lines are the change of time consumption of EvolutionarySearch when the population size is set to 10, 20 and 30, respectively. The red line is the change of time consumption of RandomisedSearchCV

spaces can be found in appendix. The setting of other conditions is shown in Table A.26.

As we can see from Figure 6.9, the results are similar to the ones for Ag_ordered_dataset. To be specific, RandomisedSearchCV is less time-consuming than our package, even though the smallest population size is set to 10. It is worth mentioning since Pt_nanoparticle_dataset is larger than Ag_oedered_dataset, the amounts of time consumption of both of these two packages are increased, comparing to the experiment on Ag_oedered_dataset. However, the time consumption is still acceptable since the time consumption will not change while the search space is enlarged.

As we can see from Figure 6.10, for the final scores that the machine learning model can achieve, in most test cases, EvolutionarySearchCV outperforms RandomisedSearchCV.

To sum up, in this section, we perform the comparison mainly between EvolutionarySearchCV and RandomisedSearchCV in terms of time efficiency and the final accuracy achieved by the returned hyper-parameter combination. For time efficiency, RandomisedSearchCV is less time-consuming than EvolutionarySearchCV in every

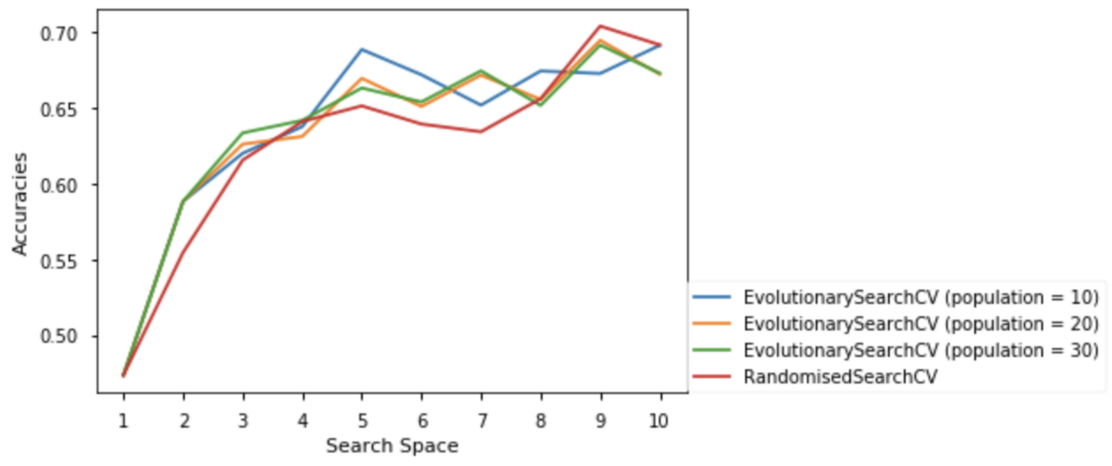


Figure 6.10: The comparison of the final accuracy between EvolutionarySearchCV and RandomisedSearchCV on Pt_nanoparticle_dataset. The green, yellow and blue lines are the change of final accuracy achieved by the hyper-parameter combination returned by when the its population size is set to 10, 20 and 30, respectively. The red line is the change of the final accuracy achieved by the hyper-parameter combination returned by RandomisedSearchCV

test case. However, according to our experiment, the time consumption of our model is only related to the population size and the size of the dataset, and the increase in the number of hyper-parameters required to be tuned or the size of the search space has a minor effect on the total amount of time consumption. Therefore, it illustrates that our model will be applicable in a real-world deployment. For the final accuracy, Figure 6.8 shows that EvolutionarySearchCV can outperform RandomisedSearchCV in most test cases.

Chapter 7

Conclusion

In this thesis, we designed and implemented a new Python package Evolutionary-SearchCV, aiming to perform proper hyper-parameter optimisation. Using the SaDE Qin and Suganthan [2005] as the backbone, we proposed a new differential evolution algorithm SaDE+. Moreover, we completed preliminary evaluations of the performance of the model via a set of experiments discussed in Chapter 5.

More specifically, in Chapter 2, we reviewed the importance of hyper-parameter tuning for machine learning models and some possible approaches to solving this problem. The advantages of evolutionary algorithms are also introduced. In Chapter 3, we discussed the original SaDE algorithm in detail and the machine learning models employed in our experiment. Moreover, we also briefly introduced the datasets used in this project. In Chapter 4, we then discussed the detailed implementation of our novel algorithm SaDE+ and some functional designs embedded in the final package. In Chapter 5, we demonstrated the procedure by which our package was experimented. We evaluated the package and its practical mechanisms in the aspects of time efficiency and overall scores. In Chapter 6, we provided the experiment results for the test cases introduced in Chapter 5.

7.1 Limitations and Future Work

We finish this thesis with some remarks on future research. In this section, let us summarise and discuss the limitations of our EvolutionarySearchCV and the results presented in this thesis. Then, we will provide possible approaches to these challenges and some future work for this project.

7.1.1 Population Size Adaptation

Our SaDE+ is backboned by the SaDE which can adapt two of three control parameters: CR and F, and learning strategy in the evolution, but the population size NP is kept fixed through the entire evolution procedure and left as a hyper-parameter which requires to be specified beforehand. Since the population size relates to the problem complexity that is difficult to be qualitatively measured, fine-tuning this hyper-parameter is time-consuming.

A possible improvement is to employ the population size adaptation mechanism proposed in (Liu and Lampinen [2003]). This approach employs fuzzy logic to dynamically adapt the population size according to two changes. The first is the change of fitness function scores between two successive generations, and another is the change of the individuals between two successive generations. It has been proved that by employing this adaptation mechanism can improve convergence speed and reduce the number of fitness function evaluations (Liu and Lampinen [2003]).

7.1.2 A More Reliable Early Stopping Mechanism

As we can see from Chapter 6, the time consumption of our model is higher than RandomisedSearchCV in every test even though the population size is set to a small number, namely 10. In order to reduce the time consumption of our model, constructing a more reliable early stopping mechanism may be one solution to this problem. Specifically, as we discussed in Chapter 6, our current early stopping mechanism is hugely impacted by randomness. Further research on this topic is a direction of improving the time efficiency of our model.

7.1.3 Another Thought to Improve Time Efficiency

Another thought to improve the time efficiency of our model is to narrow the search space down beforehand. Eagle Strategy (ES) (Yang and Deb [2010]) can be a direction of further research on this topic. ES is a two-stage search algorithm that combines global search and local search. In the original paper, ES first employs sampling methods to select regions where promising solutions may occur. Then, it will use intensive local search in these promising regions. The algorithm will stop until the stopping criterion is reached.

According to (Talatahari et al. [2015]), DE can replace the local search algorithm in stage 2. To be specific, the original DE is a global search algorithm, and it can be turned into a local search algorithm by limiting its output solutions around the promising regions. Moreover, it has been proved that ES-DE is faster to converge that the original DE in the early stage of the evolution process.

Appendix

A.1 The Test Cases and Results for the Experiment on the Population of Different sizes

	Population Size
test 1	10
test 2	20
test 3	30
test 4	40
test 5	50
test 6	60
test 7	70
test 8	80
test 9	90
test 10	100

Table A.1: 10 test cases for the experiment on the population of different size (from 10 to 100)

	Time Consumption (s)	Final Accuracy
test 1	20.17715096473694	0.7209302325581395
test 2	36.54055404663086	0.7209302325581395
test 3	47.87428379058838	0.7209302325581395
test 4	67.62697386741638	0.7441860465116279
test 5	73.20662522315979	0.7209302325581395
test 6	95.684002161026	0.6976744186046512
test 7	122.85874390602112	7209302325581395
test 8	124.07775902748108	7209302325581395
test 9	127.64793300628662	7209302325581395
test 10	151.68379378318787	7209302325581395

Table A.2: Test results for the experiment on the population of different size (from 10 to 100)

	The Number of Processes
test 1	1
test 2	2
test 3	3
test 4	4
test 5	5
test 6	6
test 7	7
test 8	8
test 9	9
test 10	10
test 11	-1

Table A.3: 11 test cases for the experiment on multi-processing (The number of processes ranges from 1 to 10 plus "-1")

	time consumption (s)
test 1	297.20479702949524
test 2	139.73627400398254
test 3	81.96020102500916
test 4	79.13515591621399
test 5	62.258535861968994
test 6	65.64718008041382
test 7	61.833558082580566
test 8	65.78614115715027
test 9	54.8595609664917
test 10	51.51805114746094
test -1	56.73696708679199

Table A.4: Test results for the experiment on multi-processing (The number of processes ranges from 1 to 10 plus "-1")

	The Value Passed to *early_stop*
test 1	2
test 2	4
test 3	6
test 4	8
test 5	10
test 6	12
test 7	14
test 8	16
test 9	18
test 10	20
test 11	22
test 12	24
test 13	26
test 14	28
test 15	30
test 16	*None*

Table A.5: Test cases for the experiment on the early stopping mechanism (The value passed to parameter *early_stop* ranges from 2 to 30 plus *None*)

	Time Consumption (s)
test 1	8.530213117599487
test 2	11.456261157989502
test 3	34.19160199165344
test 4	75.95608711242676
test 5	43.57308793067932
test 6	63.5335259437561
test 7	75.34250712394714
test 8	80.34594297409058
test 9	99.36039566993713
test 10	128.91742396354675
test 10	81.91626214981079
test 10	92.38925909996033
test 10	99.00881695747375
test 10	98.7745521068573
test 10	114.57360005378723
test 10	121.17981696128845

Table A.6: Test cases for the experiment on the early stopping mechanism (The value passed to parameter *early_stop* ranges from 2 to 30 plus *None*)

A.2 The Search Spaces for the Comparison Experiments

The 20 tables shown below are the search spaces we employ in the comparison experiments on Ag_ordered_dataset and Pt_nanoparticle_datasset. The first 10 tables are for the former, and other 10 tables are for the latter.

Max_depth	*Min_samples_split*	*Min_samples_leaf*	*Max_leaf_nodes*
1 - 3	2 - 3	1 - 3	2 - 3
criterion	*max_features*		
"gini", "entropy"	"auto", "sqrt", "log2", "None"		

Table A.7: Search space 1 for the experiment on Ag_ordered_dataset

Max_depth	*Min_samples_split*	*Min_samples_leaf*	*Max_leaf_nodes*
1 - 6	2 - 6	1 - 6	2 - 6
criterion	*max_features*		
"gini", "entropy"	"auto", "sqrt", "log2", "None"		

Table A.8: Search space 2 for the experiment on Ag_ordered_dataset

Max_depth	*Min_samples_split*	*Min_samples_leaf*	*Max_leaf_nodes*
1 - 9	2 - 9	1 - 9	2 - 9
criterion	*max_features*		
"gini", "entropy"	"auto", "sqrt", "log2", "None"		

Table A.9: Search space 3 for the experiment on Ag_ordered_dataset

Max_depth	*Min_samples_split*	*Min_samples_leaf*	*Max_leaf_nodes*
1 - 12	2 - 12	1 - 12	2 - 12
criterion	*max_features*		
"gini", "entropy"	"auto", "sqrt", "log2", "None"		

Table A.10: Search space 4 for the experiment on Ag_ordered_dataset

Max_depth	*Min_samples_split*	*Min_samples_leaf*	*Max_leaf_nodes*
1 - 15	2 - 15	1 - 15	2 - 15
criterion	*max_features*		
"gini", "entropy"	"auto", "sqrt", "log2", "None"		

Table A.11: Search space 5 for the experiment on Ag_ordered_dataset

Max_depth	*Min_samples_split*	*Min_samples_leaf*	*Max_leaf_nodes*
1 - 18	2 - 18	1 - 18	2 - 18
criterion	*max_features*		
"gini", "entropy"	"auto", "sqrt", "log2", "None"		

Table A.12: Search space 6 for the experiment on Ag_ordered_dataset

Max_depth	*Min_samples_split*	*Min_samples_leaf*	*Max_leaf_nodes*
1 - 21	2 - 21	1 - 21	2 - 21
criterion	*max_features*		
"gini", "entropy"	"auto", "sqrt", "log2", "None"		

Table A.13: Search space 7 for the experiment on Ag_ordered_dataset

Max_depth	*Min_samples_split*	*Min_samples_leaf*	*Max_leaf_nodes*
1 - 24	2 - 24	1 - 24	2 - 24
criterion	*max_features*		
"gini", "entropy"	"auto", "sqrt", "log2", "None"		

Table A.14: Search space 8 for the experiment on Ag_ordered_dataset

Max_depth	*Min_samples_split*	*Min_samples_leaf*	*Max_leaf_nodes*
1 - 27	2 - 27	1 - 27	2 - 27
criterion	*max_features*		
"gini", "entropy"	"auto", "sqrt", "log2", "None"		

Table A.15: Search space 9 for the experiment on Ag_ordered_dataset

Max_depth	*Min_samples_split*	*Min_samples_leaf*	*Max_leaf_nodes*
1 - 30	2 - 30	1 - 30	2 - 30
criterion	*max_features*		
"gini", "entropy"	"auto", "sqrt", "log2", "None"		

Table A.16: Search space 10 for the experiment on Ag_ordered_dataset

Max_depth	*Min_samples_split*	*Min_samples_leaf*	*Max_leaf_nodes*
1 - 3	2 - 3	1 - 3	2 - 3
max_features			
"auto", "sqrt", "log2", "None"			

Table A.17: Search space 1 for the experiment on Pt_nanoparticle_datasset

Max_depth	*Min_samples_split*	*Min_samples_leaf*	*Max_leaf_nodes*
1 - 6	2 - 6	1 - 6	2 - 6
max_features			
"auto", "sqrt", "log2", "None"			

Table A.18: Search space 2 for the experiment on Pt_nanoparticle_datasset

Max_depth	*Min_samples_split*	*Min_samples_leaf*	*Max_leaf_nodes*
1 - 9	2 - 9	1 - 9	2 - 9
max_features			
"auto", "sqrt", "log2", "None"			

Table A.19: Search space 3 for the experiment on Pt_nanoparticle_datasset

Max_depth	*Min_samples_split*	*Min_samples_leaf*	*Max_leaf_nodes*
1 - 12	2 - 12	1 - 12	2 - 12
max_features			
"auto", "sqrt", "log2", "None"			

Table A.20: Search space 4 for the experiment on Pt_nanoparticle_datasset

Max_depth	*Min_samples_split*	*Min_samples_leaf*	*Max_leaf_nodes*
1 - 15	2 - 15	1 - 15	2 - 15
max_features			
"auto", "sqrt", "log2", "None"			

Table A.21: Search space 5 for the experiment on Pt_nanoparticle_datasset

	Max_depth	*Min_samples_split*	*Min_samples_leaf*	*Max_leaf_nodes*
	1 - 18	2 - 18	1 - 18	2 - 18
	max_features			
"auto", "sqrt", "log2", "None"				

Table A.22: Search space 6 for the experiment on Pt_nanoparticle_datasset

	Max_depth	*Min_samples_split*	*Min_samples_leaf*	*Max_leaf_nodes*
	1 - 21	2 - 21	1 - 21	2 - 21
	max_features			
"auto", "sqrt", "log2", "None"				

Table A.23: Search space 7 for the experiment on Pt_nanoparticle_datasset

	Max_depth	*Min_samples_split*	*Min_samples_leaf*	*Max_leaf_nodes*
	1 - 24	2 - 24	1 - 24	2 - 24
	max_features			
"auto", "sqrt", "log2", "None"				

Table A.24: Search space 8 for the experiment on Pt_nanoparticle_datasset

	Max_depth	*Min_samples_split*	*Min_samples_leaf*	*Max_leaf_nodes*
	1 - 27	2 - 27	1 - 27	2 - 27
	max_features			
"auto", "sqrt", "log2", "None"				

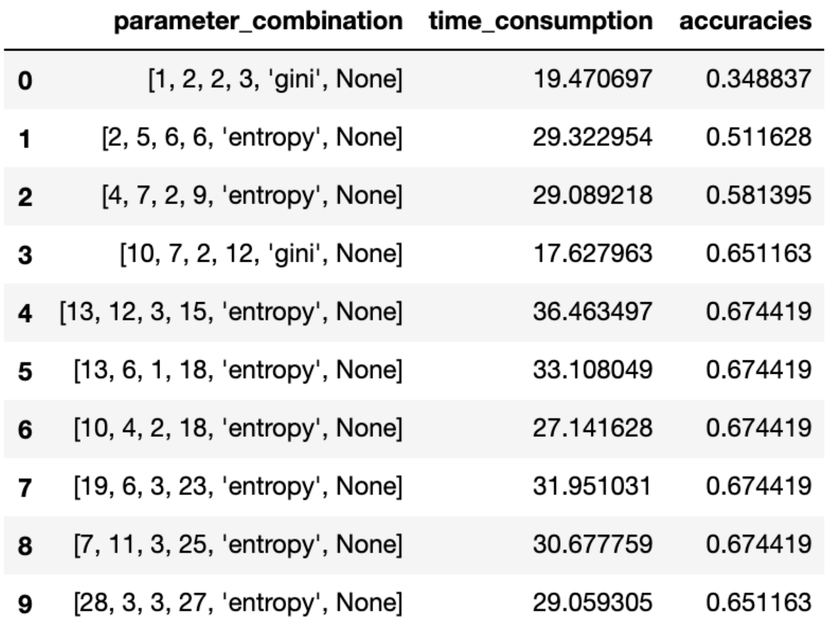
Table A.25: Search space 9 for the experiment on Pt_nanoparticle_datasset

	Max_depth	*Min_samples_split*	*Min_samples_leaf*	*Max_leaf_nodes*
	1 - 30	2 - 30	1 - 30	2 - 30
	max_features			
"auto", "sqrt", "log2", "None"				

Table A.26: Search space 10 for the experiment on Pt_nanoparticle_datasset

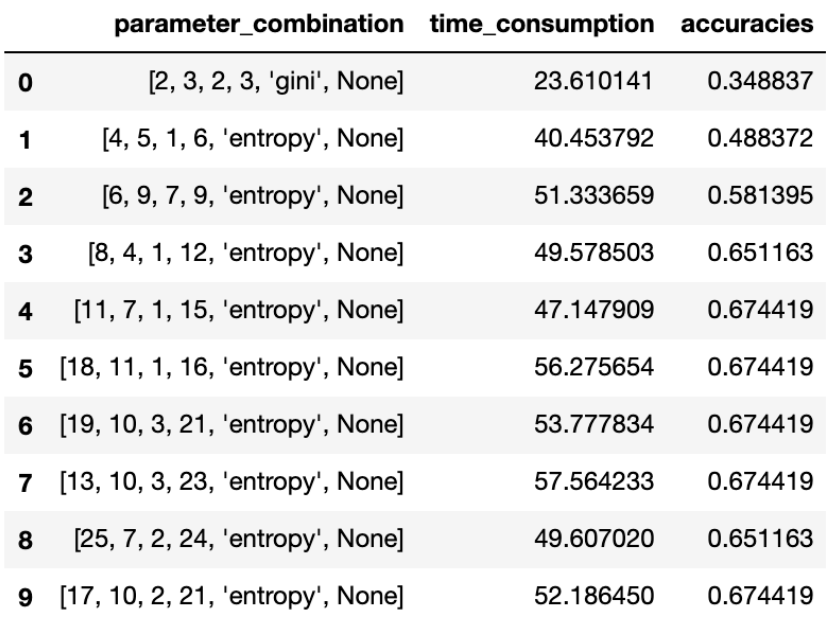
A.3 Test Results for the Comparison Experiment

This section shows the results of comparison experiments on Ag_ordered_dataset and Pt_nanoparticle_datasset.



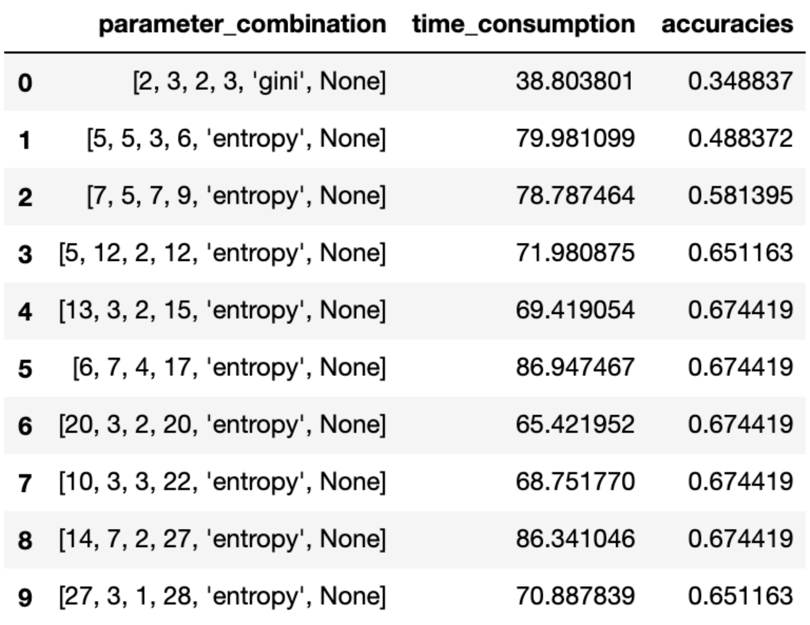
	parameter_combination	time_consumption	accuracies
0	[1, 2, 2, 3, 'gini', None]	19.470697	0.348837
1	[2, 5, 6, 6, 'entropy', None]	29.322954	0.511628
2	[4, 7, 2, 9, 'entropy', None]	29.089218	0.581395
3	[10, 7, 2, 12, 'gini', None]	17.627963	0.651163
4	[13, 12, 3, 15, 'entropy', None]	36.463497	0.674419
5	[13, 6, 1, 18, 'entropy', None]	33.108049	0.674419
6	[10, 4, 2, 18, 'entropy', None]	27.141628	0.674419
7	[19, 6, 3, 23, 'entropy', None]	31.951031	0.674419
8	[7, 11, 3, 25, 'entropy', None]	30.677759	0.674419
9	[28, 3, 3, 27, 'entropy', None]	29.059305	0.651163

Figure A.1: The time consumption and final accuracy of EvolutionarySearchCV when the population size is set to 10 (Ag_ordered_dataset)



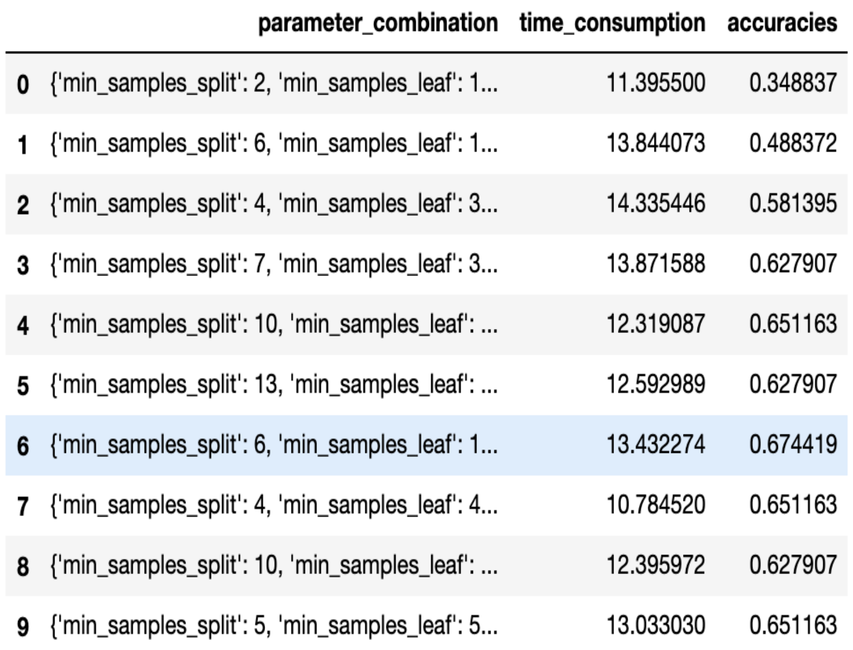
	parameter_combination	time_consumption	accuracies
0	[2, 3, 2, 3, 'gini', None]	23.610141	0.348837
1	[4, 5, 1, 6, 'entropy', None]	40.453792	0.488372
2	[6, 9, 7, 9, 'entropy', None]	51.333659	0.581395
3	[8, 4, 1, 12, 'entropy', None]	49.578503	0.651163
4	[11, 7, 1, 15, 'entropy', None]	47.147909	0.674419
5	[18, 11, 1, 16, 'entropy', None]	56.275654	0.674419
6	[19, 10, 3, 21, 'entropy', None]	53.777834	0.674419
7	[13, 10, 3, 23, 'entropy', None]	57.564233	0.674419
8	[25, 7, 2, 24, 'entropy', None]	49.607020	0.651163
9	[17, 10, 2, 21, 'entropy', None]	52.186450	0.674419

Figure A.2: The time consumption and final accuracy of EvolutionarySearchCV when the population size is set to 20 (Ag_ordered_dataset)



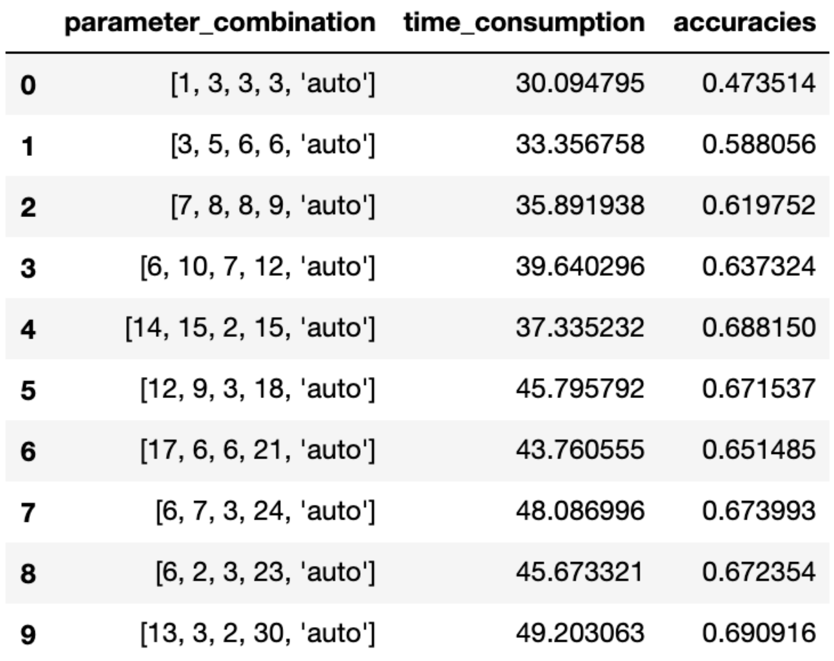
	parameter_combination	time_consumption	accuracies
0	[2, 3, 2, 3, 'gini', None]	38.803801	0.348837
1	[5, 5, 3, 6, 'entropy', None]	79.981099	0.488372
2	[7, 5, 7, 9, 'entropy', None]	78.787464	0.581395
3	[5, 12, 2, 12, 'entropy', None]	71.980875	0.651163
4	[13, 3, 2, 15, 'entropy', None]	69.419054	0.674419
5	[6, 7, 4, 17, 'entropy', None]	86.947467	0.674419
6	[20, 3, 2, 20, 'entropy', None]	65.421952	0.674419
7	[10, 3, 3, 22, 'entropy', None]	68.751770	0.674419
8	[14, 7, 2, 27, 'entropy', None]	86.341046	0.674419
9	[27, 3, 1, 28, 'entropy', None]	70.887839	0.651163

Figure A.3: The time consumption and final accuracy of EvolutionarySearchCV when the population size is set to 30 (Ag_ordered_dataset)



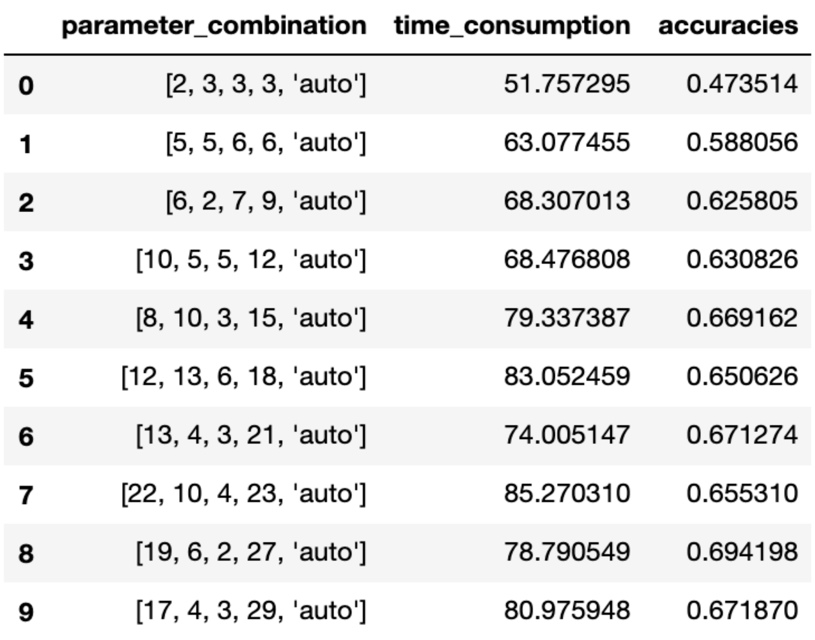
	parameter_combination	time_consumption	accuracies
0	{'min_samples_split': 2, 'min_samples_leaf': 1...	11.395500	0.348837
1	{'min_samples_split': 6, 'min_samples_leaf': 1...	13.844073	0.488372
2	{'min_samples_split': 4, 'min_samples_leaf': 3...	14.335446	0.581395
3	{'min_samples_split': 7, 'min_samples_leaf': 3...	13.871588	0.627907
4	{'min_samples_split': 10, 'min_samples_leaf': ...	12.319087	0.651163
5	{'min_samples_split': 13, 'min_samples_leaf': ...	12.592989	0.627907
6	{'min_samples_split': 6, 'min_samples_leaf': 1...	13.432274	0.674419
7	{'min_samples_split': 4, 'min_samples_leaf': 4...	10.784520	0.651163
8	{'min_samples_split': 10, 'min_samples_leaf': ...	12.395972	0.627907
9	{'min_samples_split': 5, 'min_samples_leaf': 5...	13.033030	0.651163

Figure A.4: The time consumption and final accuracy of RandomisedSearchCV (Ag_ordered_dataset)



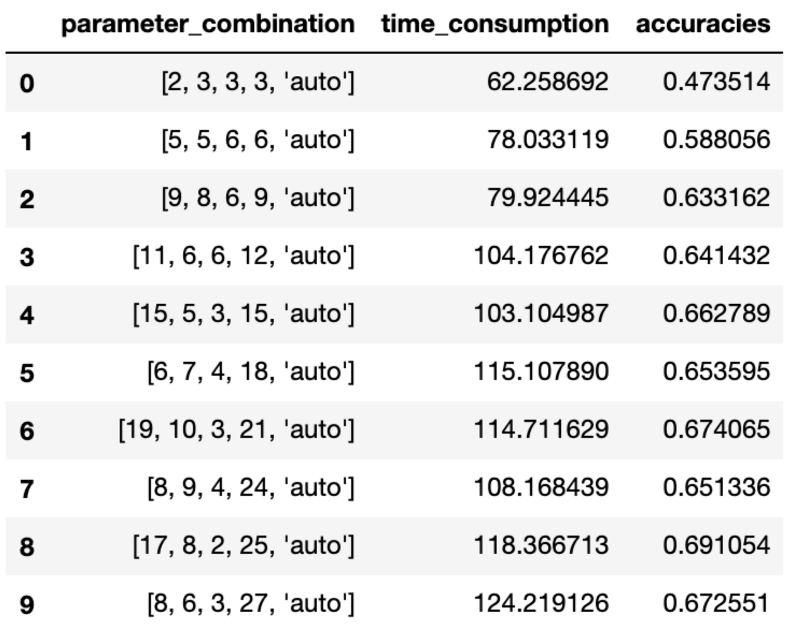
	parameter_combination	time_consumption	accuracies
0	[1, 3, 3, 3, 'auto']	30.094795	0.473514
1	[3, 5, 6, 6, 'auto']	33.356758	0.588056
2	[7, 8, 8, 9, 'auto']	35.891938	0.619752
3	[6, 10, 7, 12, 'auto']	39.640296	0.637324
4	[14, 15, 2, 15, 'auto']	37.335232	0.688150
5	[12, 9, 3, 18, 'auto']	45.795792	0.671537
6	[17, 6, 6, 21, 'auto']	43.760555	0.651485
7	[6, 7, 3, 24, 'auto']	48.086996	0.673993
8	[6, 2, 3, 23, 'auto']	45.673321	0.672354
9	[13, 3, 2, 30, 'auto']	49.203063	0.690916

Figure A.5: The time consumption and final accuracy of EvolutionarySearchCV when the population size is set to 10 (Pt_nanoparticle_datasset)



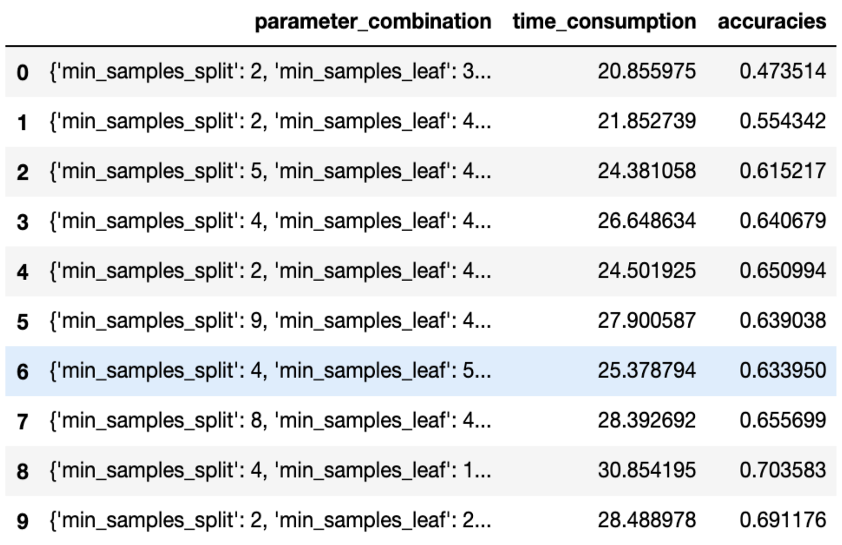
	parameter_combination	time_consumption	accuracies
0	[2, 3, 3, 3, 'auto']	51.757295	0.473514
1	[5, 5, 6, 6, 'auto']	63.077455	0.588056
2	[6, 2, 7, 9, 'auto']	68.307013	0.625805
3	[10, 5, 5, 12, 'auto']	68.476808	0.630826
4	[8, 10, 3, 15, 'auto']	79.337387	0.669162
5	[12, 13, 6, 18, 'auto']	83.052459	0.650626
6	[13, 4, 3, 21, 'auto']	74.005147	0.671274
7	[22, 10, 4, 23, 'auto']	85.270310	0.655310
8	[19, 6, 2, 27, 'auto']	78.790549	0.694198
9	[17, 4, 3, 29, 'auto']	80.975948	0.671870

Figure A.6: The time consumption and final accuracy of EvolutionarySearchCV when the population size is set to 20 (Pt_nanoparticle_datasset)



	parameter_combination	time_consumption	accuracies
0	[2, 3, 3, 3, 'auto']	62.258692	0.473514
1	[5, 5, 6, 6, 'auto']	78.033119	0.588056
2	[9, 8, 6, 9, 'auto']	79.924445	0.633162
3	[11, 6, 6, 12, 'auto']	104.176762	0.641432
4	[15, 5, 3, 15, 'auto']	103.104987	0.662789
5	[6, 7, 4, 18, 'auto']	115.107890	0.653595
6	[19, 10, 3, 21, 'auto']	114.711629	0.674065
7	[8, 9, 4, 24, 'auto']	108.168439	0.651336
8	[17, 8, 2, 25, 'auto']	118.366713	0.691054
9	[8, 6, 3, 27, 'auto']	124.219126	0.672551

Figure A.7: The time consumption and final accuracy of EvolutionarySearchCV when the population size is set to 30 (Pt_nanoparticle_datasset)



	parameter_combination	time_consumption	accuracies
0	{'min_samples_split': 2, 'min_samples_leaf': 3...	20.855975	0.473514
1	{'min_samples_split': 2, 'min_samples_leaf': 4...	21.852739	0.554342
2	{'min_samples_split': 5, 'min_samples_leaf': 4...	24.381058	0.615217
3	{'min_samples_split': 4, 'min_samples_leaf': 4...	26.648634	0.640679
4	{'min_samples_split': 2, 'min_samples_leaf': 4...	24.501925	0.650994
5	{'min_samples_split': 9, 'min_samples_leaf': 4...	27.900587	0.639038
6	{'min_samples_split': 4, 'min_samples_leaf': 5...	25.378794	0.633950
7	{'min_samples_split': 8, 'min_samples_leaf': 4...	28.392692	0.655699
8	{'min_samples_split': 4, 'min_samples_leaf': 1...	30.854195	0.703583
9	{'min_samples_split': 2, 'min_samples_leaf': 2...	28.488978	0.691176

Figure A.8: The time consumption (s) and final accuracy of RandomisedSearchCV (Pt_nanoparticle_datasset)

A.4 Other Experiments about EvolutionarySearchCV (not discussed in the thesis)

The test shown below compares the time consumption and the final accuracy of EvolutionarySearchCV, when the population size is changed from 10 to 20. This experiment is performed on Ag_ordered_dataset. The search spaces in this experiment are shown in Appendix A.2 (Table A.7 to Table A.16).

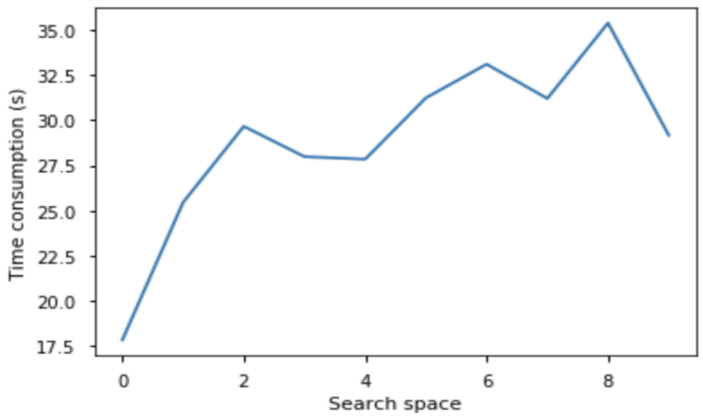


Figure A.9: The time consumption of EvolutionarySearchCV when the population size is set to 10 (Ag_ordered_dataset)

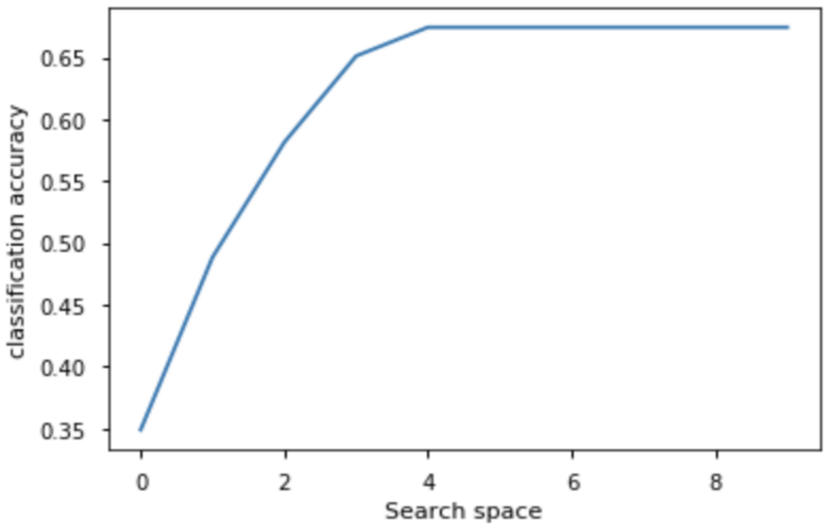
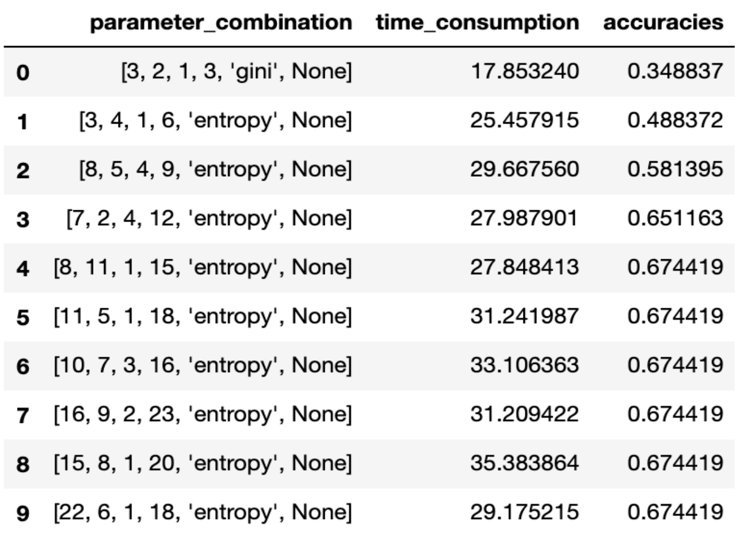


Figure A.10: The final accuracy returned by EvolutionarySearchCV when the population size is set to 10 (Ag_ordered_dataset)



	parameter_combination	time_consumption	accuracies
0	[3, 2, 1, 3, 'gini', None]	17.853240	0.348837
1	[3, 4, 1, 6, 'entropy', None]	25.457915	0.488372
2	[8, 5, 4, 9, 'entropy', None]	29.667560	0.581395
3	[7, 2, 4, 12, 'entropy', None]	27.987901	0.651163
4	[8, 11, 1, 15, 'entropy', None]	27.848413	0.674419
5	[11, 5, 1, 18, 'entropy', None]	31.241987	0.674419
6	[10, 7, 3, 16, 'entropy', None]	33.106363	0.674419
7	[16, 9, 2, 23, 'entropy', None]	31.209422	0.674419
8	[15, 8, 1, 20, 'entropy', None]	35.383864	0.674419
9	[22, 6, 1, 18, 'entropy', None]	29.175215	0.674419

Figure A.11: The returned hyper-paramter combination, time consumption and final accuracy of EvolutionarySearchCV when the population size is set to 10 (Ag_ordered_dataset)

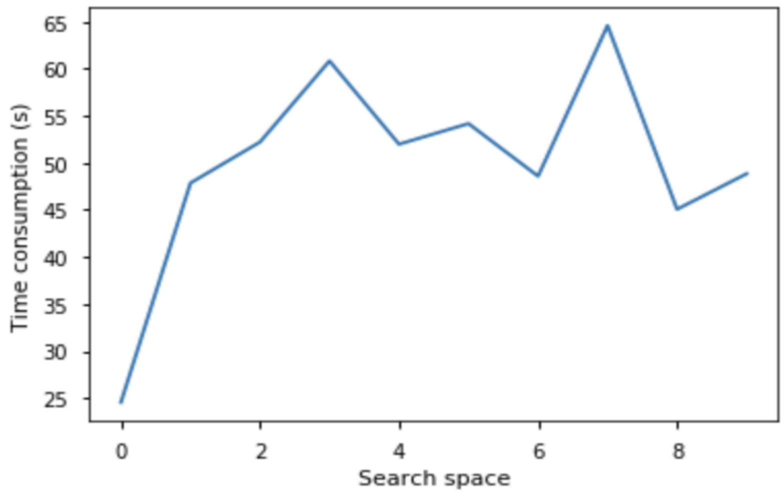


Figure A.12: The time consumption of EvolutionarySearchCV when the population size is set to 20 (Ag_ordered_dataset)

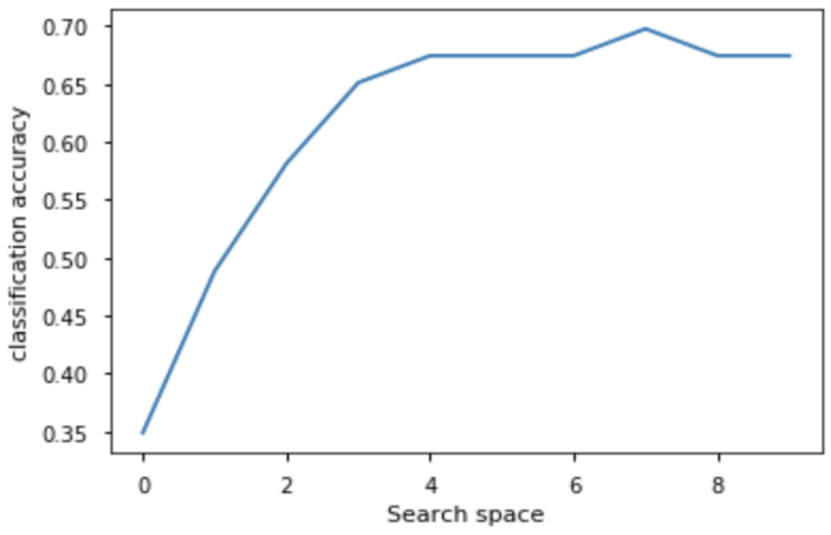
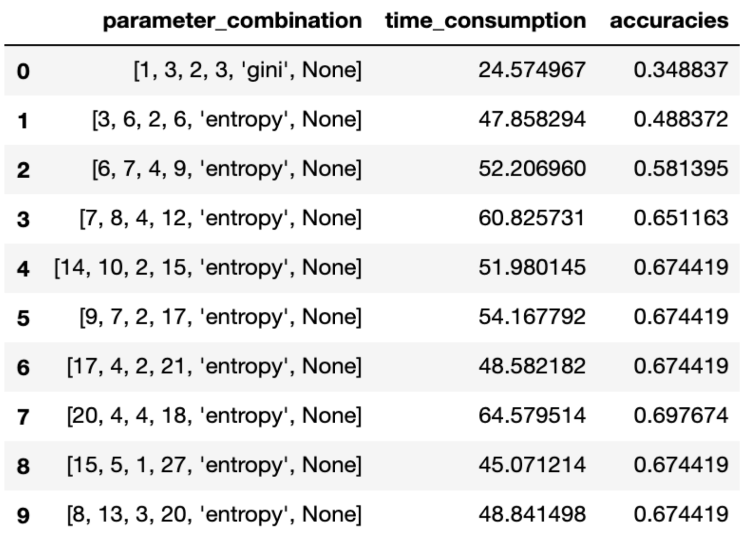


Figure A.13: The final accuracy returned by EvolutionarySearchCV when the population size is set to 20 (Ag_ordered_dataset)



	parameter_combination	time_consumption	accuracies
0	[1, 3, 2, 3, 'gini', None]	24.574967	0.348837
1	[3, 6, 2, 6, 'entropy', None]	47.858294	0.488372
2	[6, 7, 4, 9, 'entropy', None]	52.206960	0.581395
3	[7, 8, 4, 12, 'entropy', None]	60.825731	0.651163
4	[14, 10, 2, 15, 'entropy', None]	51.980145	0.674419
5	[9, 7, 2, 17, 'entropy', None]	54.167792	0.674419
6	[17, 4, 2, 21, 'entropy', None]	48.582182	0.674419
7	[20, 4, 4, 18, 'entropy', None]	64.579514	0.697674
8	[15, 5, 1, 27, 'entropy', None]	45.071214	0.674419
9	[8, 13, 3, 20, 'entropy', None]	48.841498	0.674419

Figure A.14: The returned hyper-paramter combination, time consumption and final accuracy of EvolutionarySearchCV when the population size is set to 20 (Ag_ordered_dataset)

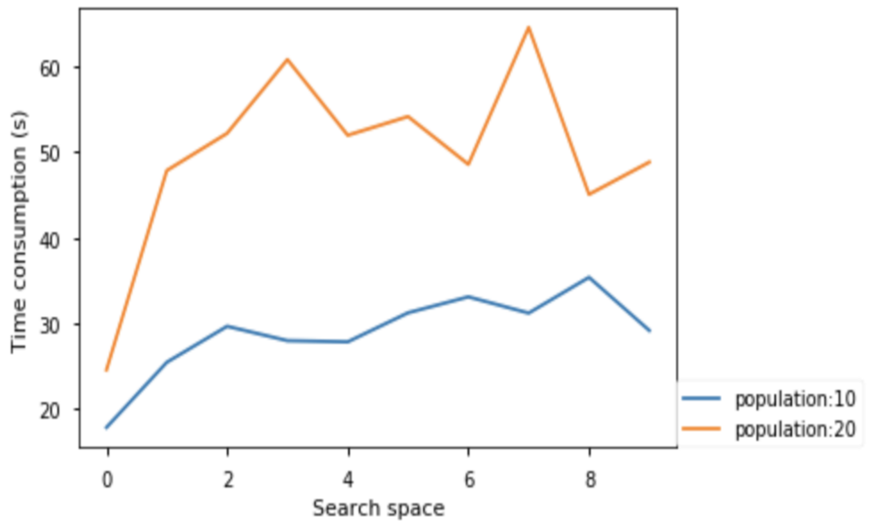


Figure A.15: The comparison of time consumption of EvolutionarySearchCV when the population size is changed from 10 to 20 (Ag_ordered_dataset)

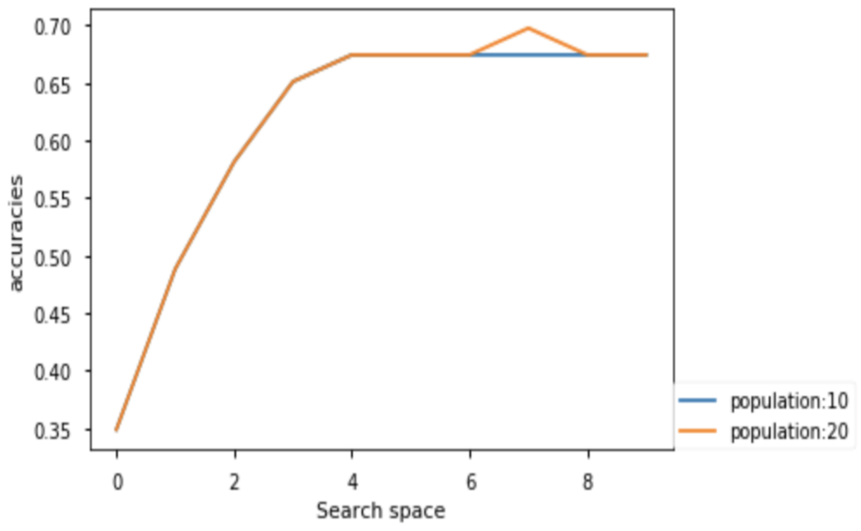


Figure A.16: The comparison of final accuracy returned by EvolutionarySearchCV when the population size is changed from 10 to 20 (Ag_ordered_dataset)

Bibliography

Avriel, M., 2003. *Nonlinear programming: analysis and methods*. Courier Corporation. (cited on page 9)

Bäck, T.; Fogel, D. B.; and Michalewicz, Z., 1997. Handbook of evolutionary computation. *Release*, 97, 1 (1997), B1. (cited on page 4)

Bahdanau, D.; Cho, K.; and Bengio, Y., 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, (2014). (cited on pages 28 and 29)

Balamurugan, R.; Natarajan, A.; and Premalatha, K., 2015. Stellar-mass black hole optimization for biclustering microarray gene expression data. *Applied Artificial Intelligence*, 29, 4 (2015), 353–381. (cited on page 5)

Barnard, A.; Sun, B.; Motevalli Soumehsaraei, B.; and Opletal, G., 2017. Silver nanoparticle data set. v3. In *Data Collection*. CSIRO. (cited on page 22)

Barnard, A.; Sun, B.; and Opletal, G., 2018. Platinum nanoparticle data set. v2. In *Data Collection*. CSIRO. (cited on page 23)

Barricelli, N. A. et al., 1954. Esempi numerici di processi di evoluzione. *Methodos*, 6, 21-22 (1954), 45–68. (cited on page 4)

Barros, M. d. O., 2014. An experimental evaluation of the importance of randomness in hill climbing searches applied to software engineering problems. *Empirical Software Engineering*, 19, 5 (2014), 1423–1465. (cited on page 4)

Bengio, Y., 2012. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, 437–478. Springer. (cited on page 9)

Bergstra, J. and Bengio, Y., 2012. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13, 2 (2012). (cited on pages ix, 3, and 4)

Bianchi, L.; Dorigo, M.; Gambardella, L. M.; and Gutjahr, W. J., 2009. A survey on metaheuristics for stochastic combinatorial optimization. *Natural Computing*, 8, 2 (2009), 239–287. (cited on page 5)

Bishop, C. M., 2006. *Pattern recognition and machine learning*. springer. (cited on pages xv, 8, and 11)

Brownlee, J., 2019. What is the difference between a parameter and a hyperparameter. *Machine Learning Mastery) Retrieved February*, 24 (2019), 2020. (cited on pages 1 and 2)

Buduma, N. and Locascio, N., 2017. *Fundamentals of deep learning: Designing next-generation machine intelligence algorithms*. " O'Reilly Media, Inc.". (cited on page 10)

Cawley, G. C. and Talbot, N. L., 2010. On over-fitting in model selection and subsequent selection bias in performance evaluation. *The Journal of Machine Learning Research*, 11 (2010), 2079–2107. (cited on page 11)

Chandy, K. M.; Misra, J.; and Haas, L. M., 1983. Distributed deadlock detection. *ACM Transactions on Computer Systems (TOCS)*, 1, 2 (1983), 144–156. (cited on page 31)

Chen, X. and Ishwaran, H., 2012. Random forests for genomic data analysis. *Genomics*, 99, 6 (2012), 323–329. (cited on page 22)

Chicco, D. and Jurman, G., 2020. The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation. *BMC genomics*, 21, 1 (2020), 1–13. (cited on page 35)

Chicco, D.; Tötsch, N.; and Jurman, G., 2021. The matthews correlation coefficient (mcc) is more reliable than balanced accuracy, bookmaker informedness, and markedness in two-class confusion matrix evaluation. *BioData mining*, 14, 1 (2021), 1–22. (cited on page 35)

Cho, K.; Van Merriënboer, B.; Gulcehre, C.; Bahdanau, D.; Bougares, F.; Schwenk, H.; and Bengio, Y., 2014. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, (2014). (cited on pages 28 and 29)

Claesen, M. and De Moor, B., 2015. Hyperparameter search in machine learning. *arXiv preprint arXiv:1502.02127*, (2015). (cited on page 2)

Das, S.; Abraham, A.; Chakraborty, U. K.; and Konar, A., 2009. Differential evolution using a neighborhood-based mutation operator. *IEEE transactions on evolutionary computation*, 13, 3 (2009), 526–553. (cited on pages 15 and 17)

dos Santos Amorim, E. P.; Xavier, C. R.; Campos, R. S.; and dos Santos, R. W., 2012. Comparison between genetic algorithms and differential evolution for solving the history matching problem. In *International Conference on Computational Science and Its Applications*, 635–648. Springer. (cited on page 5)

Duchi, J.; Hazan, E.; and Singer, Y., 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12, 7 (2011). (cited on page 9)

Fawcett, T., 2006. An introduction to roc analysis. *Pattern recognition letters*, 27, 8 (2006), 861–874. (cited on page 35)

Gämperle, R.; Müller, S. D.; and Koumoutsakos, P., 2002. A parameter study for differential evolution. *Advances in intelligent systems, fuzzy systems, evolutionary computation*, 10, 10 (2002), 293–298. (cited on page 19)

Goodfellow, I.; Bengio, Y.; Courville, A.; and Bengio, Y., 2016. *Deep learning*, vol. 1. MIT press Cambridge. (cited on page 9)

Hinton, G.; Srivastava, N.; and Swersky, K., 2012. Neural networks for machine learning. *Coursera, video lectures*, 264, 1 (2012). (cited on page 9)

Hinton, G. E.; Osindero, S.; and Teh, Y.-W., 2006. A fast learning algorithm for deep belief nets. *Neural computation*, 18, 7 (2006), 1527–1554. (cited on page 10)

Ho, T. K., 1995. Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, vol. 1, 278–282. IEEE. (cited on page 21)

Ho, T. K., 1998. The random subspace method for constructing decision forests. *IEEE transactions on pattern analysis and machine intelligence*, 20, 8 (1998), 832–844. (cited on page 21)

Hochreiter, S. and Schmidhuber, J., 1997. Long short-term memory. *Neural computation*, 9, 8 (1997), 1735–1780. (cited on page 28)

Holland, J. H., 1992. Genetic algorithms. *Scientific american*, 267, 1 (1992), 66–73. (cited on page 5)

Hsu, C.-W.; Chang, C.-C.; Lin, C.-J.; et al., 2003. A practical guide to support vector classification. (cited on page 3)

Hu, J.; Niu, H.; Carrasco, J.; Lennox, B.; and Arvin, F., 2020. Voronoi-based multi-robot autonomous exploration in unknown environments via deep reinforcement learning. *IEEE Transactions on Vehicular Technology*, 69, 12 (2020), 14413–14423. (cited on page 1)

Hutter, F.; Hoos, H.; and Leyton-Brown, K., 2014. An efficient approach for assessing hyperparameter importance. In *International conference on machine learning*, 754–762. PMLR. (cited on page 7)

Igel, C., 2014. No free lunch theorems: Limitations and perspectives of metaheuristics. In *Theory and principled methods for the design of metaheuristics*, 1–23. Springer. (cited on page 2)

Ilonen, J.; Kamarainen, J.-K.; and Lampinen, J., 2003. Differential evolution training algorithm for feed-forward neural networks. *Neural Processing Letters*, 17, 1 (2003), 93–105. (cited on page 5)

Ivakhnenko, A. G. and Lapa, V. G., 1967. Cybernetics and forecasting techniques. (1967). (cited on page 9)

Kettner, J.; Ebbers, M.; O'Brien, W.; and Ogden, B., 2011. *Introduction to the New Mainframe: z/OS Basics*. IBM Redbooks. (cited on page 30)

Kingma, D. P. and Ba, J., 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, (2014). (cited on page 9)

Kuhn, M.; Johnson, K.; et al., 2013. *Applied predictive modeling*, vol. 26. Springer. (cited on page 2)

Kuila, P. and Jana, P. K., 2014. A novel differential evolution based clustering algorithm for wireless sensor networks. *Applied soft computing*, 25 (2014), 414–425. (cited on pages xv and 14)

Lerman, P., 1980. Fitting segmented regression models by grid search. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 29, 1 (1980), 77–84. (cited on page ix)

Liu, J. and Lampinen, J., 2002. Setting the control parameter of the differential evolution method. In *Proceedings of the 8th International Conference on Soft Computing (MENDEL)*, 11–18. (cited on page 19)

Liu, J. and Lampinen, J., 2003. Population size adaptation for differential evolution algorithm using fuzzy logic. In *Intelligent Systems Design and Applications*, 425–436. Springer. (cited on page 56)

Marchese Robinson, R. L.; Palczewska, A.; Palczewski, J.; and Kidley, N., 2017. Comparison of the predictive performance and interpretability of random forest and linear models on benchmark data sets. *Journal of chemical information and modeling*, 57, 8 (2017), 1773–1792. (cited on page 22)

Mitchell, M., 1998. *An introduction to genetic algorithms*. MIT press. (cited on page 4)

Mitchell, T. M. et al., 1997. Machine learning. (1997). (cited on page 1)

MUKHERJEE, S. S.; KONTZ, M.; AND REINHARDT, S. K., 2002. Detailed design and evaluation of redundant multi-threading alternatives. In *Proceedings 29th annual international symposium on computer architecture*, 99–110. IEEE. (cited on page 31)

NAIR, V. AND HINTON, G. E., 2010. Rectified linear units improve restricted boltzmann machines. In *Icml*. (cited on page 9)

NG, A., 2017. Improving deep neural networks: Hyperparameter tuning, regularization and optimization. *Coursera. URL: https://www. coursera. org/learn/deep-neural-network [accessed 2020-06-20]*, (2017). (cited on pages 4 and 13)

OPITZ, D. AND MACLIN, R., 1999. Popular ensemble methods: An empirical study. *Journal of artificial intelligence research*, 11 (1999), 169–198. (cited on page 21)

OSBORN, G., 1902. 109.[d. 6. d.] mnemonic for hyperbolic formulae. *The Mathematical Gazette*, 2, 34 (1902), 189–189. (cited on page 9)

PIRYONESI, S. M. AND EL-DIRABY, T. E., 2020. Data analytics in asset management: Cost-effective prediction of the pavement condition index. *Journal of Infrastructure Systems*, 26, 1 (2020), 04019036. (cited on page 35)

POLIKAR, R., 2006. Ensemble based systems in decision making. *IEEE Circuits and systems magazine*, 6, 3 (2006), 21–45. (cited on page 21)

POWERS, D. M., 2020. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. *arXiv preprint arXiv:2010.16061*, (2020). (cited on page 35)

PRAJIT, R.; BARRET, Z.; AND QUOC, V. L., 2017. Swish: a self-gated activation function. *arXiv preprint arXiv:1710.05941*, (2017). (cited on page 9)

PRICE, K.; STORN, R. M.; AND LAMPINEN, J. A., 2006. *Differential evolution: a practical approach to global optimization*. Springer Science & Business Media. (cited on page 19)

PUTATUNDA, S. AND RAMA, K., 2019. A modified bayesian optimization based hyperparameter tuning approach for extreme gradient boosting. 1–6. IEEE. (cited on pages ix and 2)

QIAN, N., 1999. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12, 1 (1999), 145–151. (cited on page 9)

QIN, A. K. AND SUGANTHAN, P. N., 2005. Self-adaptive differential evolution algorithm for numerical optimization. In *2005 IEEE congress on evolutionary computation*, vol. 2, 1785–1791. IEEE. (cited on pages ix, 14, 17, 19, 20, 21, 27, and 55)

QUINLAN, J. R., 1986. Induction of decision trees. *Machine learning*, 1, 1 (1986), 81–106. (cited on page 22)

RAJAGOPAL, R., 1999. *Introduction to Microsoft Windows NT cluster server: Programming and administration*. CRC Press. (cited on page 30)

RASTRIGIN, L., 1963. The convergence of the random search method in the extremal control of a many parameter system. *Automaton & Remote Control*, 24 (1963), 1337–1342. (cited on page 4)

RIMAC-DRLJE, S.; VRANJEŠ, M.; AND ŽAGAR, D., 2010. Foveated mean squared error—a novel video quality metric. *Multimedia tools and applications*, 49, 3 (2010), 425–445. (cited on page 38)

ROBBINS, H. AND MONRO, S., 1951. A stochastic approximation method. *The annals of mathematical statistics*, (1951), 400–407. (cited on page 9)

ROGALSKY, T.; KOCABIYIK, S.; AND DERKSEN, R., 2000. Differential evolution in aerodynamic optimization. *Canadian Aeronautics and Space Journal*, 46, 4 (2000), 183–190. (cited on page 19)

ROKACH, L., 2010. Ensemble-based classifiers. *Artificial intelligence review*, 33, 1 (2010), 1–39. (cited on page 21)

ROUSE, M., 2019. Multi-core processor. *Internet: www. searchdatacenter. techtarget. com/definition/multi-core-processor*, (2019). (cited on page 30)

RUMELHART, D. E.; HINTON, G. E.; AND WILLIAMS, R. J., 1986. Learning representations by back-propagating errors. *nature*, 323, 6088 (1986), 533–536. (cited on page 9)

SAMMUT, C. AND WEBB, G. I., 2011. *Encyclopedia of machine learning*. Springer Science & Business Media. (cited on page 35)

SCHRACK, G. AND CHOIT, M., 1976. Optimized relative step size random searches. *Mathematical Programming*, 10, 1 (1976), 230–244. (cited on page 4)

SCHUMER, M. AND STEIGLITZ, K., 1968. Adaptive step size random search. *IEEE Transactions on Automatic Control*, 13, 3 (1968), 270–276. (cited on page 4)

SENI, G. AND ELDER, J. F., 2010. Ensemble methods in data mining: improving accuracy through combining predictions. *Synthesis lectures on data mining and knowledge discovery*, 2, 1 (2010), 1–126. (cited on page 11)

STONE, M., 1974. Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society: Series B (Methodological)*, 36, 2 (1974), 111–133. (cited on page 10)

Storn, R., 1996. Differential evolution design of an iir-filter. In *Proceedings of IEEE international conference on evolutionary computation*, 268–273. IEEE. (cited on page 5)

Storn, R. and Price, K., 1997. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11, 4 (1997), 341–359. (cited on pages ix, xv, 5, 7, 13, 15, 16, and 17)

Strobl, C.; Boulesteix, A.-L.; Kneib, T.; Augustin, T.; and Zeileis, A., 2008. Conditional variable importance for random forests. *BMC bioinformatics*, 9, 1 (2008), 1–11. (cited on page 22)

Sun, Y.; Ding, S.; Zhang, Z.; and Jia, W., 2021. An improved grid search algorithm to optimize svr for prediction. *Soft Computing*, 25, 7 (2021), 5633–5644. (cited on pages xv, 12, and 13)

Talatahari, S.; Gandomi, A. H.; Yang, X.-S.; and Deb, S., 2015. Optimum design of frame structures using the eagle strategy with differential evolution. *Engineering Structures*, 91 (2015), 16–25. (cited on page 56)

Tharwat, A., 2020. Classification assessment methods. *Applied Computing and Informatics*, (2020). (cited on page 35)

Wang, R. and Li, J., 2019. Bayes test of precision, recall, and f1 measure for comparison of two natural language processing models. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 4135–4145. (cited on page 35)

Yang, X.-S. and Deb, S., 2010. Eagle strategy using lévy walk and firefly algorithms for stochastic optimization. In *Nature inspired cooperative strategies for optimization (NICSO 2010)*, 101–111. Springer. (cited on page 56)

Yerushalmy, J., 1947. Statistical problems in assessing methods of medical diagnosis, with special reference to x-ray techniques. *Public Health Reports (1896-1970)*, (1947), 1432–1449. (cited on page 35)

Yu, T. and Zhu, H., 2020. Hyper-parameter optimization: A review of algorithms and applications. *arXiv preprint arXiv:2003.05689*, (2020). (cited on pages 3 and 9)