UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA

FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ ROMÂNĂ

LUCRARE DE LICENŢĂ

Învățare Automată folosind Algoritmi Evolutivi

Conducător ştiinţific
Conf. Mihai Suciu

Absolvent
Mihai Anton

2020

BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

COMPUTER SCIENCE SPECIALIZATION IN ROMANIAN

DIPLOMA THESIS

Automated Machine Learning using Evolutionary Algorithms

Supervisor
Conf. Mihai Suciu

Author
Mihai Anton

2020

Abstract

As the global data quantity already follows an exponential trend, machine learning has become present in every application, creating a great demand for general know-how, be it data scientists or computer scientists with related knowledge. Currently, the demand for work to be done surpasses the offer of such professionals, thus automatic solutions have to be found. The classical machine learning process involves data engineering, model selection and hyperparameter tuning for the chosen model. Due to the highly repetitive nature of trial and error of these tasks, automation can play a big role in optimizing time spent on them. Automated Machine Learning comes to help the process by adding different optimization techniques that help data scientists be more productive and achieve similar or better results in a shorter period of time. The thesis presents the state of the art techniques, emphasizes the usage of Evolutionary Algorithms(EAs) and proposes an AutoML Pipeline that embodies the concepts presented.

The main contribution is the usage of EAs in AutoML and the creation of a highly customizable framework the allows for fast experimentation and also on-cloud training facilities.

The rest of the thesis is structured as follows: Chapter 1 introduces the topics, while Chapter 2 provides reason about why this is a worth to research topic. Chapters 3 and 4 present the generic Machine Learning process and, respectively, the application of Evolutionary Algorithms in creating Automated Machine Learning. Chapter 5 describes the solution to support the scientifical hypothesis presented in the previous sections, while Chapter 6 proves the performance of such an approach. Lastly, Chapter 7 concludes and sets future directions to follow.

This work is the result of my own activity. I have neither given nor received unauthorized assistance on this work.

Mihai Anton

Cluj-Napoca
June 2020

Contents

List of Figures

1 Introduction

Innovation has always led to great advancements of technology. Be it computers, programming languages, or modern development frameworks, both the research and the technical work need new approaches to previously taught *state of the art* techniques in order to bring novelty to the technological world. The market demand increases and, thus, the offer of computer science in terms of techniques and new software should respond to this tendency. Today, data is everywhere, from websites to sensors, the world is surrounded by data collecting sources, increasing the need of tools to analyse it, draw insights and produce meaningful interpretations, being one of the many factors that gave birth to the field of Machine Learning, one of the most rapidly evolving paradigms of the last decade.

Machine Learning has become so popular that almost every company wants it in their product and almost every programmer tries to apply its techniques to create better data-driven software [Goodfellow et al., 2016]. Since data specific operations and analytics are taking a greater and greater time share of the average software project, developers and researchers are constantly looking for ways to make the process more efficient. This, bit by bit, leads to the need of automating existing techniques and asking computers to take over repetitive tasks that might waste a valuable amount of time. Thus, this is how the idea of Automated Machine Learning came up, aiming to automate as much as possible the trial and error nature of machine learning projects, using different optimization techniques, that are, themselves, subject to analysis and improvement.

Automated Machine Learning is the approach to Machine Learning where repetitive tasks are replaced by error proof algorithms and well known practices are conducted automatically by computers, without active human intervention. The core parts of this approach are Data Engineering and Model Training, where each of those can be fully automated if previously configured by the user. This thesis handles both parts, presenting the available *state of the art* techniques and the contribution that evolutionary algorithms bring into the whole optimization process. On top of the theoretical information provided within the following chapters, the thesis also introduces a technical approach to support the theory, namely an autonomous machine learning framework, able to detect complex models and find optimal configurations for one given dataset.

2 Motivation

Since every new product or proposed algorithm has to solve a real world problem in order to be successful, Automated Machine Learning has the main goal of saving time and lowering the error rate of humans. In the classical machine learning flow, it all starts with raw data. As the data producing sources are not perfect and often not optimized for analytics, the datasets they produce are far from ready to be fed to a learning algorithm. This creates the need for data cleaning and engineering, which usually takes more time than expected. Taking into consideration that the power of any learning algorithm (talking about better scores) relies mostly on the data it receives and less on later optimizations, the step of data/feature engineering has to be consistent and well planned in order to achieve top results [Goodfellow et al., 2016]. Right after the data engineering phase comes the model selection step, which is again, subject to trial and error, since although some tasks have well known model types that work well, this only reduces the search space, but there still leaves space for choice between different approaches. The last step, after a model is chosen, is hyperparameter optimization and learning, which offers some degrees of freedom regarding the parameter choice before the model starts the learning process automatically.

After listing only three of the major steps of a classical machine learning pipeline (raw data to trained model), there are clearly places where automation can help and automatically iterating over the search space can yield better and unexplored configurations of the data engineering algorithm, while also producing surprising model configurations which work better than expected. As statistics show, the time spent on data preparation and hyperparameter selection tasks is as high as 80% of the total project time [Herbrich and Graepel,]. The rest of time is spent on almost automated training of the model. Since there is so much time to be saved, it is worth researching better alternatives to the whole machine learning flow rather than just continuing with the classical approach. Not only are good data scientists scarce, but they are also financially intensive nowadays, so finding a secondary optimal alternative is a priority.

3 Machine Learning Process

Nowadays, when one thinks about machine learning, it's most of the time a classical approach, when it all starts with raw data, followed by some model selection trial and error, hyperparameter tuning and, finally, a phase of model training. At this stage, the data scientist in charge has to wait for the model to be trained, supervising it's evolution and the convergence charts. When considering the long time, hours or days, that takes to autonomously train the model, one does not think about the human hours involved in the pre-training phase, when all the trial and error has been performed and all the alternatives of data preprocessing and feature engineering have been tried.

Simply put, when the project is done, the ratio between the time spent on trial and error tasks in the pre-training phase and the time spent on model training is not alarmingly high in order to trigger the desire of automating the whole process. Taking a closer look, one may realize that the actual time wasted adds up to whole days when measured in the term of a month, and an alternative should be found in order to achieve a more optimal time usage.

For the time being, this is the *state of the art*, safe approach, which most of the time, if all the steps are properly taken, yields the awaited results, which might mean well performing models or insights from the data. While this process works given the prior machine learning related knowledge of the user, it is time consuming and it starts, day by day, to become the old fashioned way of doing machine learning.

As of today, the vast majority of supervised learning tasks fall in two major categories, namely regression and classification tasks. Despite the fact that other fields have an emerging growth (along the already existing exponential data growth), like clustering and anomaly detection, regression and classification are still the bottom line in machine learning. At least when it comes to structured data, businesses and companies more often try to predict unseen samples rather than decide which ones are anomalous or in which specific cluster they fall. Automated Machine Learning is still a rising trend, and might as well cover those in the near future, but for the moment, automating supervised regression and classification tasks already brings a great time saving potential.

Any machine learning algorithm can be abstractly described as an algebraic equation [Goodfellow et al., 2016], which maps some input A to some output B, taking into consideration some transformations X along with some performance metrics M which are aimed to be minimized. The generic task in machine learning is to find the optimal X that, by the metrics M, is the best approximation for the (A,B) data pair (1).

$$\textbf{Generic machine learning hypothesis: } A \times X = B \tag{1}$$

3.1 Data Preparation

The very core of every learning algorithm is data. The more, the better. Lately a lot of discussions have been made on the quantity of data, but less about the quality of it. Experiments show that in order for a learning algorithm to reach its full potential, the data that we feed to it must be as qualitative as it is quantitative. As data is often compared to other valuable assets, often regarded as the oil of the 21st century, I doubt that people would keep being interested in it as they are today if either quality or quantity would ever be absent. In order to achieve *state of the art* results in data science projects, the main material, namely data, has to be ready to be shaped and molded as our particular situation demands. Algorithms that accept data as raw and unprocessed as it is are scarce and often fail to leverage the full potential of machine learning and of the dataset itself.

Condition1	Condition2	BldgType	HouseStyle	OverallQual	OverallCond	YearBuilt	YearRemodAdd
Norm	Norm	1Fam	2Story	7	5	2003	2003
Feedr	Norm	1Fam	1Story	6	8	1976	1976
Norm	Norm	1Fam	2Story	7	5	2001	2002
Norm	Norm	1Fam	2Story	7	5	1915	1970
Norm	Norm	1Fam	2Story	8	5	2000	2000

Figure 1: Raw dataset example.[1]

The road between raw data and the actual training of one model is far from straight and often requires various techniques of data processing in order to reveal insights and to emphasize certain distributions of the features. Take for example the following dataset (Figure 1) from the real estate business. It is by far an easy job to be able to accurately predict the final acquisition price of a house considering only the raw data. There is no way a generic algorithm could make a difference between '1Story' and '2Story' from the HouseStyle column or differentiate between two different value scales, like the year in the YearBuilt column and the mark in the OverallQual column.

Data preparation has the duty of building an adequate value distribution for each column in order that a generic algorithm could actually learn features from it. Some examples are rescaling the values, turning text information into categorical or extracting tokens from continuous string values, like product descriptions. The following chapter will discuss the most common feature engineering techniques.

[1] https://www.kaggle.com/c/house-prices-advanced-regression-techniques/data

3.2 Feature Engineering

Rarely is a raw dataset ready to be fed into a learning algorithm and expected to yield good results. There are lots of feature engineering techniques that have to be applied to a dataset before running an algorithm with it.

3.2.1 Handling missing data

As the data producing sources are rarely perfect, raw datasets have missing values. Since generic algorithms do not have the ability to handle such cases and replacing them with a random value opens the possibility of obtaining any random output, methods have been implemented in order to replace them while keeping the data distribution in place, unmodified [Goodfellow et al., 2016].

The basic solution is dropping the rows or columns that contain an excessive amount of missing fields, since replacing all the empty fields with the same default value might actually bias the model rather than create valuable insights.

When the number of missing values from a column is below a certain percentile, replacing them with a meaningful default value is a fair tradeoff between keeping the size of the dataset and keeping the distributions as in the original state. While this default value can be any choice from the distribution of the current attribute, setting it to the median value has, experimentally, provided the best results. The mean value is also a reasonably good choice but, compared to the median, is more biased by outliers and skewed data distributions, as visually represented in Figure 2.

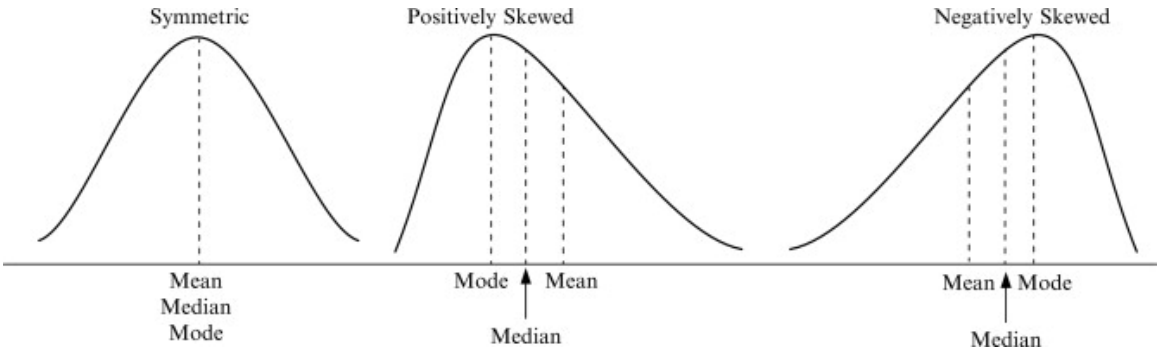

Figure 2: How skewed distributions influence the mean value.[2]

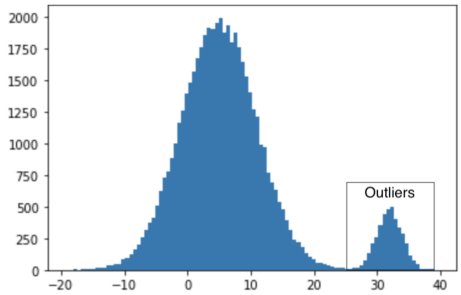

Figure 3: Outliers as seen in a distribution.

3.2.2 Handling outliers

Outliers are data points that lie far away from the majority of data samples in the geometrical space of the distribution (Figure 3). Since these observations are far from the mean, they can influence the learning algorithm in an unwanted way, biasing it towards the tails of the distribution. A common way to handle outliers is Outlier Capping, which limits the range, casting a value X in the range [m - std * delta , m + std * delta], where m is the median value of the distribution, std the standard deviation and delta an arbitrarily chosen scale factor, as in (2).

$$X = \begin{cases} m - std * delta & \text{if } X < m - std * delta \\ m + std * delta & \text{if } X > m + std * delta \\ X & \text{otherwise} \end{cases} \tag{2}$$

3.2.3 Handling categorical data

Often in various business cases data entries lie in a particular category, or have tags assigned to them. This data has a discrete distribution and has a predefined set of categories. Since category names cannot be understood by mathematical estimators, nor can be decided upon which is 'greater' compared to the others, some type of encoding, often related to as "one hot encoding" is performed. This technique converts a column vector N of size (n,1), with k unique values into a matrix M of size (n,k), using rule (3).

$$M[i][category] = \begin{cases} 1 & \text{if } N[i] = category \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

[2]https://www.sciencedirect.com/topics/mathematics/skewed-distribution

This ensures that the data resulted has a simple discrete distribution, with values in [0,1], meaning, from a learning algorithm's perspective, whether a data entry is or is not part of a particular group.

3.2.4 Scaling the data to a common distribution

Learning algorithms do not perform feature scaling and neither should them. They learn from the given features, without assigning any particular weight to them. Thus, if a dataset contains features with different distributions, the model will naturally learn that the higher values matter the most in the decision, just because when compared to others, are on a higher scale.

As an example, if a dataset contains, among others, the age of the customer (with values in [10-80]) and the day of the week in which a transaction is made (with values in [1-7]), the model trained to predict the total amount spent by a particular customer on every transaction develops a tendency to think the result is more influenced by the age rather that the day of the week, which might be proven wrong.

From a statistical point of view, the features fed into learning models should have, as much as possible, the same distribution, lowering this way the risk of biasing the model. There are two widely used techniques, namely normalization and standardization [Goodfellow et al., 2016]. Normalization, or min-max scaling (4), leaves the distribution of values unchanged, scaling them to a [0-1] range. Standardization, or z-score normalization (5), takes the standard deviation in consideration and changes the distribution into one with mean of 0 and standard deviation of 1.

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}} \tag{4}$$

$$X_{std} = \frac{X - mean}{stdev} \tag{5}$$

3.2.5 Creating polynomial features

There is often the case in machine learning when a feature is not linearly separable. Although more complex algorithms cope with the problem of non linearly separable search spaces, they might sacrifice accuracy over covering all the nonlinearities. Thus, creating polynomial features can help learning algorithms separate the search space with more ease, yielding better results in the end.

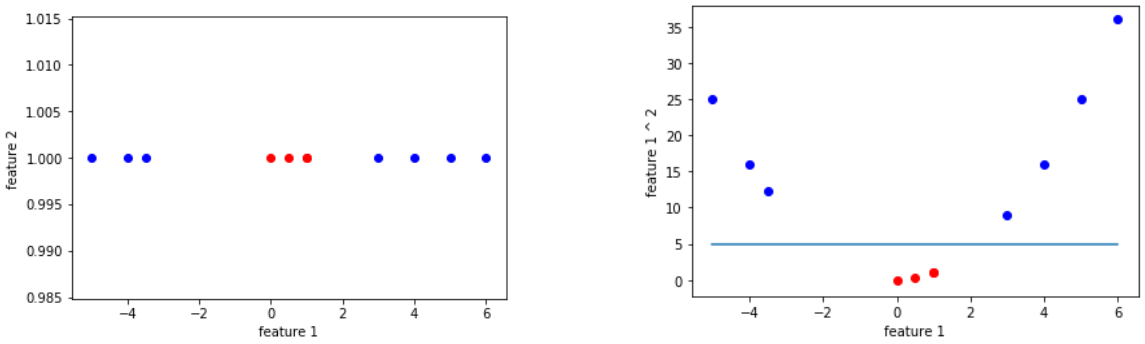

Figure 4: Using polynomial features.

In the example above, the initial dataset (the left part of Figure 4) is hard to linearly separate, considering that the goal is to correctly classify the red dots from the blue ones, since all the points lie on the same axis Y=1. Generating the second degree polynomial of feature 1 and adding it to the dataset yields a better representation of the geometrical data space, thus making it easier to be split. Although this is a shallow example, it clearly illustrates the importance of polynomial features in machine learning. On a large scale, polynomials of multi-feature combinations are taken into consideration for generating even more insights from the data.

3.2.6 Handling text data

Besides numeric attributes, datasets often contain continuous text information, which if most of the times unstructured, of undetermined length and in a non standardized format. The main instinct is to remove these feature columns, for simplicity, but they might actually be a very important factor in whether a further algorithm decision is handled properly. Also, making the most of the available features instead of dropping them might be a better choice.

The techniques used to handle this type of features have their roots in NLP (Natural Language Processing). The goal is to group sets of related words, named tokens, into one simple root, named lemma. This process is often known as lemmatization and solves the problem of reducing unstructured data into common, standardized tokens. Having the data processed in a standardized format, a process similar to one-hot-encoding occurs, similar to the way categorical features are handled. Having a text feature vector N of size (n,1) with k different lemmas aggregated over all rows, a new binary vector M is created, following the rule 6.

$$M[i][token] = \begin{cases} 1 & \text{if } token \in N[i] \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

This process determines the learning algorithm to sense even the smallest sign that a word might represent in a sentence and enables a more complete and comprehensive understanding of the data and the factors that influence results.

3.3 Model Choice

As science evolves, more and more emerging techniques replace the old *state of the art* approaches. Although new ways of doing machine learning come up every day, the models used by the majority remain the same. This chapter will briefly go over the most popular model choices that solve a wide range of problems. Even though the rest of the thesis will mainly cover neural networks, since a deep learning approach to automated machine learning is proposed, some other models are still worth the attention.

3.3.1 The perceptron

The oldest and most basic form of computer intelligence is the perceptron. Inspired by linear algebra, consisting mainly of a linear combination between the input and the weights (Figure 5), this model mimics the basic unit of the human brain, the neuron. Provided some input, the perceptron weights every feature, adds a bias and compares the result to a threshold. If above, it outputs 1; otherwise, it outputs 0. As [Rosenblatt, 1958] stated, a perceptron is a system based on rather probabilistic than deterministic principles and gains its reliability from statistical measures over large datasets. This model had been used widely at the beginning of AI research, until scientists found out that it could only solve linearly separable problems. Tested with the XOR problem, it could not properly separate the True cases from the False cases, due to the problem of non-linear separable search spaces described at 3.2.5.

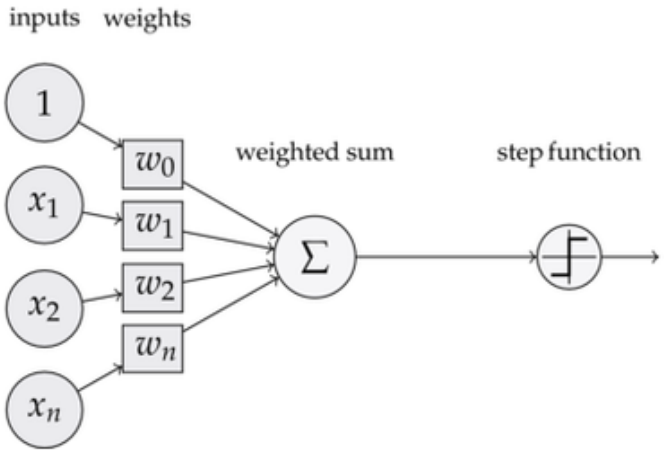

Figure 5: The perceptron.[3]

3.3.2 Decision trees

More commonly used in classification tasks, decision trees are a simple, self explanatory model, capable of reasoning the choices that led to a particular data decomposition. Due to the fact that features have the property of being split into groups, be them categorical or range based, this model leverages the capability of gradually splitting the dataset by a chosen feature until the granularity of the leaf nodes is low enough. The performance metric often used to measure the success of a decision tree is entropy [Herbrich and Graepel,].

Entropy is a measure of disorder in data. The higher the value, the less organized the dataset is. The main responsibility of decision trees is to split the data in a way that will reduce the entropy for each recurrent subtree. The aim is to be able to separate data in such a way that each leaf will contain samples from one class only.

Decision trees can easily do that, exhaustively splitting the data until each leaf contains one entry, but this causes a well known problem in machine learning, namely overfitting. Overfitting occurs when a model learns all the peculiarities of one dataset until it is able to perfectly reproduce the outputs of the data it has been trained on. When faced with unseen data, it will poorly be able to generalize. Thus, decision trees are a powerful model when balancing both the power of reducing entropy while maintaining the ability to generalize on unseen data.

[3] https://blog.knoldus.com/introduction-to-perceptron-neural-network/

3.3.3 Support vector machines

Also highly used in classification tasks, Support Vector Machines (or SVMs), are an algorithm that, given data, finds the best fitting hyperplane that accurately classifies the samples [Tang, 2013]. In other words, SVMs try to create as wide margins as possible between samples of different classes, in order to create a clear delimitation between them. If the dataset is 2 dimensional, the result generated will be a line or curve. If it is 3 dimensional, then a plane will be generated. For a higher dimension N a (N-1) dimensional hyperplane is generated, with the aim of maximizing the distance between the hyperplane and the closest points in the dataset, called support vectors.

As in 3.2.5, SVMs are also prone to poor ability of splitting non-linear separable features, thus yielding unwanted results. What fully enables the power of this model is the method called 'The Kernel Trick' it uses. Similar to what the polynomial features did, the kernel used in SVMs maps a non-linear separable data into a higher dimension where a hyperplane can be used to linearly separate it.

3.3.4 Neural networks

Having mentioned the Rosenblatt's perceptron (3.3.1), which is still in use today, it is the right time to introduce a model that has the perceptron as its building block and which is capable of learning any arbitrarily chosen function provided enough data and time, as often stated in research. Neural networks are powerful, easy to build and scale, yet complicated to train. They are widely used nowadays in various tasks, from the basic regression and classification tasks to time series prediction, image generation and autonomous driving. They are incredibly powerful if chosen with proper architecture, fed the right data and trained with the optimal parameters.

Neural networks, also named multi layered perceptrons are, in a nutshell, a list of stacked perceptrons, organized in layers (Figure 6). Each receives signals from all the neurons in the previous layer, sums up a linear combination, adds a bias and passes it through an activation function to all the neurons in the next layer, if any. All neurons in each layer L_i are connected to each neuron in the layer L_{i+1}, hence the name of fully connected neural networks.

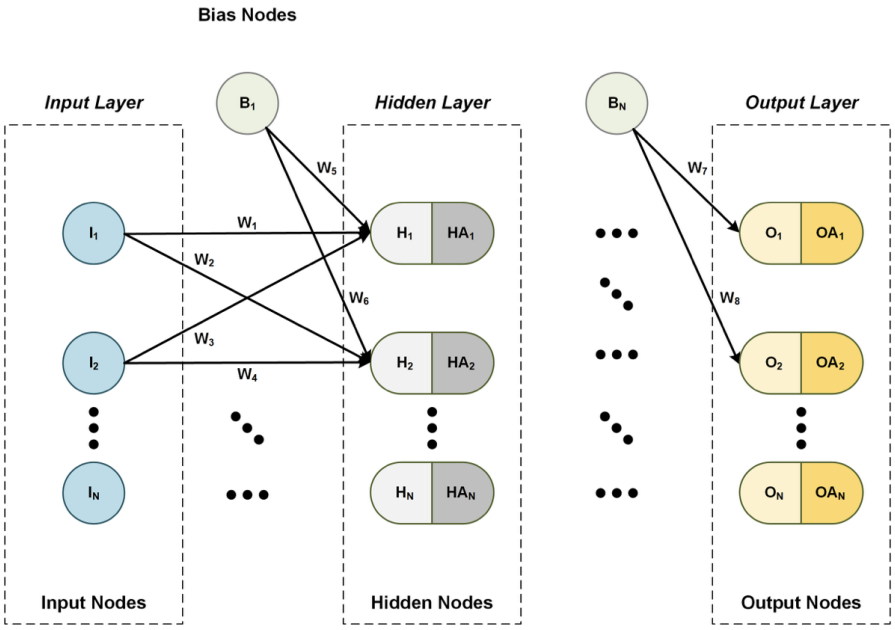

Figure 6: The architecture of a neural network.[4]

Often related to as 'feed-forward' neural networks, these models have a simple to understand training flow. Having a train set D(x,y), a validation set $D_{val}(x_{val},y_{val})$, a set of weights W, a set of biases B and a loss function L, a complete learning cycle consists of the following phases. The data is forwarded through the network, according to (7), then an error E is computed in the output layer (8). Having E and the activated value of each neuron j in the layer i as a_{ij}, E is distributed backwards through the network in a process called backpropagation of errors, each weight and bias receiving its own share of the error, according to (9), depending on the influence they had over E. The goal is to create the partial derivative of each weight and bias with respect to the cost function (9), then, through gradient descend, update them according to the rule (10), scaled by a factor λ, named learning rate, measuring the convergence speed of the model towards a minimum.

$$\begin{aligned} z_i^l &= \sum_{j=0}^{n} (w_{ij}^{l-1} \times a_j^{l-1}) + b_i^l \\ a_i^l &= activation(z_i^l) \end{aligned} \tag{7}$$

$$E = L(x_{val}, y_{val}) \tag{8}$$

[4]https://dzone.com/articles/the-very-basic-introduction-to-feed-forward-neural

Having the error in the layer l+1, the inner error (before activation), of the ith neuron in layer l is:

$$\begin{aligned} \delta n_i^l &= activation'(z_i^l) \cdot \sum_{j=0}^{n} (w_{ji}^{l+1} \cdot \frac{\delta E}{\delta n_j^{l+1}}) \\ \delta w_{ij}^l &= a_j^{l-1} \cdot \delta n_i^l \\ \delta b_i^l &= \delta n_i^l \end{aligned} \tag{9}$$

$$\begin{aligned} w_{ij}^l &= w_{ij}^l - \lambda \cdot \delta w_{ij}^l \\ b_i^l &= b_i^l - \lambda \cdot \delta b_i^l \end{aligned} \tag{10}$$

Although the math behind training a neural network might seem complicated, it is all reduced to simple matrix multiplications. Thus, taking into account the optimizations for such operations and the ability to massively parallelize the task, neural network learning is getting faster with the advancements in technology, especially in GPUs and TPUs [Jouppi et al., 2017], which have experimentally proven to be very effective in training neural networks.

As simple as it seems, when faced with real data, training neural networks can often lead to unexpected problems. Poorly choosing activation functions and learning rates can produce one of the two most common problems, vanishing gradients and exploding gradients [Goodfellow et al., 2016]. While the first one makes convergence slow and usually prohibits the incipient layers of the network from learning, the latter produces overflow and often ∞ or NaN results.

3.4 Hyperparameter Tuning

Any model solves its particular problem and offers a mathematical abstraction over the complex operations that happen behind the scenes. Although models generally simplify the work a data scientist has to do, often one line of code being required to fit a model, they rarely yield the expected results using the out of the box configuration. Models have many parameters and can be tuned in a multitude of ways, leading to search spaces with hundreds, maybe thousands of possible combinations. The duty of the user is generally to find the best hyperparameter combination in order to obtain the best result. This is rarely an easy choice and usually is very time consuming. It is a step that is prone to time consuming trial and error, as further described in 3.6.

Hyperparameter are the choices made in the construction of one model before the training phase starts, as [Yu and Zhu, 2020] emphasis. They can't be automatically learned by the model and, what is more important, the performance of the training itself is highly influenced by the quality of choices regarding hyperparameter tuning. This is usually a hard problem, without having a unique solution. In choosing a good value for one parameter, one may make sacrifices regarding others. This means that the values of hyperparameters are highly coupled, a change in one triggering a change in others as well. The hyperparameter tuning problem is often subject to the No Free Lunch Theorem, that states that there is no general optimization method superior to others for all optimization problems [Serafino, 2013]. On a very high level this theorem states that in hyperparameter tuning as well as in other fields, a configuration that works well on a specific problem might be inferior to others when evaluated with other metrics. Thus, there is no unique way of setting hyperparameters, each method bringing its advantages as well as disadvantages. A very skilled data scientist might already know a set of patterns that work well on specific problems, and might start with them as a base for further optimizations.

Since HPO (HyperParameter Optimization) has often the highest influence in the final result, along with the quality of data, many have considered approaches to automate this process and converge to optimal configurations. [Snoek et al., 2012] proposes a Bayesian Optimization approach to finding the best configuration, converging steadily towards the statistically best known solution. [Wei et al., 2019] proposes recording the best configurations that worked on previous datasets and when a new task comes, the configuration used would be the best one used on the most similar recorded dataset, thus leveraging the power of previous time consuming searches.

3.5 Training

Having chosen the proper configuration of hyperparameters, the next phase is the training of the model. The best practice is splitting the data into training, validation and test sets. Experimentally, the best performing split ratio of a dataset D into D_{train}, D_{test} and D_{val} is roughly 70%, 20% and 10%. The train set is used for optimizing the models weights, while the validation is used for measuring the impact of HPO and HP (HyperParameter) changes in consecutive runs. The test set is used to score the model after a complete training session has been performed.

The objective in a training session is loss minimization. The loss function L measures how well does the model learn the data given and also offers the 'feedback' needed, in terms of the derivative of each parameter p (11). Using the feedback from the loss function, the model is expected to converge to a global optimum, provided that the loss function is convex. Non convex loss functions often fail to converge to global

optimum and get stuck in local areas, surrounded by a positive gradient area in the close proximity, as described in Figure 7.

$$\frac{\delta L}{\delta p} \tag{11}$$

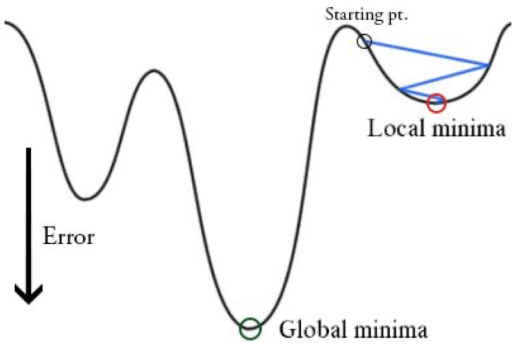

Figure 7: Non-convex function.[5]

3.6 The time consuming nature of trial and error

Trial and error happens in every project, more often when new approaches are tried or when there does not exist an universal solution that solves all the problems, most often referred to as an *One Size Fits All.*. It is inevitable and all data scientists struggle with it [Gaveau et al., 2017], being, after all, the root of all new findings and research topics. No two datasets are identical and every new one comes with its own peculiarities and uncommon distributions. Judging from the perspective of the Butterfly Effect Theory [Ghys, 2015], two similar but not identical datasets can yield two opposing, highly different solutions, thus there is no clearly defined pattern on which exact approach to take in solving data science problems. Trial and error is a crucial step in the current definition of a successful machine learning project and is considered to be a mandatory step for data scientists to take in order to achieve the desired results. Converting to an automated approach, where a computer takes on the task of trying and failing as many times as necessary, might be a big leap towards a new approach to data science and as the paper further discusses, a switch that is worth pursuing.

After all, the goal is to minimize the time between the idea and the product, with optimal results. In some cases, when time to market is an important parameter, one

[5]https://www.cs.ubc.ca/labs/lci/mlrg/slides/non_convex_optimization.pdf

may decide that near-optimal performance outweighs the long time spent on creating the perfect product. Being more clear, instead of wasting 100% of the time to presumably achieve 100% of the results, one may decide to wisely use only 20% of that time in order to get 80% of the results, gaining the time advantage. Again, the Pareto principle applies here as well, stating that more often the most consistent part of a project is built in the first quintile of the time, while the rest is being used to cover all the little details and often consists of a much unproductive work compared to the first 20% of the time. Automated Machine Learning is here to assist, if not completely replace the work of finding the little details and covering all the unfilled corners of the search space until the objective is met.

4 Automated Machine Learning

Automated machine learning is the subfield of Machine Learning responsible for self improving and automatically searching for the best HP (HyperParameter) configuration and model, including training various models and deciding which is the best performing.

4.1 Definition and Description

Automated machine learning is clearly an optimization challenge. There is no well known algorithm or a *one size fits all* approach to solve all learning problems. Various configurations have to be searched in order to decide which one is performing best. Taking into account that data scientists are not dealing with a finite and continuous search space, but rather an infinite and non convex one, heuristics have to be developed in order to find a configuration that is tangent to the best solution. A convex search space is a mathematical space where the solution converges to a global optimum, since there is just one such point. A non-convex space is similar to a hilly region, where there are many local optimums (individual hills), and just one of those is the global optimum (the peak among the hills). Since the landscape or the theoretically named search space is infinite, heuristics have to be developed with the goal of finding a reasonably good optimum in the least amount of time. These algorithms come in different forms, from conventional search methods, like grid or random search, or more advanced methods inspired by nature, namely Bayesian optimizations and evolutionary algorithms. Although all stated above will be covered in the following chapters, the proposed solution consists of optimizing machine learning pipelines using evolutionary algorithms.

In their paper, [Feurer et al., 2015] see AutoML as a CASH problem (Combined Algorithm Selection and Hyperparameter Optimization). It is defined as the pursuit of algorithm $A \in \mathscr{A}$ and its hyperparameter configuration $C \in \mathscr{C}$ that, trained and tested over an evenly distributed train and validation set $D_{train}(x_{train}, y_{train})$, $D_{val}(x_{val}, y_{val})$ minimizes the loss function L, as in (12).

$$A^*, C^* \in \underset{A^i \in \mathscr{A}, C^j \in \mathscr{C}}{\operatorname{argmin}} \; L(A^i{}_{C^j}, D_{\text{train}}, D_{\text{val}}) \tag{12}$$

4.2 Evolutionary Algorithms

Evolutionary algorithms are inspired by the Darwinian theory that evolution is subject to a set of primal operations as the survival of the fittest, genome crossover and random mutation as a result of adapting to the environment. They also rely on the fact that small improvements in the genome of individuals can, over time, using the technique of survival of the fittest, lead to greater advancements of the species. Being fit in such a population is highly subjective and is a function of the environment of the specific species. Generally a fitter individual is better in the environment in which he lives and at the tasks he is supposed to carry out. Gradually eliminating less fit members from the population and allowing, with a greater probability, the fittest to reproduce, will, over a number of epochs (the time unit in evolutionary algorithms), lead to an overall fitter population.

Using evolutionary algorithms in automated machine learning leverages the power of both evolution and statistics, since they mimic the trial and error performed by data scientists when approaching a data science project, and also follow the statistical fitter individuals, always pursuing the search in places where it is known that something is to be found. Also, they do not hesitate to explore new paths from time to time, leading to the well known exploration-exploitation dilemma, which the paper will return to in the following chapters.

Looking at the flow of evolutionary algorithms (Figure 8), the approach seems natural, since, as human beings, we are accustomed to it. The flow is a simple way of searching through the tightest corners of the search space and providing untaught configurations. Provided enough time and the right crossover, mutation and selection logic, evolutionary algorithms are able to converge to global optimums and yield the awaited result. Their advantage is that the speed of search is tangential to the speed of light and they do not make mistakes as humans often do.

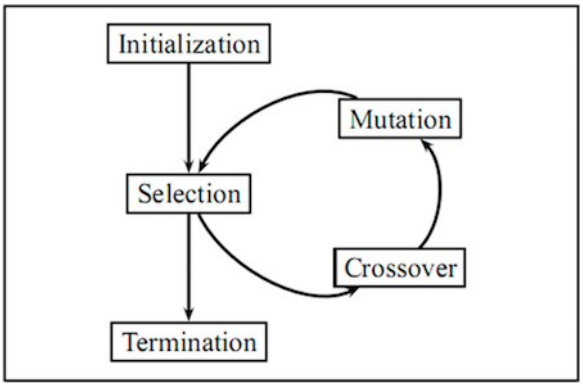

Figure 8: The steps of evolutionary algorithms.[6]

Using the concepts of evolutionary algorithms in automated machine learning, one can search through the configuration space more efficiently, finding gradually better methods. Genomes can be considered as different configurations and their fitness is some metric on the dataset, combined with other metrics regarding the training phase, such as convergence rate or the derivative of loss drop over time. Data engineering is also subject to evolutionary optimisations, since the techniques of generating features can be combined in various ways in order to extract the most valuable information from the dataset. Both feature engineering and model training using evolutionary algorithms can lead to interesting results, subject to further analysis by data scientists or artificial intelligence enthusiasts.

Due to the complexity of converting all the machine learning into an evolutionary pipeline, the paper will only cover the usage of EAs in Deep Learning, while offering information and paths to follow the bigger goal.

As also mentioned previously in this section, EAs have some basic operations that guarantee the coverage of the search space provided enough time, namely random initialization, selection, cross over and mutation. In the proposed solution, those 4 operations help the Neural Architecture Search in the following ways:

- **Random Initialization**
  Neural networks have a few parameters available for tuning, some of them being the number of layers and the count of neurons in each, activation functions, dropout rates and the learning rate. During the initialization phase, the specifications of each neural network that is being built are randomly selected from the available pool, be it a list of predefined choices or a continuous interval. The randomness provides diversity in the population.

- **Selection**
  At each epoch, two chromosomes are randomly selected for crossover. Multiple selection procedures exist, each with empirically proven good performances. To name a few, random selection selects a random chromosome, tournament selection samples a k-sized random population and returns the best from it and roulette selection, which randomly selects individuals with a probability directly proportional to their fitness, as in (13) (i.e. the fitter a chromosome is, the higher the chance for selection). For the proposed solution, the latter is used since it provided the best performances.

$$p_{\mathrm{i}} = \frac{fitness_{\mathrm{i}}}{\sum_{j=1}^{n} fitness_{\mathrm{j}}} \tag{13}$$

[6]https://towardsdatascience.com/introduction-to-evolutionary-algorithms-a8594b484ac

- **Cross Over**
  After 2 chromosomes are sampled from the population, a cross over is performed, also known as an XO operator, providing an offspring (*off*). In the proposed solution, for each property of the neural network, the offspring either inherits it from one of its parents with a given probability, or creates a combination of both (e.g. the average for the learning rate).

- **Mutation**
  After the offspring *off* is generated , the mutation operator $\mathscr{M}$ is applied in order to slightly transform its properties. The transformation $\mathscr{M}(off)$ yields the mutated offspring off^{x}. In the solution described in the next chapter, the mutation consists of random modifications in continuous parameters or random switches in discrete ones.

4.3 Advantages

Various techniques have become the *state of the art* way of doing machine learning nowadays. To name a few, every data scientist handles missing data, categorical data is almost always encoded and text fields are rarely fed as raw as they are into learning algorithms. Also, experienced practitioners already know a well behaving learning rate and momentum to use, and are already accustomed to the fact that Mean Squared Error should be the starting point when fitting a regression dataset.

This paper does not aim to defeat these standard practices and replace them with radical new ones, but to challenge the status quo in machine learning and help researchers achieve more in a lower span of time.

AutoML is a new field, with relevant research starting around 2012, when AutoWEKA was the first automated machine learning proposal to enter the market. As [Thornton et al., 2012] explain, their approach was towards fully automating the machine learning flow using Bayesian optimizations and a predefined pool of models and other methods, having the 'why' of helping non-experts leverage the power of machine learning in their projects, with improved performances.

Since the beginning of the informational era, automation has played a huge role in the development of emerging industries. Machine learning has been proven effective in the last few years and nowadays has become standard in data-driven companies. Since both automation and machine learning have brought impressive results in technology, combining them will further improve the outcomes of data science related projects. The main advantages of AutoML is the time it saves and the insights of the data it provides. One one hand, AutoML is a solution that does not have to be supervised,

thus is able to run for days without people having to check the progress or to make corrections. On the other hand, the time spent in searching the configuration space yields interesting model configurations (as discussed later in the results section), analyzing those bringing the possibility of switching directions or taking new decisions.

4.4 State Of The Art

Machine learning has developed a pretty vast field of tools and techniques to be used by data scientists. There are many good approaches to a related project and many techniques compete to achieve the best results. Trying to automate this process requires using all the *state of the art* techniques in an intelligent way in order to achieve their full potential.

[He et al., 2019] mention various AutoML techniques in their survey, proving once more that the challenge of building a *state of the art* Automated Machine Learning system is still a hot research topic, with lots of paths to pursue. Among others, they mention as core techniques the necessity of transforming feature engineering in an automated routine and then using different search techniques for model training, starting from the most basic ones, namely grid and random search to the more advances Bayesian Optimizations (BO), Reinforcement Learning (RL) and Evolutionary Algorithms (EA).

4.5 Automated Feature Engineering

As often stated in data science related articles, the performance of one machine learning algorithm is as good as the data it is exposed to. Raw data is rarely ready to be fed to an algorithm and even if the data formats might be directly usable (i.e. all the features are numeric and no NaNs are present), deep insights from the data might be hidden by different distributions of variables. Different techniques are related as the *state of the art* nowadays in feature engineering, as normalization, standardization or generating polynomial features. As [Abadi et al., 2015] states, those operations are subject to tedious work that often takes a longer than expected period. As more and more articles show, the data processing phase of a data science project is the most time consuming, taking up to 80% of the total time.

In order to limit time waste, various automated techniques emerge to increase project speed and to leverage the power of automation. Thus, all the human feature engineering gradually becomes automated feature engineering. The most basic flow is to implement all the *state of the art* techniques over all features and either let the chosen

algorithm automatically decide which features are worth putting higher weights on or make use of Principal Component Analysis [Gewers et al., 2018] in order to reduce the dimensionality of data and increase training speed.

4.6 Neural Architecture Search (NAS)

Although the feature engineering phase of the AutoML flow relies on some standard practices which have empirically proven effective, searching a good model to fit them might be a more tedious searching task. Usually, given a dataset, the model choice seems easy at the first sight, since patterns have already emerged which model to choose for specific tasks.

What is not that simple is the hyper-parameter choices made for one model, since taking into account the variations of each parameter the search space can become large enough to rule out the possibility of exploring it fully. Such parameters can be the number of layers in neural networks, the optimizer used, learning rates or the kernel used in a SVM classifier. The problem at hand is known as HPO (HyperParameter Optimization), as [Snoek et al., 2012] better explain.

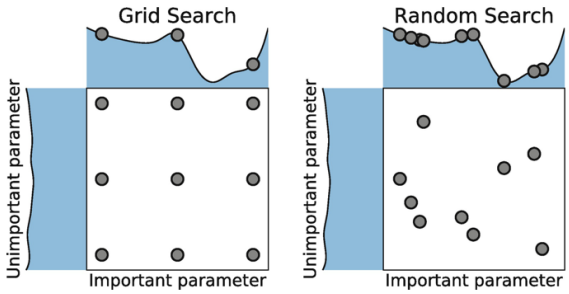

Figure 9: Grid and random search.[7]

Due to the vast search space and the non-convex nature of NAS (Neural Architecture Search), some methods have been proven effective in the last years. A brief presentation of the best will follow while a more complex survey can be found in the paper of [Elsken et al., 2018].

To start with, the 2 most simple ways of searching a non-convex space are grid search and random search (Figure 9). Although they reveal some of the hidden niches of the search space, they do not take advantage of previous passes, thus not leveraging

[7] https://medium.com/@cjl2fv/an-intro-to-hyper-parameter-optimization-using-grid-search-and-random-search-d73b9834ca0a

historical findings. As [Yao et al., 2018] also point out, simple search methods do not take advantage of past iterations.

Converting from uninformed search methods to informed ones, research has shifted AutoML to the fields of Reinforcement Learning (RL), Bayesian Optimization (BO) and Evolutionary Algorithms (EA). Those 3 techniques are the most used in today's trials of fully automating the process of learning [Elsken et al., 2018]. Due to the complexity of the problem to be solved, each search method from the above stated has good and bad parts. BO takes the advantage of the statistical measures of the previously searched space, RL mimics agent learning and EA leverages the power of evolution.

Designing NAS as a Reinforcement Learning problem might seem a straightforward task. The environment is the dataset itself, the actions taken result in new model configurations and the reward is the performance on the validation dataset. Many research papers discuss approaching NAS using RL; for example, [Baker et al., 2016] use Q-learning to train an image classifier using CNNs. Due to the fact that this approach does not enable agents to observe the external state during the sequential process, [Elsken et al., 2018] names this problem as a *stateless multi-armed bandit problem*.

Bayesian Optimizations(BO) rely on statistical models to determine the next point to sample [Snoek et al., 2012]. It is used when the objective function is difficult to evaluate and often requires large amounts of time to compute, as in the Automated Machine Learning case. It enables the model to make use of the previous trials and to have a tendency towards the best known region of the non-convex search space.

Evolutionary Algorithms(EA), as also mentioned in the introduction of this chapter, mimic the way nature converges towards better populations, combining and replacing individuals gradually in order to create fitter members. The approach presented in this paper uses EA to find the minimum of the non-convex space of deep learning models, as described in 4.2. EAs consist of a simple flow, where each time unit is an epoch. During an epoch, depending on the implementation, one or more genomes are created as a result of the crossover between 2 members of the population. The offspring created is subject to some random mutations and then, depending on the specific implementation, is appended to the population or it replaces the least fit member of it, thus maintaining the population size constant over time.

4.7 Using Evolutionary Algorithms in AutoML

Although [He et al., 2019] present a series of successful frameworks that operate using BO and fewer that use EA, the approach in this paper presents top results with benchmark datasets using the latter method. Along with the top results, some inter-

esting statistics have resulted from the evolutionary flow, namely interesting and worth researching deep learning model architectures, that do not respect the conventional way of building them.

Evolutionary algorithms have proven effective in multiple research scenarios. Introducing them in the Machine Learning field is expected to help searching the vast, no-convex space. One main problem when searching these spaces is the exploration-exploitation dilemma. Although it makes common sense exploring the 'wealthy' regions of the search space, other, more rewarding niches remain untouched. Evolutionary Algorithms find the proper trade-off between whether to exploit known regions or to explore new ones, yielding, as also mentioned above, results that would have not been found otherwise.

4.8 Automated Model Training

The leap towards enabling computers to make their own decision regarding model training might seem untrusty, but, indeed, empirical tests show that various model architecture insights are released by autonomous training. In order to make Machine Learning truly Automated, models have to be trained by themselves.

The task of automatically training chosen models is simple, since it only requires the already decided model together with it's configuration and the desired training time or epoch count. Afterwards, models are waited to provide results in order to decide whether or not they yield good performances

4.9 The No Free Lunch Theory

As also mentioned above, finding the optimal model choices inside an Automated Machine Learning flow is subject to trial and error combined with previously learnt statistics. It is worth mentioning again that the No Free Lunch Theory states that there is no universal algorithm superior to all other algorithms when challenged over all the possible optimization problems, as also described by [Wolpert and Macready, 1997].

This abstractly explains that, since the problem solved by AutoML is highly dynamic, no universal solution is able to cover all the variations. If one particular model or AutoML configuration solves one very niched problem very well, it does not mean that this model can be replicated and used on different problems, since the performance can be poor. Thus, it is hard to rank possible configurations or to name one as being the best. Automated Machine Learning exists in the first place to find new ways of solving problems without being explicitly programmed.

4.10 The Exploration-Eploitation Dilemma

In Automated Machine Learning as well as in real life, there might be some patterns that bring wealth and others that don't. As individuals, we tend to stay away from the latter and to converge towards the former. Often, people get comfortable enough with one specific pattern that goes well that they do not bother to explore other variations.

The same happens in Automated Machine Learning as well. Somehow, AutoML frameworks have to leverage both the power of mining known well performing regions (exploitation), but also the diversity that looking for better alternatives yields (exploration).

5 An Automated Machine Learning Pipeline

The practical work developed to support the previously mentioned research idea is an Automated Machine Learning framework that allows fast experimentation of different models and autonomously finding optimum learning methods. The 2 major steps of the pipeline that I have developed are the data engineering phase and the learning phase. The first one produces valid and optimized inputs for learning algorithms while the second takes as input a dataset and creates the best algorithm that accurately turns the independent variables into usable models.

The whole pipeline is configurable through a configuration file for ease of use. The goal is to produce a system that allows both highly skilled data scientists and engineers with lower skills in machine learning to experiment different configurations without explicitly coding them, in order to bridge the gap between machine learning ideas and real prediction models and save time.

Starting with raw data, the pipeline is able to do all the available feature engineering tasks. Beginning with data cleaning, polynomial features generation, one hot encoding and normalization, the first step is able to produce valid inputs for learning algorithms. As future work this step will also incorporate a Principal Component Analysis phase.

The second step is the place where the real optimization and research starts to show its strengths. The first option to customize is to choose a default model from the currently available Random Forest, SVM and Neural networks, which can each be customized in order to experiment with different approaches. The second choice is the evolutionary search option, which as explained above in the research part, cycles through different configurations and gradually converges to global optimums.

Since the complexity of building such a system is very high, I have chosen to focus the most part of my work on using evolutionary algorithms on optimising neural networks for machine learning problems. Although this already is complex and can solve many real world problems, I consider it to be a solid proof of concept for further development using ensemble learning or blending models together for more advanced results. The difficulty of putting many models together is that since the training algorithm of each varies a lot, it is hard to find a common evaluation function that produces consistent metrics for all the models.

5.1 Description

The initial goal was to be able to train models faster, from raw data to usable model files. Since the available frameworks were scarce and not easy to use, there was a great demand for a framework that would leverage the power of machine learning through simple configuration files and highly visual results. The Automated Machine Learning framework, which would also be referred to as 'the pipeline' through the next chapters, manages the complete flow, from a raw dataset to a train pipeline model, while remembering the choices made for further predictions with new data.

A problem often met by data scientists is the reusability of a model, meaning they should be able to use a trained model with ease even after training. The data engineering step is where the problem appears in the first place. Data scientists analyze the dataset, decide on which features to keep and which to engineer, along with the proper engineering methods chosen. They usually write a script that contains the transformations F over the dataset D, which solves the training problem, meaning that the data can enter the training phase as desired by the users. The problem comes after the training has been done, when new data comes to be predicted. Suppose the dataset D', which is as raw as D was before the transformations occurred to it. D' is not able to enter the model since it may have a different number of columns, missing features and different data distributions. D' has to pass through the transformations F in the first place before the model is able to make any prediction, as was the case with dataset D before the training phase.

This iteration loop produces problems especially for people new to machine learning, since when they are satisfied with the performance of the model, they rarely remember to save the data processing methods used or the script used to automatically do them. When they realize that they want to use the model in a real prediction use case, they find it impossible due to the fact that they don't have the necessary mappings to transform the data.

The pipeline described in this chapter solves the problem of remembering the data processing steps made, along with every value and categorical arrays associated with them, thus it becomes an easy to use tool for prototyping data science projects or as an end-to-end solution for programmers with little knowledge in machine learning. The pipeline has some flows, which will be described in later chapters, but, from a bird's eye view, it is able to do 2 main tasks:

- Given a dataset D and a new Pipeline object along with a configuration file, it does automatically all the data processing steps configured, searches for the best model as specified in the configuration file and trains the best one for the time the user passes as an argument. Also, along this process, it remembers all the steps taken

along the way in a recursive data structure, described later as a Mapper, so they are reproducible in future uses.

- Given a dataset D' and a saved Pipeline object, previously fitted on a dataset similar in attributes and distributions as D', it is able to transform D' with the exact same steps used in training and to predict using the model it has learnt previously.

5.2 Architecture

Designing the whole framework was a quite complex challenge, since Automated Machine Learning is a new field in computer science and few complete, end-to-end solutions are available. The main goal was to create a flexible architecture, with few dependencies, so it allows for complex customization from the user without the need of writing code. The pipeline is composed of two main cells, the Data Processor and the Learner, each handling independent tasks, as in Figure 10.

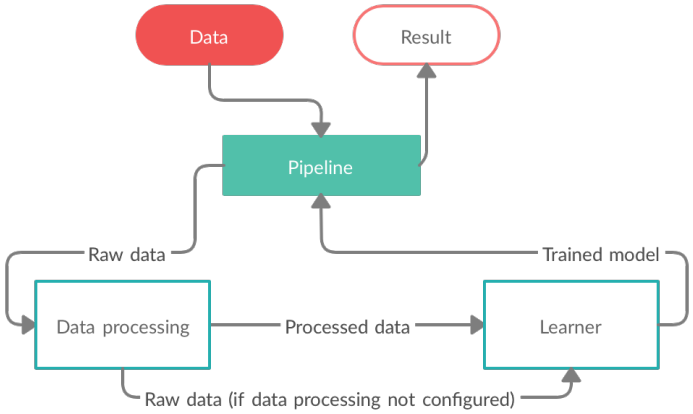

Figure 10: The basic pipeline flow.

Both the Data Processing and the Learner modules have independent submodules, all the system acting like a recurring pipeline, with linked modules that are responsible for independent tasks. The configuration file, also a multilevel recurring JSON document, is handed to the in charge component, based on which it makes some low level decisions. Each module has 2 main tasks: perform changes on the input or pass the result unchanged to the next component. The whole architecture is tree based, meaning that components act as individual roots when passing responsibility to the children, and as recurring child nodes when passing the result to the parent node. Figure 10 is the actual visualization of the first 2 levels in the application structure, but can be seen as a general pattern for how each subtree is built and how the flow works.

Fitting the pipeline is passing through the tree of independent tasks in a DFS (depth first search) manner, handing a task to each child and waiting for a response. Using this kind of architecture makes it very easy to control the small pieces of the puzzle without compromising the whole system. It is also easy to define configuration files, since the recurring nature of this way of constructing software allows building the bigger picture by taking small individual steps rather than building it as a whole from the beginning. This modular approach, often related to as Micro Services allows for great control and low coupling between modules. Thus, it allows building large infrastructures using granular steps rather than big leaps.

As explained in 5.1, the specific problem this architecture is solving requires active memorization of changes that occur, so later those can be applied as well in new scenarios. Building a modular architecture is also a great starting point for creating a convenient saving data structure. Although it is very similar to a normal key-value dictionary, the Mapper (the custom structure for saving states in the flow), acts as a recurring structure that is able to save (key, value) pairs as well as recurring Mappers, making it ideal for the generic architecture pattern. Using this component, each module can map it's own data, as well as collecting the Mapper objects each sub-module creates and mapping them accordingly. This recurring collection of data is useful since it allows for separation of concerns, where separate parts of the application do not have the duty of understanding how other components process and save data. Moreover, this particular structure leverages the power of Python dictionaries, which can be easily saved as binary or JSON files, making them universal and portable, one of the key goals of the Pipeline as a whole.

Since modern businesses demand, the architecture of the Data Engineer in particular, if not of the whole Pipeline in general, is data driven, meaning that data structures named Data Frames (from the popular Python library "Pandas") flow between modules. In a nutshell, each component (seen abstract as a module, but it can be a function or a group of instructions), receives a dataset in the form specified above, does changes over it and returns the result in the same form, creating, thus, an uniform communication layer, with standardized structures instead of mixing different ones.

Moving to the Learner module, the architecture perspective changes a bit since the number of granular instructions decreases. The generic input for the Learner is a dataset (assumed to be ready to enter a learning algorithm), along with the configuration dictionary associated with this part of the infrastructure. The main goal of this module is to identify the needs from the configuration, select a model, tune its hyperparameters and proceed to training it, returning the trained model at the end of the session.

For simplicity of the flow, a few architecture decisions are worth mentioning. Since model configuration can get complex, with tens of variables to be tuned, the Factory design pattern came in handy. Moving the model creation logic into a Factory re-

duced the errors and made the infrastructure work more smoothly and definitely easier to understand. Also, aggregating the common model methods into an AbstractModel class made it easier to manipulate objects around the Pipeline, leveraging the adaptability of the Strategy design pattern. The model factory builds each model according to the specifications and returns an abstract model instance, which, among others, has two main functionalities: train() and predict().

Inspired by the powerful machine learning Tensorflow [Abadi et al., 2015], callbacks have been introduced in model training sessions, showing their real capability at the deployment stage (discussed in chapter 5.9). A callback is a simple model that receives an abstract function as an argument, and calls it with some given parameters at some point in the Pipeline. As simple as it seems, creating lists of callbacks and passing them deep in the pipeline enables the user to create callable custom functions outside the Pipeline infrastructure. As the model passes some key moments in the training session, some specific callbacks might be called with specific information. This proved very useful when the Web Socket communication interface was implemented. Allowing callbacks to 'extract' information from the middle of the training process allowed the infrastructure to pass the information from the depth of parameter tuning up to the web interface, so the user gets notified as the model improves.

5.3 How to configure

As also stated in the introduction of this chapter, the Pipeline infrastructure relies heavily on a configuration file, that has predefined keys and default values, but allows the user to tune the models as he desires. The extraction of the most important steps in a configuration file allows for greater abstraction, moving the decision from the framework's code to the end user. It also prohibits the Pipeline to take hard-coded, unchangeable decisions.

Reading this section, one might doubt the "Automatic" nature of this Automated Machine Learning framework, thinking that some work has to be done in properly configuring it. Although the Pipeline is open to complex customization that allows skilled users to decide how to do certain processing steps, the default values assigned to predefined configuration file together with the power of evolutionary algorithms that search various model configurations, enables the Pipeline to be fully autonomous and empowers even engineers with little machine learning background to use it.

Whether the user uses the online version of the framework or the on-premises library, passing configurations to the Pipeline is made very easy. The website offers a configuration tree that dynamically expands and offers details about each possible step, together with the available option to choose. The on-premises version allows the

user to configure the pipeline provided that he reads the user documentation available in the Pipeline package as well on the Docs page of the site (hosted temporarily at anon.antonmihai.com). The configuration offers choices for well known parameters such as normalization methods, outlier capping factors, learning rates and population sizes. The on-premises and the cloud versions of the configuration can be previewed in Figure 11.

Figure 11: On-premises and cloud Pipeline configuration.

One tunable parameter that differentiates this implementation from many others in the market is the time parameter. In the classical machine learning flow, programmers train models using epochs, relying on intuition on how many to choose for an optimal train time. To make this framework truly automated and a useful tool for its users,

it has to be on-time. Thus, the time parameter can be specified in a simple string format of the form **<days>d <hours>h <minutes>m <seconds>s**, where each parameter inside **<>** represents a positive integer, enabling the user to train the model for an almost exact time (+-2% due to irregularities in epoch durations). One particular useful use case for this configuration parameter is overnight training. Sometimes data scientists leave models to train overnight, so when they come the next day they get the results. It is often the case when they realise that either the model has had a short train time, without leveraging the whole period, either it still has to run for a long period, too long that it is not useful and has to be stopped. All these are caused by poor epoch count choices. Since epochs do not offer a consistent and comparable amount of time, it is better to use them dynamically, like this framework does. These being said, the framework can be configured to be on-time, specifying the time available to get a result.

Along time, the second most important parameter is the predicted column. No regression or classification task can be performed without the goal in mind, namely a feature to predict. This choice is simple and is often decided before the framework is even used.

To sum up, the framework has many tunable parameters. They can be chosen as the defaults, leaving them unchanged, or they can be tuned until the perfect configuration results. It is also important to note, though, that the predicted feature (goal) and the train time are the only mandatory parameters to set for the pipeline to start working.

5.4 Models included

Many frameworks today operate after the following principle: *Batteries included, but removable*. It makes common sense to build complex systems, from which people can choose the exact components they need. The model, as an abstract entity, lays at the core of an Automated Machine Learning framework. It is the model that makes the framework intelligent and it is the model that puts "Learning" in Machine Learning. It is the only part that develops intelligent behaviour provided enough quality data and time.

Many would consider that diversity beats specificity. It might be the case in some product implementations, but in a highly generic learning framework, blending dozens of models together might yield a hard to untangle combination rather than a winning cooperation. The decision to keep the model pool small was made considering that, according to the Pareto law, 20% of the models might prove effective in 80% of Machine Learning tasks. Thus, the models used are SVMs, Random Forests and, last but not the least, Neural Networks. Statistically, these models outperform niche implementations on generic tasks, making them the best choice for the moment.

It is worth mentioning that the Pipeline has two training subflows, one in which the user specifies the model to train and trains it according to the configurations provided in the configuration file and one in which the user configures a range of possible values and lets the Evolutionary Algorithm find the best performing model. For the first of the flows, each of the 3 models is available to train and experiment with. For the other one, namely the "evolutionary flow", only neural networks are available in the current implementation, due to generic evaluation problems described in 5.6. As a summary, it is almost impossible to directly compare the training performances of two different models, since they have different training algorithms. For example, in order to decide how fit a neural network chromosome is in the evolutionary flow, the framework takes into consideration the convergence graph, as well as the derivative of the loss over time. It would be impossible to pretend Random Forests to yield the same metrics since their learning algorithm is not gradient based.

Although the model pool is kept small, it can increase exponentially by enabling the evolutionary flow to experiment with various configurations. Although two models might be of the same type, it is the configuration and hyperparameter choices that make much of a difference in overall performances. Thus, keeping the model choices count low allows the pipeline to experiment more deeply with the available ones rather than taking a shallow approach in evaluating too many with no notable result.

5.5 Frameworks used

We live times when the choice pool of frameworks to use is exceptionally big. As we speak a dozen of new frameworks get created, with different use cases, with the common goal of making product development easier, faster and more reliable. Even this paper describes a framework that is intended to help data scientists and machine learning enthusiasts achieve better results in their projects. We have long passed the era when every little detail was implemented from scratch and code was not reused for generic purposes. Nowadays common software development patterns emerge and are rapidly replaced by frameworks that fit well in different scenarios.

As the complexity of the framework developed along this paper is notably high, implementing all the details from scratch would have generated a very long and tedious development journey, full of trials and errors and bugs arising from the void. Using highly customizable and controllable frameworks allowed the complexity to grow without compromising understanding of the underlying principles of machine learning and full stack development. Also, due to the fact that usually behind frameworks work large and experienced teams makes their product scalable and highly optimized, perfect for using it as a strong base for a more complex product. Moreover, using frameworks yields built in performance, security and optimisation.

It is worth to mention that the project, although having it's most valuable part the Automated Machine Learning framework, is built as a full stack solution, allowing for cloud based training as well as on-premises usage. The full stack nature of the product brings the need of using already *state of the art* technologies and frameworks for a manageable development. In the next two sections two different framework types will be discussed, namely data science related frameworks (that were used to build the Automated Machine Learning application), and product related ones, used to build the end-to-end solution.

5.5.1 Data Science frameworks

Since Machine Learning has been an emerging field of research in the last decade, numerous frameworks have been built to really leverage the power of it. From data analytics to model training and deep learning frameworks, they all blend together and can be used as small building blocks in creating the bigger picture. As this is a high-level framework by itself, it uses and incorporates various others in order to achieve built-in complexity and scalability.

Starting with the most important building block in creating a flexible and scalable Automated Machine Learning solution, I am using PyTorch [Paszke et al., 2019], a very performant and robust deep learning framework, able to be customized and used also as a high level API as well as a lower level, controllable library. The whole infrastructure of self customizing neural networks is built on top of PyTorch, leveraging both the power of machine learning and the lower level controllability. As any other deep learning library, the basic and most important object that flows through the entire infrastructure is the tensor, a multidimensional array capable of abstractly describing a point in a high order geometrical space. A neural network in PyTorch can be easily defined by a series of layers, that contain no more than a set of weights through which data passes, some activations and a set of dropout values. What is remarcable of PyTorch is that the only method that has to be implemented is the forward function, describing how the data passes from the input layer to the output one, while the framework remembers by itself the gradients using the autograd built in feature, being able to automatically adjust weight according to the gradients accumulated. Thus, this makes PyTorch a framework that easily builds and trains neural networks, while being able to adapt to flexible demands.

As discussed in earlier chapters, the whole architecture of the framework is as much as possible data driven, meaning that data is the most important object that flows from one component to the other. As data can be saved and manipulated in various

ways, a common method has to be found in order to create a smooth flow. For this, the Python library Pandas was used, which is a data science and statistics library, able to cope with big amounts of data, acting similar to in-memory databases, with data science add-ons.

As mentioned in the previous section about Model Choices, neural networks are the foundation of the Automated Machine Learning Pipeline, but 2 others are also included for diversity. For SVMs and Random Forests the pipeline uses Scikit-learn, also known as SkLearn in the Python community [Pedregosa et al., 2011], which is a very popular machine learning framework used by many in different projects. It exposes simple APIs and off the box working models, which makes a training session as simple as one line of code, mostly calling the fit() method on the specific model.

5.5.2 Product frameworks

Going in a top down manner, the first to mention is the framework used for building the website, namely React. It is a commonly used tool for building complex websites, with multiple components. Mentioning components, it is worth mentioning that React is a modular tool, used for building the whole with small, measurable and manageable steps. The granular nature of React allows for a steep learning curve and for building complex sites taking small steps rather than big, often not completely understood leaps.

Linking the frontend to the backend created the need of a strong web server building framework. The most popular choices for Python are Flask and Django. In building the Automated Machine Pipeline, the latter was used. It's scalable and easy to build, allowing for easy experimentation towards building a web API. The main principle in Django is to split the problem into independent applications instead of managing everything together. Each application is then linked into a common API and exposed to whoever wants to connect.

The communication between the frontend and the backend is made using the built-in REST capability in Django. Since the learning processes in the Pipeline can take hours to complete, a bidirectional communication had to exist in order to keep the user informed while training the models. This is how Django Channels made their way into the whole product, allowing Django to use Web Sockets to communicate bidirectionally with the client.

Django has a built-in server that is meant to be used specifically for development purposes. It offers a small degree of concurrency, but it is still not production ready. Thus, 2 other frameworks were used to make the server scalable and ready to serve multiple connections with ease, namely Gunicorn and Daphne. The first one is a

WSGI (Web Server Gateway Interface) that handles multiple requests and is able to balance the load it receives before handing it over to Django. The second is used for the web socket communication, since it is an ASGI (Asynchronous Server Gateway Interface) server.

5.6 Metrics and model scoring

Building Automated Machine Learning is challenging from many points of view. One of them is creating a unified scoring method that would weigh in the same way multiple models in order to decide which is the best and which should not be considered any more.

The classic flow in machine learning is to score models based on metrics on some validation dataset, which is often sampled from the initial data source. While this metric provides some sort of valid evidence, it's still a shallow approach to evaluate the quality of the model. Since models are randomly created, with the initial weights and biases being usually sampled from a normal distribution, they do not start at the same level, thus the evolution of the loss over time should also be considered when creating the final score.

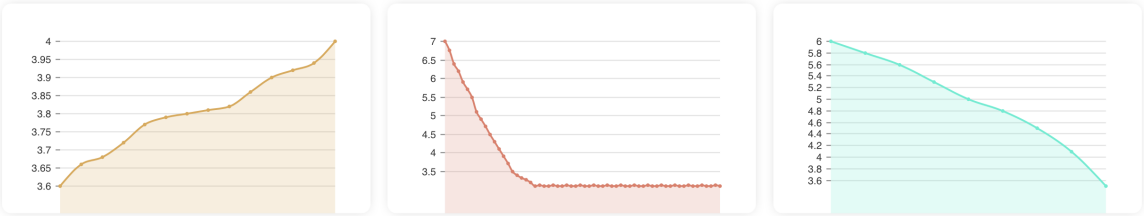

Figure 12: Loss drop comparison.

Analyzing a sample loss drop from the results page of the project (Figure 12), we see 3 different models, each with its own convergence graph. At a first glance, the one on the left seems to perform poorly, while the remaining ones look like they learn from the data. If the metric would rely 100% on the final performance of the model, the models described above would not differ much, since they all have final losses in the range of [3.4, 4]. A data scientist looking at those converge graphs would seize some patterns immediately. The model on the left would be discarded in the first place, since it only got worse with time. The model in the center, although providing the best final score, stagnates in learning for almost two thirds of the epochs and does not provide any promise for further loss drop. Considering the last model, that would be the one to be further trained, since it demonstrates constant loss drop, lowering its scores steadily

over the epochs.

Providing this brief proof of why the final loss is not the only metric to be used in overall model scoring, the following question arises: How does one automatically replace the hypothetical data scientist that analysed the result in the paragraph above? The answer could come in many forms, but leveraging the power of calculus, the Pipeline automatically computes the derivatives of the loss function, detects the regions of rising loss and penalizes the model for the epochs in which it does not improve. This way, from the models described above, the best one, as chosen automatically by the framework, is the green one (the rightmost one).

Having said this, it is the time to discuss why it is hard to blend models together and obtain a smooth loss function for all of them. Neural networks use backpropagation for the training algorithm and provide this kind of graph, as in Figure 12. Random forests, on the other side, cannot create a relevant statistic, since they only improve over time. Taking into account other model specific metrics other than final dataset score makes it hard to blend multiple approaches in the same flow. Using only neural networks for the evolutionary flow is a choice based on the desire to achieve overall performance for the framework. It allows the user to go deeper in evaluating their performances, since custom metrics can be used that work on this specific type of model only.

5.7 Pipeline flows

Every product has some flows, or main functionalities. Each time a user interacts with it, they use one of the available well-known flows. Since the pipeline is a modular and customizable framework, it can have various flows depending on the configuration the user provides. Although the pipeline behaves randomly taking into account the user's desires, it can be seen as a Finite Automata on a highly abstract level, where some predefined states exist, allowing the user to go from one to the other without compromising the integrity of the final result.

Below there will be 7 flows presented, as designed and planned before the Pipeline was implemented. Although variations may exist, these are the flows a user needs in order to fit a dataset, convert data and generate predictions.

- The evolutionary flow
  - A pipeline that receives raw data and returns the best model already trained.
  - Raw input → Pipeline (Feature engineering, Learning) → Model+Savable pipeline (for further data conversion using the same logic and future

retraining using the same model) = **FLOW1** (**FLOW2** + **FLOW4**)

- Data engineering flow
  - The pipeline receives the raw data and is used for data cleaning and feature engineering.
  - Raw Input → Pipeline(Feature engineering) → Cleaned data+Savable pipeline (for future conversion using the same logic) = **FLOW2**
- Training flow
  - The pipeline receives presumably clean and optimised data and is used to train a model.
  - Data → Pipeline (Default Learning) → Trained model (Using the default model configured and tuned by the user, no need of savable pipeline, although possible, because the only useful object in the future will be the saved model, used to predict or further train) = **FLOW3**
  - Data → Pipeline (Evolutionary Learning) → Trained model (Using the evolutionary algorithms technique, which can be customised in the configuration file by the user, for narrowing the search space; as in the case of **FLOW3**, the only really usable end product is the model which can be saved and reused) = **FLOW4**
- Data conversion flow
  - Using a previously saved pipeline (with the logic inside) the algorithm converts raw data (similar in shape and attributes as the data that the pipeline has seen before) to usable data.
  - Raw Data → Saved pipeline → Converted data = **FLOW5**
- Prediction flow
  - Using a previously saved Pipeline or Model, predicts the output for new data.
  - Raw data → Pipeline → Output (The saved pipeline also contains the model inside; it converts the data and returns the prediction) = **FLOW6**
  - Processed data → Model → Output (The saved model is used to predict already converted data = **FLOW7**

5.8 Performance of the framework

Provided the desired available time to search for architectures and train the best one, the Pipeline is able to deliver a fully trained model along with statistics and charts regarding the tried ones, in the specified time frame. To be further analysed by data scientists are the evolutionary convergence graph (Fig. 13) and the pool of convergence rates for each model, similar to the ones in Fig. 14.

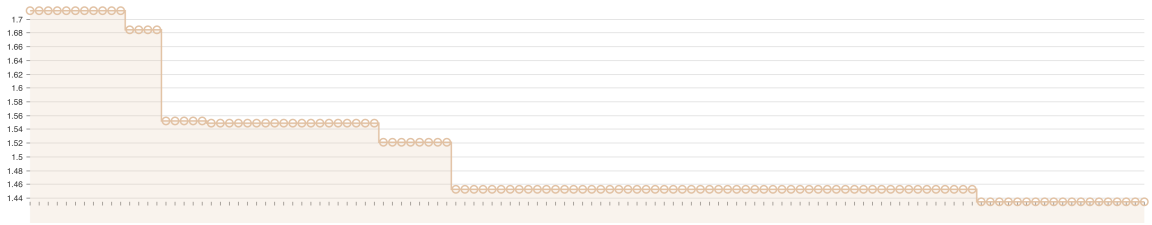

Figure 13: Evolutionary convergence sample.

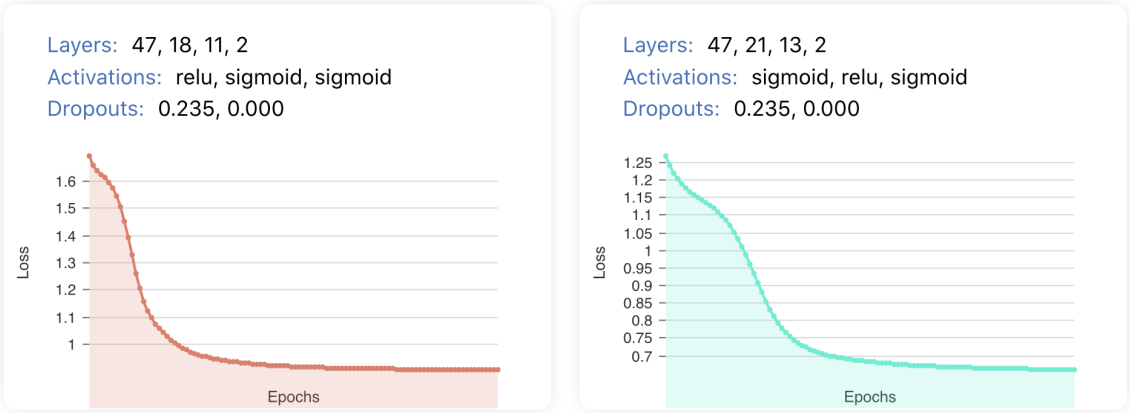

Figure 14: Model convergence sample.

```
Evolutionary Model -
  - Best model:
NeuralNetwork - Layers: [784, 73, 79, 21, 131, 74, 113, 10] | Activations: ['relu', 'linear', 'relu', 'linear', 'linear', 'linear', 'relu', '
  - Top models:
Score: 0.23439 Model: NeuralNetwork - Layers: [784, 73, 79, 21, 131, 74, 113, 10] | Activations: ['relu', 'linear', 'relu', 'linear', 'linear
Score: 0.27371 Model: NeuralNetwork - Layers: [784, 59, 98, 78, 116, 66, 48, 10] | Activations: ['relu', 'linear', 'relu', 'linear', 'linear'
Score: 0.29090 Model: NeuralNetwork - Layers: [784, 61, 90, 71, 102, 70, 51, 52, 10] | Activations: ['relu', 'linear', 'linear', 'linear', 'l
Score: 0.68510 Model: NeuralNetwork - Layers: [784, 62, 85, 76, 122, 69, 118, 10] | Activations: ['relu', 'linear', 'linear', 'linear', 'sigm
Score: 1.61798 Model: NeuralNetwork - Layers: [784, 57, 85, 74, 123, 66, 108, 10] | Activations: ['relu', 'linear', 'relu', 'linear', 'sigmoi
Score: 2.35401 Model: NeuralNetwork - Layers: [784, 71, 127, 26, 18, 72, 102, 82, 10] | Activations: ['sigmoid', 'relu', 'linear', 'sigmoid',
Score: 2.35729 Model: NeuralNetwork - Layers: [784, 59, 123, 21, 108, 78, 92, 57, 10] | Activations: ['sigmoid', 'sigmoid', 'relu', 'relu', '
Score: 2.37682 Model: NeuralNetwork - Layers: [784, 52, 140, 20, 93, 65, 107, 50, 10] | Activations: ['sigmoid', 'sigmoid', 'relu', 'relu', '
Score: 2.38269 Model: NeuralNetwork - Layers: [784, 70, 80, 21, 107, 74, 102, 52, 10] | Activations: ['sigmoid', 'sigmoid', 'relu', 'relu', '
Score: 2.38490 Model: NeuralNetwork - Layers: [784, 58, 136, 23, 100, 66, 100, 55, 10] | Activations: ['sigmoid', 'sigmoid', 'linear', 'relu'
```

Figure 15: Best 10 models after EA.

Both Fig. 13, Fig. 14 and Fig. 15 yield insights on how the evolutionary frame-

work performed on the specific dataset. The graphical ones provide visual information on the actual loss convergence of the Pipeline, while the textual one presents the top 10 models tried, ordered by their scores, showing information about the layer choices, as well as activations and dropout rates used. Using those charts, data scientists may draw conclusions on which models to pursue next.

Also, the Pipeline has been tested, along other *state of the art* frameworks on benchmark datasets, where the proposed solution yielded above average results. A detailed analysis will follow in chapter 6.

5.9 Deployment

The complexity of the pipeline creates a great demand for resources, relying on the machine it runs on. Running it locally for development and testing was a good and powerful enough alternative, but for production uses or just for better performances, other alternatives had to be considered. To split the problem in two groups, there was the need for frontend deployment on a webserver and also deployment of the backend of a cloud computing server.

Deploying the React application was simple, using FTP (File Transmission Protocol) for uploading the built application files on the hosting provider's platform. The real challenge was in deploying the server, since it is an aggregation of multiple services that have to work together to respond to the frontend's needs.

The server, besides the Automated Machine Learning framework which is the base brick of the whole infrastructure, multiple other layers had to be provided, as also mentioned in 5.5.2. The whole application was built using Docker [Willis,], which is a Container based service that creates a virtual environment similar to a virtual machine that enables users on different operating systems and setups to run the same code without any further dependencies. Since Docker is designed to run a single container, another service named docker-compose had to be included, in order to be able to run multiple containers at the same time. Those containers are the following: Python based image for the gunicorn wsgi server, Python based image for the daphne asgi server, nginx image for the reverse proxy, redis image for the websockets and a Postgres image for database storage.

There are multiple cloud providers on the market, each with its advantages and disadvantages. The first choice was Heroku, since it is a popular cloud vendor, allowing for ease of use and simple customizations. After repeated trials and error, I have found it impossible to deploy docker-compose based containers on it, since it allows only single image containers. After some research on the available options, although

complicated at the first use, AWS(Amazon Web Services) was the choice to go with. Due to the multitude of services and a complicated pricing schema, it has a tedious learning procedure. After cycling through the documentations and the available options, the EC2 (Elastic Compute Cloud) service, which provides virtual machines with public IP's was the perfect choice. The whole project was uploaded in the cloud, built, run and, using the public IP it yields a very smooth flow.

5.10 How to use

As the previous chapters state, this Pipeline is capable of developing intelligent behaviour and trains complicated and effective models. The workflow behind is not that simple and, although designed to automatically perform tasks for a given project, some instructions have to be provided to fully leverage the capability of the framework. The Pipeline class is the core of the framework, capable of performing all the flows described above. Depending on the configuration, it might train one or more models in a flow, only to choose the best and keep it in the end. Thus, a Pipeline that has finished training will contain one single trained model. To start with, the framework has two dependencies, namely data and configuration. The dataset is the obvious need, since without it this tool would be useless. The most minimal configuration, as also stated above, requests the predicted column and the train time. Provided only those parameters, the Pipeline will perform all of the state of the art techniques, engineer the data and find the model autonomously. Obviously, the configuration can be fully customized, as described above. Below I will describe the two major use cases of the framework, namely the on-premises and the cloud one.

5.10.1 On-premises usage

For users who prefer to keep their data locally, the Pipeline package is available on Github (link to repository). It can be cloned locally and used as the needs demand. Provided that the local Python environment has all the needed dependencies, found in the requirements.txt file in the package root, the following code (Figure 16), can be used together with the Pipeline package and provides the most minimal setup needed to start training it. This code will give you the best possible found Pipeline along with it's trained model.

```
from Pipeline import Pipeline, read_csv

data = read_csv("data.csv")                    # read the dataset
config = {                                     # create a minimal configuration dictionary
    "DATA_PROCESSING_CONFIG": {
        "PREDICTED_COLUMN_NAME": "Survived",
    },
    "TRAINING_CONFIG": {
        "TIME": "10s",                         # specify the time
        "PREDICTED_COLUMN_NAME": "Survived"    # and the predicted column
    }
}

pipeline = Pipeline(config=config)             # create the pipeline with the augmented configuration
model = pipeline.fit(data)                     # fit the pipeline and get the model
```

Figure 16: Code for minimal setup.

5.10.2 Cloud usage

The cloud functionality was implemented in the idea that powerful public resources (in this particular case AWS) can be used in order to provide a fair trade between cost of operation and performance. The following steps explain how to use this version of the framework.

First, drag and drop a .csv file in the cloud page(17):

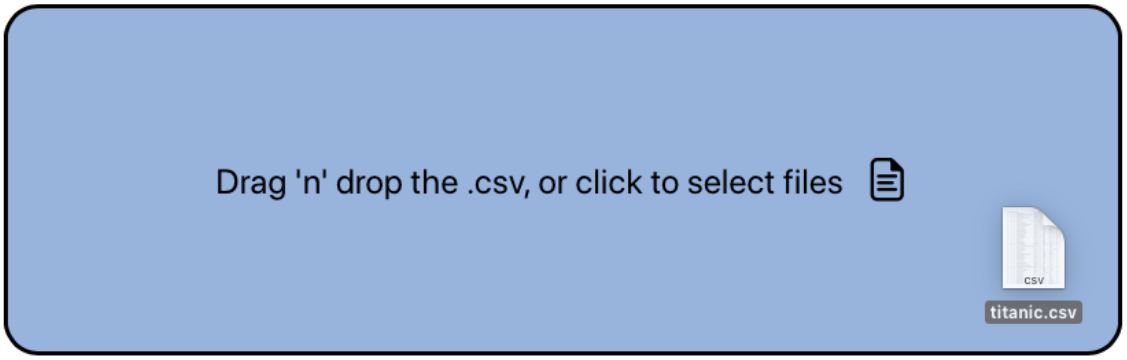

Figure 17: Data dropping area.

After the dataset is uploaded to the cloud storage, a preview is shown and the configuration components appear (18). After imputing the right configuration, setting the time and the predicted column, start the fitting process by clicking the orange button:

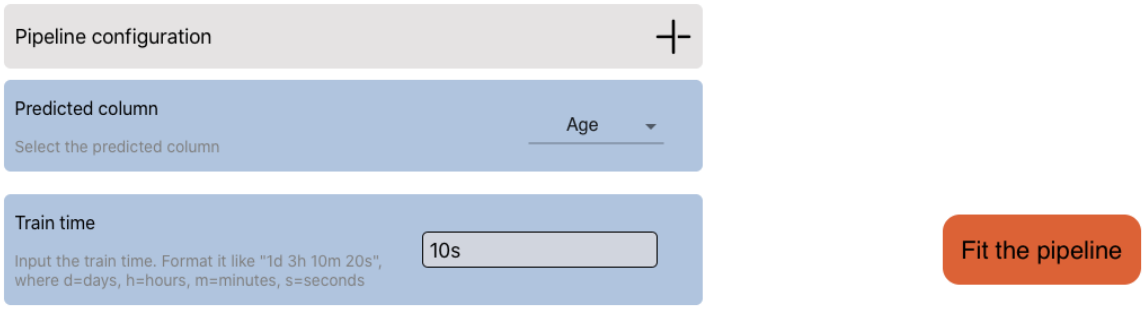

Figure 18: Customization area.

The training process will start and all the convergence graphs, together with intermediate information will be displayed interactively. After the training process is done, to download the pipeline press the orange button at the bottom of the page. A binary file will be downloaded which is able to be used locally.

6 Performance Measures

Dataset \ Framework	Pipeline		Autokeras		MLBox		TPOT		Sklearn Random Forest		Sklearn KNN	
	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test
Titanic	90%	85%	84%	82%	94%	84%	94%	82%	86%	82%	99%	82%
Mnist	97%	96.5%	100%	64%	82%	67%	100%	98%	100%	98%	97%	96%
Wisconsin breast cancer	98%	100%	99%	94%	100%	96%	98%	98%	97%	96%	100%	98%
Credit fraud detection	99.9%	99.9%	99.92%	99.92%	99.91%	99.9%	99.94%	99.94%	99.8%	99.8%	99.9%	99.9%
Iris	97.5%	93.3%	97.5%	93.3%	100%	93.3%	97.5%	96.6%	95.8%	96.6%	100%	95%

Table 1: Multiple frameworks tested on multiple datasets

The previously Pipeline has been tested with various datasets, more or less popular to ensure that diversity is handled and that the desired autonomy is fully reached. For benchmark testing, the MNIST Digits Dataset [Lecun et al., 1998] and other popular classification datasets were used. For comparison, 3 other AutoML frameworks and 2 standard models have been considered, namely AutoKeras [Jin et al., 2018], MLBox, TPOT [Olson and Moore, 2019] and Scikit-learn Random Forest, Scikit-learn KNN [Pedregosa et al., 2011].

The Benchmark tests are presented in Table 1, showing the accuracies of the aforementioned learning algorithms on the benchmark datasets. Both the train and test set accuracies are displayed, after a 5-fold cross validation phase.

Figure 19: Train set results.

Figure 20: Test set results.

Figures 19 and 20 present in a more visual way the train and test accuracies from Table 1. Once more, the proposed Pipeline (represented in blue dots) which shows top accuracies in all scenarios, is directly comparable with *state of the art* frameworks and has an above average capability of generalization, as shown by high accuracies in the test graph as well.

It is worth mentioning that the proposed Pipeline, the AutoML comparison frameworks and the Scikit-learn classifiers were used with the default configuration, once again to prove that Evolutionary Algorithms strive to find a good model without being explicitly configured. Also, the framework handles regression tasks with the same

ease as it does with classification, but the latter was used since accuracies are a better metric for presenting results and testing on benchmark datasets.

7 Conclusions and Future work

Developing sustainable Automated Machine Learning frameworks drives the whole field towards the big goal of achieving general Artificial Intelligence. Often AutoML is related to 'automating automation', since it adds another layer of autonomy to the already available machine learning tools.

The proposed solution in this thesis is far from that goal, and so are almost all the other available tools, since achieving general artificial intelligence in an automated fashion is still far away. Further directions would be towards adding EAs to the whole pipeline, including the data engineering part, to not only discover peculiar neural network combinations, but also various ways of blending together feature engineering methods for achieving better scores.

Also, since the research in this field is still on the rise, further scoring functions have to be found in order to enable ensemble learning under an unified scoring method. Also, leveraging the power of computer clusters to train pools of models in parallel would as well speed up the process.

Summing up, the aim of the thesis is to emphasize the importance of automating Machine Learning for analysis and time saving purposes. Furthermore, it presented results that confirm the viability, generality and performance of the proposed Evolutive Algorithms based AutoML Pipeline.

References

[Abadi et al., 2015] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

[Baker et al., 2016] Baker, B., Gupta, O., Naik, N., and Raskar, R. (2016). Designing neural network architectures using reinforcement learning.

[Elsken et al., 2018] Elsken, T., Metzen, J. H., and Hutter, F. (2018). Neural architecture search: A survey.

[Feurer et al., 2015] Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., and Hutter, F. (2015). Efficient and Robust Automated Machine Learning. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems 28*, pages 2962–2970. Curran Associates, Inc.

[Gaveau et al., 2017] Gaveau, J., Martret, C. J. L., and Assaad, M. (2017). Performance analysis of trial and error algorithms.

[Gewers et al., 2018] Gewers, F. L., Ferreira, G. R., de Arruda, H. F., Silva, F. N., Comin, C. H., Amancio, D. R., and da F. Costa, L. (2018). Principal component analysis: A natural approach to data exploration.

[Ghys, 2015] Ghys, (2015). *The Butterfly Effect*, pages 19–39.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

[He et al., 2019] He, X., Zhao, K., and Chu, X. (2019). Automl: A survey of the state-of-the-art.

[Herbrich and Graepel,] Herbrich, R. and Graepel, T. Machine Learning.

[Jin et al., 2018] Jin, H., Song, Q., and Hu, X. (2018). Auto-keras: An efficient neural architecture search system.

[Jouppi et al., 2017] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., luc Cantin, P., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V.,

Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., and Ross, J. (2017). In-datacenter performance analysis of a tensor processing unit.

[Lecun et al., 1998] Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324.

[Olson and Moore, 2019] Olson, R. S. and Moore, J. H. (2019). TPOT: A Tree-Based Pipeline Optimization Tool for Automating Machine Learning. In Hutter, F., Kotthoff, L., and Vanschoren, J., editors, *Automated Machine Learning: Methods, Systems, Challenges*, The Springer Series on Challenges in Machine Learning, pages 151–160. Springer International Publishing, Cham.

[Paszke et al., 2019] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library.

[Pedregosa et al., 2011] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, É. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12(85):2825–2830.

[Rosenblatt, 1958] Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386.

[Serafino, 2013] Serafino, L. (2013). No free lunch theorem and bayesian probability theory: two sides of the same coin. some implications for black-box optimization and metaheuristics.

[Snoek et al., 2012] Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms.

[Tang, 2013] Tang, Y. (2013). Deep learning using linear support vector machines.

[Thornton et al., 2012] Thornton, C., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2012). Auto-weka: Automated selection and hyper-parameter optimization of classification algorithms. *CoRR*, abs/1208.3719.

[Wei et al., 2019] Wei, Y., Zhao, P., Yao, H., and Huang, J. (2019). Transferable neural processes for hyperparameter optimization.

[Willis,] Willis, J. Docker and the three ways of devops.

[Wolpert and Macready, 1997] Wolpert, D. H. and Macready, W. G. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82.

[Yao et al., 2018] Yao, Q., Wang, M., Chen, Y., Dai, W., Li, Y.-F., Tu, W.-W., Yang, Q., and Yu, Y. (2018). Taking human out of learning applications: A survey on automated machine learning.

[Yu and Zhu, 2020] Yu, T. and Zhu, H. (2020). Hyper-parameter optimization: A review of algorithms and applications.