Warsaw University of Technology

FACULTY OF
MATHEMATICS AND INFORMATION SCIENCE

Master's diploma thesis

in the field of study Computer Science
and specialisation Artificial Intelligence

Automated Feature Engineering

Łukasz Wasilewski
student record book number 268821

thesis supervisor
dr hab. inż. Przemysław Biecek, prof PW

WARSAW 2019

....

supervisor's signature author's signature

Abstract

Automated Feature Engineering

The thesis concerns the area of automated machine learning. One of the steps of machine learning is to prepare and transform data, in other words, feature engineering. The thesis is devoted to automating the process.

The first chapter is an introduction and contains information about motivation, objectives, and the thesis structure. The goal of the thesis is to automate the feature engineering process by creating a tool that uses the reinforcement learning technique. Automating the process could potentially save the high valued time of well-qualified specialists from various domains.

The second chapter contains theoretical foundations explaining concepts used in the thesis. First of all, the feature engineering process is described, including transformations that can be applied in order to produce better quality features. Moreover, the concepts and techniques connected with reinforcement learning are introduced.

The third chapter explains how the feature engineering process can be presented as a reinforcement learning process. Moreover, problem representation and used notation are presented.

The fourth chapter contains a brief overview of existing tools that automate the machine learning process. It focuses on the ways that the tools automate the feature engineering part of the machine learning pipeline. Moreover, the chapter includes technical documentation of the implemented tool, which is the solution overview, description of used libraries, technologies, and the user manual.

The fifth chapter contains an analysis of the performed experiments of the created tool. First of all, the investigation of the learning process is carried out. Next, the behavior of the algorithm with the usage of the different values of parameters is examined. Furthermore, the overall performance of the feature engineering process performed by the tool, along with applied transformations, are analyzed.

The last chapter is a summary of the thesis. It includes conclusions and presents potential areas of further research and ways of developing and improving the implemented tool.

Keywords: feature engineering, reinforcement learning, machine learning, AutoML, artificial intelligence

Streszczenie

Automatyczna inżynieria zmiennych w modelach predykcyjnych

Praca porusza problem dotyczący automatycznego uczenia maszynowego. Jednym z etapów uczenia maszynowego jest odpowiednie przygotowanie i przekształcenie danych wejściowych. Proces ten nazywamy inżynierią cech. Niniejsza praca dotyczy automatyzacji tego procesu.

Pierwszy rozdział stanowi wstęp do pracy i zawiera motywację, cele oraz opis struktury pracy. Celem pracy jest automatyzacja procesu inżynierii cech poprzez stworzenie narzędzia używającego technikę uczenia się ze wzmocnieniem. Automatyzacja tego procesu może potencjalnie zaoszczędzić wiele cennego czasu wykwalifikowanych specjalistów z różnych dziedzin.

Drugi rozdział zawiera podstawy teoretyczne wyjaśniające pojęcia stosowane w pracy. Przede wszystkim został opisany proces inżynierii cech w tym transformacje, które można zastosować w celu uzyskania lepszej jakości danych. Ponadto zostały omówione pojęcia i techniki związane z uczeniem ze wzmocnieniem.

Trzeci rozdział wyjaśnia, w jaki sposób można przedstawić proces inżynierii cech jako proces uczenia ze wzmocnieniem. Ponadto zaprezentowano reprezentację problemu i stosowaną do tego celu notację.

Czwarty rozdział zawiera krótki przegląd istniejących narzędzi automatyzujących proces uczenia maszynowego. Przede wszystkim omówiony został tutaj sposób, w który istniejące narzędzia automatyzują inżynierie cech. Co więcej, rozdział ten zawiera dokumentację techniczną stworzonego narzędzia tzn. opis rozwiązania, w tym używane biblioteki, technologie oraz instrukcję obsługi.

Piąty rozdział zawiera analizę przeprowadzonych eksperymentów. Na początku została przeprowadzona analiza procesu uczenia się. Następnie badane jest zachowanie algorytmu przy użyciu różnych wartości parametrów oraz w jakim stopniu inżynieria cech wpłynęła na skuteczność modelu.

Ostatni rozdział jest podsumowaniem niniejszej pracy magisterskiej. Zawiera wnioski, a także przedstawia potencjalne obszary dalszych badań oraz sposoby rozwijania i propozycje ulepszeń zaimplementowanego narzędzia.

Słowa kluczowe: inżynieria cech, uczenie przez wzmocnienie, uczenie maszynowe, AutoML, sztuczna inteligencja

Warsaw, ...

Declaration

I hereby declare that the thesis entitled „Automated Feature Engineering", submitted for the Master degree, supervised by dr hab. inż. Przemysław Biecek, prof PW, is entirely my original work apart from the recognized reference.

....

Contents

1. Introduction

1.1. Motivation

This thesis concerns rapidly developing areas of machine learning - the automatic building of machine learning models (AutoML). The structure of most of the AutoML systems consists of three parts - feature selection, feature transformation, and tuning of the classifier. This master thesis examines the possibilities of automation of the feature engineering process through a reinforcement learning algorithm. Reinforcement learning algorithms have led to spectacular successes in computer game systems such as Go or Chess [Silver et al., 2018]. Applying that technique to the feature engineering process allows verifying how much this success can be repeated in the case of automatic model building.

Moreover, the feature engineering process is very time consuming and requires domain knowledge of high qualified specialists. These factors make the process very expensive. Automation of feature engineering will save much time of these people and make the whole machine learning process cheaper and faster. What is more, in most cases, when dealing with "real-world" data, the process is an inherent part of the machine learning pipeline. Applying even simple feature engineering can significantly increase the machine learning algorithm's performance.

1.2. Objectives of this thesis

The purpose of the thesis is to propose, design, and implement an algorithm based on reinforcement learning, which would support the feature engineering for binary classification. The result is a methodology and a tool to facilitate the automatic building of machine learning models by automating the feature engineering process. Moreover, experiments verifying to what extent the proposed algorithm and its implementation make the feature engineering process easier and faster will be carried out.

1.3. Thesis structure

The first chapter contains motivation and outlines the goals of the thesis. The following chapter presents the main theoretical foundations, such as feature engineering, reinforcement learning. It describes what feature engineering is and outlines main feature engineering techniques, e.g., missing values imputing feature encoding and scaling. Moreover, the chapter introduces reinforcement learning techniques with connected concepts such as Markov Decision Process or policy. Furthermore, multiple methods of solving problems modeled as reinforcement learning problems are presented. The third chapter shows how to apply reinforcement learning techniques to a feature engineering problem in order to automate the process. The fourth chapter briefly characterizes existing tools that are capable of performing automatic feature engineering. Moreover, it contains the technical documentation of implemented solution: implementation details and the user manual. In the fifth chapter, the results of performed experiments are presented along with discussion. The last chapter summarizes the thesis and suggests possible ways of further research.

2. Theoretical foundations

2.1. Introduction

According to [Shalev-Shwartz and Ben-David, 2014], machine learning (ML) is a process of automatic detection of meaningful and important patterns in data. In other words, having given some input data, the goal of ML is to learn how to produce correct output for new, unknown previously data, basing on known data in order to solve a given task. Machine learning lies somewhere in the intersection of mathematics, statistics, and computer science. The meaningful patterns are usually detected by statistical models implemented on a computer machine. In order to achieve satisfying results, wide specialists knowledge from many areas is required. Thus automation of the process of building and implementing the machine learning process is a method of saving high valued time of those specialists.

Reinforcement learning is one of the techniques used in the area of machine learning. Contrary to the other techniques as supervised or unsupervised machine learning, reinforcement learning does not rely on previously collected data. Instead of learning basing on the examples, it learns by interaction with the environment in order to find optimal behavior [Sutton and Barto, 2018].

The key aspects of reinforcement learning are: a learning agent - a component that makes decisions or takes actions basing on collected experience, an environment with which it is interacting, a reward/penalty function that gives a positive or negative feedback to the agent and makes agent possible to learn rules and conditions of the environment in order to discover the best strategy. The optimal behavior is not easy to learn - sometimes it is better to sacrifice a big immediate reward to maximize the overall result in the future. Figure 2.1 shows schema of general reinforcement system. The agent acts in the environment and receives feedback, which is the immediate reward and the following state.

This chapter describes the feature engineering process and presents some of the feature engineering techniques. Moreover, reinforcement learning concepts are introduced basing on [Sutton and Barto, 2018]. Additionally, I propose some techniques to solve reinforcement learning problems.

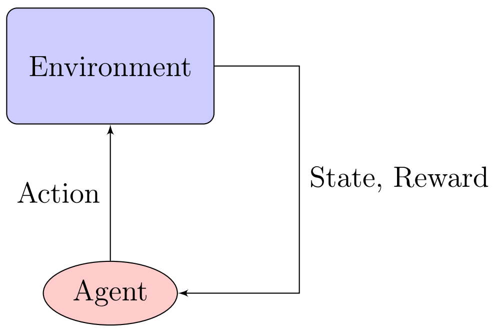


Figure 2.1: General reinforcement learning system schema

2.2. Feature engineering

2.2.1. Machine learning pipeline

Definition 2.1 (Feature). A feature f is a representation of some property of a characterized object. The type of the feature can be, for example, a numeric value, a string, a categorical value, a date-time, etc. Space of all features is denoted by $\mathcal{F}$.

Definition 2.2 (Data matrix). Data matrix $X \in \mathcal{F}^{n \times m}$ is a matrix with n rows and m columns, where n is a number of characterized objects described using some set of features $F = \{f_1, f_2, ..., f_m\}$, where m is a number of features and $f_1, f_2, ..., f_m \in \mathcal{F}$.

Definition 2.3 (Dataset). Let $X \in \mathcal{F}^{n \times m}$ be a data matrix. Let $y \in \mathbb{R}^n$. A dataset is a pair:

$$D = \langle X, y \rangle. \tag{2.1}$$

A vector y is called a *target vector*.

Definition 2.4 (Task). A task is a problem characterized by a given dataset $D = \langle X, y \rangle$.

Definition 2.5 (Model). Let D be a dataset. Let A be a machine learning algorithm. A model M_D^A is a function build using the machine learning algorithm A and the dataset D, mapping a data matrix $X \in \mathcal{F}^{n \times m}$ into a vector $z \in \mathbb{R}^n$:

$$M_D^A \colon \mathcal{F}^{n \times m} \to \mathbb{R}^n. \tag{2.2}$$

Definition 2.6 (Measure of performance). A measure of performance p is a function:

$$p \colon \mathbb{R}^n \times \mathbb{R}^n \to [0, 1]. \tag{2.3}$$

The examples of measures of performance are F1, AUC ROC, accuracy.

Definition 2.7 (K-fold cross-validation performance). Let $D = \langle X, y \rangle$ be a dataset, where $X \in \mathcal{F}^{n \times m}$ and $y \in \mathbb{R}^n$. For an index set $\mathcal{I} = \{1, 2, ..., n\}$ let $\{\mathcal{I}_1, \mathcal{I}_2, ..., \mathcal{I}_k\}$, where $k \leqslant n$, be a set of not empty index sets such that $\mathcal{I} = \bigcup_{i=1}^{k} \mathcal{I}_i$. These sets are also called folds. $D_{(-i)} = \langle X_{(-i)}, y_{(-i)} \rangle$, where $X_{(-i)}$ is a data matrix which is created by removing rows with indexes $\mathcal{I}_i$ from data matrix X, and $y_{(-i)}$ is a vector created by removing elements with indexes $\mathcal{I}_i$ from y. Analogically $D_{(i)} = \langle X_{(i)}, y_{(i)} \rangle$, where $X_{(i)}$ is a data matrix which is created by selecting only rows with indexes $\mathcal{I}_i$ from data matrix X, and $y_{(i)}$ is a vector created by selecting only elements with indexes $\mathcal{I}_i$ from y.

A k-fold cross validation performance for a machine learning algorithm A, a measure of performance p, a dataset $D = \langle X, y \rangle$, and a number of folds k, we call a function:

$$CV_A^p(D, k) = \frac{1}{k} \sum_{i=1}^{k} p\Big(M_{D_{(-i)}}^A(X_i), y_i\Big). \tag{2.4}$$

There are different ways of splitting a dataset into folds; for example, we can create folds with an equal number of elements where indexes for each split are chosen randomly. Another way of splitting the dataset is a stratified fold, where each fold has an equal number of elements, and indexes are chosen in a random way, but preserving the same target vector values distribution in each fold.

Definition 2.8 (Transformation). A transformation t we call a function:

$$t \colon \mathcal{F}^{n \times m} \to \mathcal{F}^{n \times q}. \tag{2.5}$$

As applying transformation on a dataset $D = \langle X, y \rangle$, we mean applying the transformation to its data matrix. $t(D) = \langle t(X), y \rangle$

Definition 2.9 (Feature engineering). Let $T = \{t_1, t_2, ..., t_l\}$ be a set of some transformations, where l is a number of transformations. Let $X \in \mathcal{F}^{n \times m}$ be data matrix, Let $\hat{X}$ be a set of columns derived from X by applying all transformations and all possible compositions of transformations from $\mathcal{T}$ on X. The goal of feature engineering is to find such data matrix X^* consisting of original and derived features that maximizes a k-fold cross-validation performance for a given algorithm A and a measure of performance p. Set of columns of data matrix X is denoted by col(X).

$$\begin{gathered} X^* = \underset{X' = [F_1 | F_2]}{\arg\max}\, CV_A^p\Big(\langle X', y \rangle, k\Big), \\ F_1 \subseteq col(X), \\ F_2 \subseteq \hat{X}. \end{gathered} \tag{2.6}$$

Such formulation of feature engineering problem has been presented in [Khurana et al., 2017]. The goal of ML is to build a model basing on known input data, which can be evaluated later on new, unknown at the time of building, data in order to predict a correct target vector. In other words, ML has to learn how to solve a given task. One of the ways to accomplish the goal of solving a given task is to follow a machine learning pipeline. As described in the article "7 Steps of Machine Learning" [Guo, 2017] machine learning pipeline consists of seven steps:

1. **Gathering raw data**

   First of all, we need to collect a bunch of real data, examples of some phenomenon. In this data, patterns and dependencies are detected.

2. **Preparing, transforming and cleaning raw data**

   In order to make the raw data readable by a machine learning model, it has to be properly prepared. That includes applying transformations and removing redundant and not important data.

3. **Selecting ML algorithm and building a model**

   This step includes selecting a proper machine learning algorithm that basing on the input data is capable of solving a given task. For each task, the model selected algorithm and build model can be different.

4. **Training a model**

   The goal of this step is to detect the patterns in given data in order to solve the task. In most cases, it is done in an iterative manner, in multiple training steps.

5. **Evaluating a model**

   After training, the model has to be evaluated on "unseen" data in the training step to check if the task is solved properly and how far it is from the ideal solution.

6. **Optimizing a model - parameter tuning**

   Basing on the evaluation results, if the model does not produce satisfying results, it can be improved by tuning its parameters. Changing the parameters requires retraining the model and reevaluating the model's performance.

7. **Solving task, making predictions**

   Then when we achieved good performance of the model, we can use it to solve the task in a "real world" situations and problems. Of course, during the usage of the model, we can still gather data and then improve the model using new examples.

In each step of the machine learning pipeline, we can go back to one of the previous steps when we are not able to train and optimize the model in order to improve its performance. It is worth noticing that in every step, knowledge and time of well-qualified specialist is required, thus the more automatic the process is, the more cost-efficient and less time consuming it becomes.

Basically, feature engineering is the second step of the machine learning pipeline. In other words, it is building a different representation of available data, which can help machine learning model to produce better results. Usually it requires domain knowledge of the data subject.

2.2.2. Feature engineering techniques

There are a lot of different feature engineering techniques (transformations). In this section, I present the most common techniques that can be used to deal with numeric, date/time, and categorical data.

A. Feature selection

Feature selection is a method of narrowing down the number of features, which leads to a reduced model's complexity. Cleaning out the data and pruning away not important features should be one of the first steps in the feature engineering process. According to [Zheng and Casari, 2018] feature selection techniques falls into three categories:

1. **Filtering methods**
   This kind of feature selection techniques removes features that possibly will not be used by a model. Features that should be filtered out are chosen, for example, by computing variance of each feature, those that fall below some threshold are removed. This type of methods perceives each feature as a separate one without taking into account dependencies between a group of features.

2. **Wrapper methods**
   In contrary to filtering techniques, wrapper methods look for combinations of multiple features. This type of techniques iteratively selects subsets of features and measures performance of the selected subset, then the best group of features is preserved. Thanks to that approach, features considered as uninformative as a single ones, but important with a subset of other features, are not removed. However, wrapper methods are much expensive when it comes to computation power. An example of such method is the Boruta feature selection algorithm [Kursa and Rudnicki, 2010].

3. **Embedded methods**

   Embedded methods preserve a balance between the complexity of wrapper methods and cheapness of filtering technique - they consider a combination of features, but not so expensive as wrapper methods. Feature selection, in that case, is performed in the model training process, and selection is made basing on the used ML algorithm. Thus, chosen features are relevant for the specific model. An example of that method is a decision tree. The decision tree in each node selects a feature, basing on which the node is split. Then removing features that were not used in any split criterion can be considered as not relevant for that model.

B. Missing values imputing

Datasets differ when it comes to data quality. Not always, we may expect to have a good quality dataset without any missing values. Many "real world" datasets can contain NaN, null or blank values, which can have a negative impact on the machine learning model. Some ML algorithms do not even work at all when dataset with missing values is provided. There are some techniques dealing with the problem [Somasundaram and Nedunchezhian, 2011]:

1. **Removing records or features with missing values**

   First and probably the most basic solution is to remove records or features containing missing values. However, it is not always the right way to overcome the problem. The side effect of the removal is reducing some amount of relevant and important data, which can lower the models' performance.

2. **Imputation by a constant value**

   Another way of handling missing data is to replace all unknown data with some constant value, for example, with 0. The constant value can also be chosen basing on known data. In case of numeric feature minimum, the maximum, mean, or median value can be computed, and missing values can be filled with the value. For other types of data e.g., categorical data or numeric data as well, the filling value could be the most frequent value. In that case, no data is lost, however, it can introduce a bias in the data. Moreover, it does not reflect correlations between features, which can be misleading for the model.

3. **Predicting missing value using machine learning models**

   Alternatively, for imputing missing values, machine learning algorithms can be applied. For this purpose, a regression model estimating the missing values can be built. The model trains on known values and then tries to predict unknown values. Moreover, some clustering methods can also be applied. The example of such method is the k nearest neighbors algorithm. The method fills missing values by looking at k most similar records in a given dataset. This methods try to predict unknown features basing not only on one feature but also on other ones. Thanks to that, imputed values are more accurate and fit better in the dataset. Nonetheless, such methods are much expensive in terms of computing power than calculating the median or most frequent value.

C. Feature encoding

Feature encoding is used to transform categorical features. A categorical feature represents some label or category, usually by text or number. When categories are represented by text, we have to encode them to numbers in order to be correctly processed by a model, because usually, ML models deal better with numeric than text data.

1. **Categories as numbers**

   First and the simplest method is assigning for each category a number from 1 to k, where k is a number of categories. However, this method has a side effect - categories will be ordered. It is not always the desired state, for example, having three categories "dog", "mouse", "cat", which are encoded into 1, 2, 3 would create some order in initially disordered data, what can be misleading for a model.

2. **One-Hot encoding**

   One-Hot encoding helps to overcome the ordered categories effect. Instead of numerical values from 1 to k, the sequence of binary values is used. For each category, we create a new feature indicating whether the described entity is in that category or not. Every category is represented as a vector of length k. For example, for three categories: "black", "red", "yellow", there are three new binary features "b", "r", "y" created. The categories are encoded in the following way:

Table 2.1: One-Hot encoding example

Category	**b**	**r**	**y**
black	1	0	0
red	0	1	0
yellow	0	0	1

3. **Dummy encoding**

   In order to encode k categories using One-Hot encoding we have to add k number of features, however, it can be done using $k-1$ features - this is what dummy encoding does (c_1 and c_2 are new features created by the transformations):

Table 2.2: Dummy encoding example

Category	c_1	c_2
black	1	0
red	0	1
yellow	0	0

4. **Target encoding**

   Target encoding uses a target vector to encode categorical features. For every category mean of the corresponding values in the target vector (y) is computed.

Table 2.3: Target encoding example

Category	**y**	**value**
black	1	1
red	1	0.67
yellow	1	0.5
yellow	0	0.5
red	0	0.67
red	1	0.67

D. Feature normalization

Some ML algorithms, such as linear regression or logistic regression, are sensitive to the scale of input values. If some numeric values are too high or too low compared to others, it could have a negative impact on whole models' performance. In that case, features should be normalized or scaled [Zheng and Casari, 2018]. The scaling is usually performed in the scope of a single feature, in other words, we take into consideration only values of the single feature at once. Let $X \in \mathcal{F}^{n \times m}$ be a given data matrix, where n is number of rows and m is number of columns (features), then $X^j = [x^j_1, x^j_2, ..., x^j_n]$, is a j-th column of X which values will be scaled ($j \in [1, m]$)

1. **Min-Max Scaling**

   This type of scaling squeezes or stretches features to the range of $[0, 1]$. Then the min-max scaling formula is defined as follows:

$$\hat{X}^j = \frac{X^j - \min(X^j)}{\max(X^j) - \min(X^j)}. \tag{2.7}$$

2. **Standardization**

   Another way of normalization features is standardization. After applying that technique, the scaled feature has a mean of 0 and a variance of 1. Standardization is formulated as:

$$\hat{X}^j = \frac{X^j - \text{mean}(X^j)}{\sigma}, \tag{2.8}$$

σ - standard deviation of X^j.

E. Mathematical operations

Another example of feature engineering techniques is mathematical operations. Depending on the characteristic of given data and a given task to solve, various mathematical operations can be applied on features, for instance, sum, difference, multiplication, trigonometric functions, hyperbolic functions, polynomials, etc. Basically, any mathematical operation can be used in a feature engineering process. As shown in [Heaton, 2017], some of the machine learning algorithms benefit from applying such transformations.

Some of the mathematical functions such as sum, difference, multiplication require more than one argument. This means that the operation is applied to multiple features at once, which can be useful in finding important correlations and dependencies between features.

Moreover, transforming features using logarithm, root, or exponential operations changes the feature distribution. This type of operation can transform non-linear correlations into linear correlations, which can also be helpful in finding correlations in data. What is more, right or left skewness of data can be reduced by applying such transformations, which can positively influence a model's performance [Zheng and Casari, 2018]. This type of transformations should be used, for example, when using machine learning models that assume that given data has normal distribution e.g., linear regression.

F. Binning

Binning is dividing numeric data into intervals called bins. When dealing with numeric data, the exact value sometimes is not so important. In some cases, more informative is whether the value falls into some rage or if it exceeds some threshold. This means that we can achieve better results after replacing the original values by categorical or numeric features with fewer distinct values than the original ones. Moreover, when collected data is noisy or contains observation errors, using binning can smooth the data and make the model less sensitive to such errors or corrupted data. There are two ways of creating bins [Zheng and Casari, 2018]:

1. **Fixed-width binning**
   In fixed-width binning, each bin contains a specific numeric range of values. The width of bins is defined manually or automatically using, for example, linear or exponential scale. It is fast and easy to compute, however, the disadvantage of that method is that the number of records falling into each bin can significantly differ. In edge cases, some of the bins can be even empty or contain all of the records if the ranges are improperly defined.

2. **Quantile binning**
   In order to reduce the side-effect of fixed-width binning, the quantile binning can be applied. This technique uses quantiles to specify the ranges of every bin adaptively. Quantiles split data into bins having the same number of records in each. The example of quantile is median, which splits data into two parts, each containing half of data. Similarly, quartiles, deciles, and cetiles, etc. can be used to split data into four, ten, or hundred equal bins, respectively. Using this technique, even when having gaps in data, would not create empty bins.

When the data is put into bins, it has to be encoded into features. The bins can be encoded in two ways:

- **categorical** - each bin is treated as a class and relevant label is assigned to each value depending on which bin the transformed value falls into,
- **substitute value** - instead of keeping original value, for each bin substitute value is computed e.g mean, median, min, max etc.

G. Extracting new features

Another useful feature engineering technique is creating new features basing on existing ones. For example, from the date-time feature following new features can be created:

- day of a week, a month, year,
- a week, a month, year number,
- is a weekend, is a working day, is a holiday,
- time of day e.g. a morning, a noon, an afternoon, an evening, day or night.

Feature engineering cannot always be performed in the same way. Set, number and way of creating features are dependent on given data, machine learning model, and task. For example, having a dataset that has all features scaled does not require scaling, but not all datasets are scaled. As stated in [Heaton, 2017], different machine learning models require applying different feature engineering techniques.

Moreover, when it comes to feature selection, the balance in the number of features has to be preserved. The more number of features, the more detailed the feature set is. However, it does not always mean that the feature set is informative, and it results in better performance of the model. More features also mean that the model usually requires more computational power and time to train. What is more, a lot of irrelevant features can be deceptive for a machine learning algorithm. On the other side, fewer features mean that the model could be not able to solve the task at all due to a lack of useful information.

Therefore, feature engineering is a time-consuming process, and hence usually, the cost is high. However, it is an essential step in the ML pipeline and should not be omitted. Thanks to feature engineering, we can make machine learning algorithms to build models achieving better performance.

2.3. Reinforcement learning

2.3.1. Markov Decision Process

In order to use the reinforcement learning technique, we have to model a considered problem as a *Markov Decision Process (MDP)*. There are many different definitions of MDP (see more [A. Feinberg and Shwartz, 2002], [Puterman, 1994], [Sutton and Barto, 2018]); however, they share similar key aspects. In this case, the following definition is used:

Definition 2.10 (Markov Decision Process). A Markov Decision Process is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, p)$, where:

- $\mathcal{S}$ - set of possible states (may be infinite),
- $\mathcal{A}$ - set of possible actions (may be infinite),
- $\mathcal{R}$ - set of possible rewards ($\mathcal{R} \subseteq \mathbb{R}$),
- $p\colon \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ - states transition probability function.

An agent acts in an environment in discrete time steps $t = 0, 1, 2, ...$ At each step the environment is in some state $S_t \in \mathcal{S}$ and the agent performs an action $A_t \in \mathcal{A}$. As a result of the action in the next time step, the agent gains a reward r, and the environment changes its state to

S_{t+1}. State transition and rewards are obtained according to p, which is defined as a conditional probability function:

$$\begin{aligned} p\left(s', r | s, a\right) = \Pr\left\{S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a\right\}, \\ s, s' \in \mathcal{S}, \\ a \in \mathcal{A}, \\ r \in \mathcal{R}. \end{aligned} \tag{2.9}$$

Definition 2.11 (Policy). A policy is defined as a probability distribution over the actions $\mathcal{A}$, given a state $s \in \mathcal{S}$.

$$\pi \colon \mathcal{S} \times \mathcal{A} \to [0, 1] \tag{2.10}$$

In other words, a policy $\pi(s|a)$ returns a probability of performing an action $a \in \mathcal{A}$ when being in a state $s \in \mathcal{S}$. Space of all policies is denoted by Π.

The agent takes actions according to its current policy π, that is, it takes actions with probability defined by the policy π. The policy does not have to remain unchanged during the whole reinforcement learning process. The agent's main goal is to maximize the sum of gained rewards - it means that the agent has to find the best policy, which makes the objective possible to accomplish. As a result of the agent's experience gained during acting in the environment, it is possible to improve the policy.

Definition 2.12 (Return). Let N be a finite number of steps in which an agent reached a final state. A return (G_t) is the expected sum of gained rewards counting from time step t:

$$G_t = r_t + r_{t+1} + r_{t+2} + ... + r_N. \tag{2.11}$$

The above definition assumes that we have a final state which is reached in N steps. However, it is not true in all cases. When we consider continuing tasks in which it is not possible to define a final state, the discounted return should be used.

Definition 2.13 (Discounted return). A discounted return is defined as follows:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}, \tag{2.12}$$

where $\gamma \in [0, 1)$ is a *discount factor.*

The discount factor is introduced to make a return to be a finite value in continuing tasks. For the purposes of this thesis, the discounted return will be considered further.

The goal of the agent is to find the best policy that maximizes a return. In order to be able to compare policies a policy's performance should be measurable. For this purposes the value functions are introduced.

Definition 2.14 (State-value function). A state-value function v_π is expected return value that can be gained when starting from state s and then following policy π while taking further steps:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]. \tag{2.13}$$

Definition 2.15 (Action-value function). An action-value function, also called Q function, for a policy π is a function measuring the expected return for an action a taken in state s and then following the policy in further steps.

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]. \tag{2.14}$$

Both state and action value functions can be formulated as recursive functions with respect to possible successor states s' and successor actions a'. The state-value function can be defined as follows:

$$\begin{aligned} v_\pi(s) =& \mathbb{E}_\pi[G_t|S_t = s] = \\ & \mathbb{E}_\pi[r_{t+1} + \gamma G_{t+1}|S_t = s] = \\ & \sum_{a\in\mathcal{A}} \pi(s|a) \sum_{s'\in\mathcal{S}} \sum_{r\in\mathcal{R}} p(s', r|s, a) \Big[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']\Big] = \\ & \sum_{a\in\mathcal{A}} \pi(s|a) \sum_{s'\in\mathcal{S}, r\in\mathcal{R}} p(s', r|s, a) \Big[r + \gamma v_\pi(s')\Big]. \end{aligned} \tag{2.15}$$

Similar equation can be derived for the action-value function:

$$\begin{aligned} q_\pi(s, a) =& \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \\ & \mathbb{E}_\pi[r_{t+1} + \gamma G_{t+1}|S_t = s, A_t = a] = \\ & \sum_{s'\in\mathcal{S}} \sum_{r\in\mathcal{R}} p(s', r|s, a) \Big[r + \gamma \sum_{a'\in\mathcal{A}} \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s', A_{t+1} = a']\Big] = \\ & \sum_{s'\in\mathcal{S}, r\in\mathcal{R}} p(s', r|s, a) \Big[r + \gamma \sum_{a'\in\mathcal{A}} \pi(s'|a') q_\pi(s', a')\Big]. \end{aligned} \tag{2.16}$$

One policy is better than another when

$$v_{\pi'}(s) \geqslant v_\pi(s), \quad \forall s \in \mathcal{S}. \tag{2.17}$$

Definition 2.16 (Optimal policy). An optimal policy π^* is a policy for which following equations are met:

$$\begin{aligned} v_{\pi^*}(s) &= \max_{\pi \in \Pi} v_\pi(s), \quad \forall s \in \mathcal{S}, \\ q_{\pi^*}(s,a) &= \max_{\pi \in \Pi} q_\pi(s,a), \quad \forall s \in \mathcal{S}, \quad \forall a \ \in \mathcal{A}. \end{aligned} \tag{2.18}$$

The goal of the agent is to find the best policy π^*, in other words, it has to find the policy for which value functions will have the highest values.

Let $v^*(s)$ be a state-value function for the optimal policy. $v^*(s)$ can be formulated using action-value function:

$$v^*(s) = \max_{a \in \mathcal{A}} q_{\pi^*}(s,a), \quad \forall s \in \mathcal{S}. \tag{2.19}$$

We can expand $v^*(s)$ using equations (2.16) and (2.19):

$$\begin{aligned} v^*(s) =& \max_{a \in \mathcal{A}} q_{\pi^*}(s,a) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r | s, a) \Big[r + \gamma \max_{a' \in \mathcal{A}} q_\pi(s', a') \Big] = \\ & \max_{a \ \in \mathcal{A}} \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r | s, a) \Big[r + \gamma v^*(s') \Big]. \end{aligned} \tag{2.20}$$

Let $q^*(s,a)$ be a action-value function for the optimal policy π^*. Using equation (2.16):

$$q^*(s,a) = \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r | s, a) \Big[r + \gamma \max_{a' \in \mathcal{A}} q_\pi(s', a') \Big]. \tag{2.21}$$

Definition 2.17 (Bellman equations). Equations (2.20) and (2.21) are called Bellman equations:

$$\begin{aligned} &(2.20) \quad v^*(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r | s, a) \Big[r + \gamma v^*(s') \Big], \\ &(2.21) \quad q^*(s,a) = \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r | s, a) \Big[r + \gamma \max_{a' \in \mathcal{A}} q_\pi(s', a') \Big]. \end{aligned}$$

The solution for Bellman equations is an optimal policy π^*. If the transition function and thus the value functions are well defined, it is possible to solve the problem in a direct way. When this condition is met we can find π^* using dynamic programming algorithms. However, sometimes the transition function is not known in advance. In that case, the reinforcement learning is a solution.

2.3.2. Reinforcement learning methods

Reinforcement learning can be seen as an extension of dynamic programming algorithms that finds the optimal policy π^*. Because it does not require the transition function to be known in advance, reinforcement learning is more robust than dynamic programming methods. Reinforcement learning extends the classic methods by adding some kind of machine learning. There are two kinds of reinforcement learning methods: *model-free* and *model-based.*

Definition 2.18 (Model-free algorithm). We call a reinforcement learning algorithm to be *model-free* when agent by interacting with the environment learns its dynamics and updates the value functions locally instead of approximating the transition function and building the model of the environment.

A. Monte Carlo methods

Monte Carlo methods relies on approximating a value function v (it can be either state-value or action-value function) for a current policy π. The estimates are done by generating sample episodes (state, action sequences) using current policy π, then averaging returns (sum of gained rewards) for each state. Using computed average returns, the current state-value function v is updated. The more episodes are generated, the more returns for each state are collected, the state-vale function v is more accurate. However, the goal of the algorithm is to find the best policy π^*, not just estimating the value function v for the current policy π. In order to find the best policy π^*, current policy π. has to be updated. This kind of methods works according to the idea of generalized policy iteration (GPI). The idea consists of two steps, first of all, the current value function v for current policy π is approximated, then, by making steps in a greedy way according to the current value function v, a policy π is improved. Monte Carlo methods are simple, however, they require many iterations to convergence to the optimal policy π^*.

B. Temporal difference methods

Temporal difference (TD) methods are a class of methods that combine both Monte Carlo (MC) and Dynamic Programming ideas. As the Monte Carlo methods, the TD methods are based on experience gained by the agent during performing actions in the environment, which dynamics is not known in advance. Similarly to MC, TD methods relies on approximating the value function v, and the policy π according to gained experience. This means that both MC and TD methods are kind of generalized policy iteration methods. The difference between that two types of algorithms is in the moment of updating the value function v. In MC the value function is updated when the rewards during the whole episode are collected, in other words, when having the accumulated reward (return) for the whole episode. On the other side, in TD methods the value function v is updated immediately after taking an action without the need to wait for the whole episode to finish. Straight after receiving the reward of taken action, the value function is improved by the error made by value function at that time:

$$\begin{aligned} & v(s) \leftarrow v(s) + \alpha[r + \gamma v(s') - v(s)], \\ & s \in \mathcal{S}, \\ & s' \in \mathcal{S} \quad - \quad \text{state after taking an action in state } s, \\ & r \in \mathcal{R} \quad - \quad \text{reward obtained by transition from } s \text{ to } s', \\ & \gamma \in (0,1] \quad - \quad \text{discount factor}, \\ & \alpha \in (0,1] \quad - \quad \text{learning parameter}. \end{aligned} \tag{2.22}$$

One of the most widely used temporal difference algorithm is *Q-learning* algorithm [Watkins, 1989]. The algorithm is based on learning an action-value function q, not a state-value function. In each step of the algorithm the action-value function is updated in the following way:

$$\begin{aligned} & q(s,a) \leftarrow q(s,a) + \alpha[r + \gamma \max_{a' \in \mathcal{A}} q(s',a') - q(s,a)], \\ & a \in \mathcal{A}, \\ & s \in \mathcal{S}, \\ & s' \in \mathcal{S} \quad - \quad \text{state after taking action a in state } s, \\ & r \in \mathcal{R} \quad - \quad \text{reward obtained by transition from } s \text{ to } s', \\ & \gamma \in (0,1] \quad - \quad \text{discount factor}, \\ & \alpha \in (0,1] \quad - \quad \text{learning parameter}. \end{aligned} \tag{2.23}$$

Moreover, the key concept of the *Q-learning* is learning optimal action-value function q^* directly, rather than trying to build the optimal policy π^* . Thus next steps in the learning process are chosen using policy, which is derived straightforwardly from the current action-value function q. Having the optimal action-value function q^* defined, choosing the highest-valued action in the current state is an equivalent of acting under the optimal policy π^*. However, when the current action-value function q is not the optimal action-value function q^*, we cannot always choose the best action (act under current policy derived from q). It could lead to find local optimum for the action-value function q, not necessarily the optimal q^*. In order to not get stuck in local optimum, random action is chosen with some given probability ϵ and action under current policy is selected with probability $1 - \epsilon$. This way of choosing the next action is called *ϵ-greedy*. Thanks to that, we can preserve exploitation vs. exploration balance.

The schema of the *Q-learning algorithm*:

Algorithm 1: Q-learning algorithm

```
1  $\forall s \in \mathcal{S}, a \in \mathcal{A} \quad q(s,a) \leftarrow$ arbitrary value (e.g. 0)
2  foreach episode do
3      $s \leftarrow$ initial state
4      foreach step of episode do
5          Choose an action $a$ for state $s$ using policy derived from $q$ function ($\epsilon$ - greedy)
6          Take action $a$ in state $s$
7          $r, s' \leftarrow$ reward and next state (results of taking action $a$ in $s$)
8          $q(s,a) \leftarrow q(s,a) + \alpha[r + \gamma \max_{a' \in \mathcal{A}} q(s',a') - q(s,a)]$
9          $s \leftarrow s'$
10     end
11 end
```

Definition 2.19 (Model-based algorithm). We call a reinforcement learning algorithm to be *model-based* when agent by interacting with the environment, learns its dynamics, and approximates the transition function, in other words, builds the model of the environment online.

A. Dyna-Q

An example of a model-based algorithm is the *Dyna-Q* algorithm [Sutton and Barto, 2018]. The key part of the Dyna-Q algorithm is building a model of an unknown environment. The algorithm consists of four processes occurring continuously, that are planning, acting, model-learning, and updating action-value function. The agent in the Dyna-Q is taking steps (acting) according to the current learned value function in ϵ-greedy way. After taking action in the current state, the agent observes the next state and gained reward, then it builds the model basing on observed results (model-learning). The model is stored in a tabular form - for each observed state and taken action in that state gained reward and following state is remembered. Then the planning process takes place. Actions and states are sampled randomly from all previously observed states and using the learned model over time, the reward and following state is obtained. Afterward, using simulated experience, the action-value function is updated. In order to use the Dyna-Q algorithm, the environment must be deterministic.

2.3.3. Linear approximation of value functions

The value functions can be represented in tabular form. Notwithstanding, when having a lot of states or actions (sometimes tends to infinity), it is not possible to store all the values for all states or all state, action pairs. In order to be able to deal with huge or even infinite problems, we have to approximate the value function as a parametrized function with fewer parameters than the number of actions or states. One of the ways of approximating the value function is approximating it using a linear function.

Definition 2.20 (State vector). Let $f_1, f_2, ..., f_d$ be functions that maps state to a real value:

$$f_1, f_2, ..., f_d : \mathcal{S} \to \mathbb{R}. \tag{2.24}$$

The $f(s) \in \mathbb{R}^d$ vector is called a *state vector* when:

$$f(s) = (f_1(s), f_2(s), ..., f_d(s)). \tag{2.25}$$

In other words, a *state vector* is a vector that describes a state using a set of real values. The way of representing a state makes it possible to perform computations in that state. We can use a *state vector* to approximate a value function for the state.

Definition 2.21 (Linear approximation). Let $w \in \mathbb{R}^d$ be a weight vector. Let $f(s)$ be a state vector that corresponds to a state s. The *linear approximation* of a value function for state s we call a function which is defined as inner product between w and $f(s)$:

$$\hat{v}(s, w) = w^T f(s) = \sum_{i=1}^{d} w_i f_i(s). \tag{2.26}$$

Monte Carlo and Temporal Difference methods rely on approximating the value functions over time, during agent performing actions in not known environment. In order to have a more accurate linear approximation over time, the weights have to be updated in every time step. Let w_t be a weight vector in step $t = 0, 1, 2, ...$ A common method of updating the weight vector is stochastic gradient descent (SGD). General schema of updating the weight vector in case of SGD is:

$$w_{t+1} = w_t + \frac{1}{2}\alpha[v_\pi(s_t) - \hat{v}(S_t, w_t)]^2 = w_t + \alpha[v_\pi(s_t) - \hat{v}(S_t, w_t)]\nabla\hat{v}(s, w). \tag{2.27}$$

The gradient of the linear approximation with respect to w is:

$$\nabla\hat{v}(s, w) = f(s). \tag{2.28}$$

In case of temporal difference methods instead of explicitly updating the value functions, value functions are updated indirectly by improving weights of linear approximation. For state-value function:

$$w_{t+1} = w_t + \alpha[r_{t+1} + \gamma w_t f(s_{t+1}) - w_t f(s)]f(s). \tag{2.29}$$

The similar equation we can create for action-value function (Q-learning). In that case, the state vector is not only dependent on a state, but also on an action. Analogically, the update is done in the following way:

$$w_{t+1} = w_t + \alpha[r_{t+1} + \gamma \max_{a' \in \mathcal{A}} w_t f(s_{t+1}, a') - w_t f(s, a)]f(s, a). \tag{2.30}$$

Accoridng to [Irodova and Sloan, 2005] in order to ensure the convergence of the algorithm, features has to be normalized.

2.4. Summary

In this chapter, the concepts of feature engineering and reinforcement learning are introduced. Feature engineering is an important part of the machine learning pipeline. Improving the quality of the dataset and transforming it into a more understandable representation for ML models can drastically increase the model's performance without the need to tune the parameters. However, there are a lot of various feature engineering techniques that can be applied in different situations, depending on what kind of data the model has to deal with. Due to that, feature engineering is usually a time-consuming task, and it requires domain knowledge of specialists in order to achieve satisfactory results. Nonetheless, in many cases, good quality of data is key to success.

In order to use the reinforcement learning technique to solve a problem, it has to be modeled as a Markov Decision Process. Then explicit or approximate value functions should be defined to find the solution, which is an optimal policy. Value functions can be represented in tabular form. However, when there are a lot of states or actions defined, approximations of value functions have to be applied. There exist numerous algorithms than can be used to compute the best policy. The algorithms are divided into two categories, model-based, e.g., Dyna-Q and model-free, e.g., Monte-Carlo, Temporal Difference Methods. Model-based methods are building a model of the environment when learning. On the other side, model-free methods try to approximate value functions instead of creating the environments' model. Depending on the characteristic of a problem (environment, states, actions), a proper type of algorithm is chosen. For the purposes of this thesis, I have chosen the Q-learning algorithm - explanation in chapter 3.

3. Feature engineering as a reinforcement learning problem

3.1. Introduction

This chapter presents how to automate the feature engineering process using a reinforcement learning technique. The proposed concept is based on the approach described in the article *Feature Engineering for Predictive Modeling using Reinforcement Learning* [Khurana et al., 2017].

First of all, used notation and a problem representation are presented. The next step is transforming a feature engineering problem into a problem that can be modeled as a *Markov Decision Process* in order to be able to apply the reinforcement learning technique. The goal is to automate the process of feature engineering.

3.2. Notation and problem representation

Definition 3.1 (Transformation graph). A transformation graph is a directed acyclic graph, where each node represents a dataset, which can be an initial dataset D_0 (root node) or dataset derived from its parent by applying a transformation on the parent's dataset. Each edge represents such transformation, the direction of the edge is from parent to derived node.

Having an initial dataset D_0 and a defined finite set of transformations $\mathcal{T}$, the feature engineering problem can be represented as a transformation graph. The visual representation of the graph is shown in section 4.3.1.

Definition 3.2 (Score). For a machine learning algorithm A and measure of performance p (e.g. F1) a score of a node n is the k-fold cross-validation performance of a dataset $D = \langle X, y \rangle$ represented by the node n. The score is denoted by $S_A^p(n)$:

$$S_A^p(n) = CV_A^p(D, k). \tag{3.1}$$

A node with the highest score is the solution of the feature engineering problem defined as a transformation graph. However, it is computationally impossible to build a complete graph and choose the best node. First of all, the same transformation can be applied repeatedly on derived nodes, which makes the complete graph to have an infinite number of nodes. Moreover, even when the height of the graph (longest path from the root to any other node) is constrained, the number of nodes and corresponding edges grows exponentially, thus computation of cross-validation score across the entire graph is very expensive. Therefore the graph has to be built and explored in a heuristic manner. Additionally, in this thesis, the focus is to find possibly the best node in a limited budget. In terms of budget, time, or the number of iterations are considered.

The outline of transformation graph exploration is presented in [Khurana et al., 2017]

Algorithm 2: Transformation graph exploration

```
Data: Dataset $D_0$, Budget $B_{max}$, Set of transformations $\mathcal{T}$, measure $p$, algorithm $A$
1 Initialize $G_0$ with root $D_0$
2 while $i < B_{max}$ do
3     $b_{ratio} = \frac{i}{B_{max}}$
4     $\langle n^*, t^* \rangle \leftarrow \underset{\langle n,t \rangle \in C_i}{\arg\max} R(G_i, \langle n, t \rangle, b_{ratio})$
5     $G_{i+1} \leftarrow$ Apply $t^*$ to $n^*$ in $G_i$
6     $i \leftarrow i + 1$
7 end
Result: $\underset{n \in \theta(G_i)}{\arg\max} S_A^p(n)$
```

G_i – graph at step i,

$\theta(G_i)$ – nodes of graph at step i,

$\lambda(n, n')$ – transformation used to create node n' from n,

$C_i = \{\langle n, t \rangle : n \in \theta(G_i) \wedge t \in \mathcal{T} \wedge \nexists n' \in \theta(G_i) \ \lambda(n, n') = t\}$ – set of pairs of node and transformation which hasn't been already applied to that node.

The algorithm presents the general methodology of building and exploring the transformation graph. Until the budget is not exhausted, at each step, possible actions are compared using R function. After selecting the best possible node and transformation, the transformation is applied to the selected node, causing the creation of a new node. The result is the best node, in terms of score, with the corresponding dataset. It is worth noticing that the key part of the algorithm is the R function. In every step, according to the function, the decision is made, which transformation on which dataset should be applied in the current graph state. Overall exploration depends on the definition of the R function.

3.3. Applying reinforcement learning

The graph exploration process can be modeled as *Markov Decision Process*.

1. **States**

   A state $s_i = \langle G_i, b_{ratio} \rangle$ is dependent on a step i and consists of two factors: a graph at step i and the remaining budget ($b_{ratio} = \frac{i}{B_{max}}$). The entire space of states is $\mathcal{S} = \bigcup_{i=1}^{\infty} s_i$.

2. **Actions**

   An action a $= \langle n, t \rangle \in C_i$ is a pair of an existing node in the graph G_i and a possible transformation. The entire space of actions is $\mathcal{A} = \bigcup_{i=1}^{\infty} C_i$.

3. **Rewards**

   The entire space of rewards is $\mathcal{R} = [0, 1]$.

4. **States transition probability function**

   Let G_i be a transformation graph at step i, $s = \langle G_i, \frac{i}{B_{max}} \rangle$, $a = \langle n, t \rangle$, $n \in \theta(G_i)$, $t \in C_i$, G_{i+1} - a graph created by applying a transformation t on a node n, $n' = \theta(G_{i+1}) \setminus \theta(G_i)$. For a state s and action a, the states probability transition function is defined as follows:

$$p\left(s', r | s, a\right) = \begin{cases} 1, \quad \text{when} & \begin{aligned} r &= \max_{n' \in \theta(G_{i+1})} S_A^m(n') - \max_{n \in \theta(G_i)} S_A^m(n) \\ s' &= \left\langle G_{i+1}, \frac{i+1}{B_{max}} \right\rangle \end{aligned} \\ 0, \quad \text{otherwise.} \end{cases} \tag{3.2}$$

   In other words, taking the action a in the state s causes a transition to the state s' and obtaining the reward r.

Having the graph exploration problem modeled as the MDP, it is possible to solve it using reinforcement learning. The goal of reinforcement learning is to find the best policy π^*. In that case *Q-learning* method is used. Moreover, instead of learning Q-function directly, because the size of S is infinite, Q-function is approximated using a linear approximation:

$$\begin{aligned} &Q(s, a) = w^t \cdot f(s, a), \quad \text{where} \\ &f(s, a) \text{ - a state vector,} \\ &w \text{ - a weight vector dependent on used transformation,} \\ &t \text{ - a part of action } a = \langle n, t \rangle. \end{aligned} \tag{3.3}$$

state vector is dependent on state $s = \langle G_i, b_{ratio} \rangle$ and an action $a = \langle n, t \rangle$ and consists of such factors:

1. The node n's score ($score(n)$).
2. The transformation t's average of gained immediate rewards so far ($avg\_rew(t)$).
3. The number of times a transformation t has been used so far from the root node to n ($count(t, n)$).
4. The difference between the node n's and its parent node n_{parent}'s scores ($gain(n, n_{parent})$).
5. The difference between the node n's parent n_{parent}'s and its grandparent $n_{grandparent}$ node's scores ($gain(n_{parent}, n_{grandparent})$).
6. The node n's depth ($depth(n)$).
7. The exhausted budget (b_{ratio}).
8. The ratio of number of features in the node n's dataset and the node with highest number of features - n_{ratio}.
9. Is the transformation t a feature selection? ($is\_fs(t)$).
10. Does the node n's dataset contain numerical features? ($contains\_num(n)$).
11. Does the node n's dataset contain datetime features? ($contains\_date(n)$).
12. Does the node n's dataset contain categorical features? ($contains\_categorical(n)$).
13. Does the node n's dataset contain missing values? ($contains\_missing(n)$).
14. Is the transformation t a missing value imputation? ($is\_imputing(n)$).

Performing an action $a = \langle n, t \rangle$ in a state s_i results in transition to a state s_{i+1} and a reward r_{i+1}. The update of a weight vector is done after each transition in a following way:

$$w^t = w^t + \alpha[r_{i+1} + \gamma \max_{a' \in C_{i+1}} Q(s_{i+1}, a') - Q(s_i, a)] f(s, a). \tag{3.4}$$

3.4. Summary

The chapter describes how to model the transformation graph exploration as The chapter describes how to model the transformation graph exploration as *Markov Decision Process.* In order to find the optimal policy, the reinforcement learning technique is used - Q-learning with linear action-value function approximation. Moreover, in order to use the method state had to be modeled as a state vector. The next step is to provide learning data for the model and learn the best policy in order to perform feature engineering in an efficient and automatic way.

4. Software solutions

4.1. Introduction

Automatic machine learning is an area of interest for many engineers and scientists. Due to that, there were many tools created, facilitating and automating the machine learning process. This chapter briefly describes some of the existing solutions dealing with the AutoML area, especially the automation of the feature engineering process. Moreover, as a result of the thesis, an implementation of the algorithm described in the previous chapter is created. The chapter contains technical documentation of the tool along with the user manual.

4.2. Existing tools

4.2.1. TPOT

Tree-based Pipeline Optimization Tool (TPOT) for Automating Data Science [Olson et al., 2016] is a tool that automatically designs the whole machine learning pipelines using genetic programming. The tool performs data cleaning, feature engineering, model selection, and parameter optimization. TPOT generates multiple tree-based pipelines and then optimizes it using genetic programming to maximize the performance of the pipeline for given data.

In a feature engineering step, TPOT performs various types of numeric feature scaling e.g., min-max scaling, standardization. Moreover, as a feature selection, it uses a recursive feature elimination strategy (wrapper method), a strategy selecting top k features, top n percentile of features, and features that are above the minimum variance threshold. Additionally, a variant of Principal Component Analysis is used as a feature decomposition [Martinsson et al., 2011]. The selection of these feature engineering techniques is also done as a part of improving the machine learning pipeline using a genetic programming algorithm. It is worth noticing that changing the machine learning algorithm or doing a parameter optimization in some cases requires different prepossessing of data.

4.2.2. Auto-WEKA

Auto-WEKA [Thornton et al., 2013] uses Bayesian optimization to automate building machine learning pipeline, which is feature engineering, model selection, and hyper-parameter tuning. When it comes to feature engineering, Auto-WEKA supports a wide range of complex feature selection methods. Other feature engineering techniques are not used.

4.2.3. Auto-sklearn

Auto-sklearn [Feurer et al., 2015] is a tool performing automatic machine learning. It automates the process of feature engineering, model selection, and hyper-parameter tuning. For that purpose, it uses Bayesian optimization, meta-learning, and ensemble construction.

Auto-sklearn uses 14 types of feature engineering methods, such as one-hot encoding, missing-values imputation, feature selection. The tool, similarly to Auto-WEKA, uses Bayesian optimization methods to chose which technique should be applied for the current pipeline.

4.3. Implemented solution

4.3.1. Overview

The created python package is called *AFERL* (**A**utomatic **F**eature **E**ngineering using **R**einforcement **L**earning). The solution is available as open-source software under GNU General Public License v3.0 at URL: `https://github.com/wasilewskil/AFERL`. It can be used as a tool that automates the feature engineering process or as a guide, which shows possible ways of dataset transformation and its influence on the machine learning model's performance. After performing a training process, in the first case, AFERL transforms dataset using a set of defined transformations, in the second case, it produces a transformation graph on which the different transformation pipelines and theirs influence on the performance can be observed (Figure 4.1).

4.3.2. Used libraries and technologies

1. **Python** [Van Rossum and Drake Jr, 1995] - high-level programming language.

2. **Numpy** [Oliphant, 2006] - a python package providing efficient data computation methods and data representation structures.

3. **Scikit-learn** [Pedregosa et al., 2011] - a python package providing a lot of useful tools in an area of machine learning such as machine learning models, feature engineering tools, and more.
4. **Graphviz** [Gansner and North, 2000] - a graph visualisation tool.
5. **Scipy** [Jones et al., 2001] - a tool used for scientific computing, moreover, it provides methods of reading datasets from a file.
6. **Pandas** [McKinney et al., 2010] - a package used for manipulating and analyzing data.

4.3.3. User manual

A. FeatureEngineer class

FeatureEngineer is the main class of the solution. It is used for performing an automatic feature engineering process - handles both training and transforming processes.

```
class aferl.feature_engineer.FeatureEngineer(estimator, max_iter=100,
learning_rate=0.1, discount_factor=0.99, epsilon=0.15, h_max = 8,
cv=5, w = None, datetime_format = None, w_init = np.zeros(14),
random_state = None, scoring = 'f1_micro', transformations = None)
```

Table 4.1: FeatureEngineer class parameters

Parameter	**Description**
estimator (required)	Estimator (machine learning model) that has fit() and predict() methods. Used for calculating a node's score (e.g., LogisticRegression from the scikit-learing package).
max_iter (optional)	The maximum number of iterations (budget). It can be a single number or an array of numbers. In case of array, training process is performed multiple times with budgets specified in the array. Default value: 100.
learning_rate (optional)	Defines a learning step size in each algorithm's iteration. Should be a value form (0, 1). Default value: 0.1.
discount_factor (optional)	Discount factor used in the value function. Should be a value from (0, 1). Default value: 0.99.
epsilon (optional)	Probability of taking a random step (not according to current policy). Should be a value from [0, 1]. Default value: 0.15.
h_max (optional)	The maximum height of transformation graph. Default value: 8.
cv (optional)	The number of folds used in cross validation. Default value: 5.
w (optional)	A dictionary with weights learned before. Key: the name of transformation, Value: a numpy array with 14 numeric values representing weights for a value funtion.
datetime_format (optional)	Date-time format used in given dataset e.g "%a %b %d %H:%M:%S %Y"
w_init (optional)	The initial weights value used for all transformations. Ignored when the **w** parameter is provided. Default value: numpy.zeros(14).
random_state (optional)	Seed passed to pseudo number generators used in package.
scoring (optional)	The scoring function used for calculating performance in cross validation. Accepts all scoring functions allowed for the scikit-learn library. Default value: 'f1_micro'
transformations (optional)	List with transformations that will be used in training and transforming processes. Allowed transformations: 'fsp20_fs', 'fsp50_fs', 'fsp75_fs', 'ite_imp', 'sim_imp', 'knn_imp', 'log', 'sqrt', 'square', 'sin', 'cos', 'tanh', 'aggregation', 'sigmoid', 'minmax', 'zscore', 'sum', 'diff', 'mul', 'div', 'ohe_enc', 'bin_nom', 'bin_mean', 'tbi'. Explanation of the symbols in section B.. By default all transformations are used.

Table 4.2: FeatureEngineer class methods

Method	Description
`fit(X, y, data_info, missing_value_mark)`	Method used for a training process. **X** - a numpy array representing features of dataset. **y** - a numpy array containig target vector. **data_info** - list containing information about the type of features. It should have length equal to the number of features. Accepted values: 'numeric', 'nominal', 'datetime', 'boolean', 'timestamp'. **missing_value_mark** - a character, number or string that defines missing value in provided data. (optional) Returns learned weights.
`transform(X, y, missing_value_mark)`	Method used for performing actual feature engineering. Before using that method fit() must be called. Parameters similar as above. Returns transformed dataset.
`save_transformation_graph(path, filename)`	Saves a transformation graph in the .pdf format in a given path with a given filename. Before using that method fit() must be called.
`save_weigths (path, filename)`	Saves current weights to a .json file. Before using that method fit() must be called. Parameters similar as above.

B. Implemented transformations

There are implemented various types of transformations that can be used with different types of data. They can be divided into seven categories.

1. **Feature selection transformations**

   This class of transformations is implemented using `SelectPercentile` class from scikit-learn library. It is a filtering method that filters features basing on the scoring function. In that case, a function `f_classif` is used as a scorer, which computes the value of ANOVA F-test. Basing on that score best x percent of features are selected. Currently there are implemented transformations selecting 20% ("fsp20_fs"), 50% ("fsp50_fs") or 75% ("fsp75_fs") of features.

2. **Missing value imputing transformations**

   There are implemented three types of missing values imputation:

   - Filling missing values with the median for numeric features and with the most frequent value for other types of features ("sim_imp").
   - Imputing using the KNN algorithm ("knn_imp").
   - Imputing using a regression model (see more `IterativeImputer` from the scikit-learn library). It predicts missing values in an iterative manner by building a regression model with the usage of all features ("ite_imp").

3. **Encoding transformations**

   The next class of transformation is encoding transformation used to encode categorical data into more meaningful features. In this case, implemented transformation uses one hot encoding technique ("ohe_enc").

4. **Binning transformations**

   Binning transformations divide numerical values into bins. Implemented transformations use quantile binning methods that divide numeric features into 10 bins, each having the same number of values. Then every value in the bin can be encoded in one of two implemented ways:

   - by mean of edges of the bins ("bin_mean"),
   - by categorical (nominal) feature ("bin_nom").

5. **Scaling transformations**

   In order to scale numerical features there are used two methods of scaling:

   - min-max scaling ("minmax"),
   - standardization ("zscore").

6. **Math transformations**

   Moreover there are implemented transformations applying mathematical functions onto numerical features: sine ("sin"), cosine ("cos"), hyperbolic tangent ("tanh"), logarithm ("log"), square root ("sqrt"), square ("square"), sigmoid function ("sigmoid"), summation ("sum"), difference ("diff"), multiplication ("mul"), division ("div"). The last four listed transformations takes all possible pairs of columns and performs respective mathematical operations.

7. **Others**

   Apart from transformations described above, there is implemented a date-time feature extraction method ("tbi") which separates date-time into: day of the year, day of the week, year, month, day of month, hours, minutes, seconds. Moreover, it is possible to use aggregation ("aggregation"), which adds features counting minimum, maximum, mean, and standard deviation of all numerical values for each row.

Applying a feature selection transformation on a parent's node results in a new node representing a dataset with a data matrix with fewer features than the parent's one. Missing value imputation transformation applied on a parent's dataset results in a dataset with a data matrix with the same number of columns and rows, but with missing values replaced. Every other transformation adds new features to the parent's dataset's data matrix. Applying a transformation results in a dataset created from the parent's by adding new features to its data matrix.

Moreover, after applying each transformation, duplicate columns are removed. This operation is done to limit the number of columns, especially that they do not add any useful information to the dataset and can even make the model harder to train.

Machine learning models used for experiments were implemented as a part of the *scikit-learn* package [Pedregosa et al., 2011]. These algorithms accept only numerical features, so in order to be able to train the models, the input dataset has to be initially transformed. Date-time features are encoded to timestamp, and categorical features are encoded using label encoding - every string receives an integer value, missing values are replaced with 0. The initial transformations' results can be changed further by other transformations e.g., places when previously were zeros are remembered and can be transformed using other imputing transformation. Similarly, categorical values encoded as numbers can be transformed into boolean vectors using One Hot Encoding.

C. Training

If weights for a state vector are unknown, first step for performing automatic feature engineering is a training process. If weights were previously calculated and can be provided in the FeatureEngnineer constructor. In that case the provided weights will be a starting weights for a training process. Training process steps:

1. Choose and construct a proper estimator object (machine learning model).

2. Construct a FeatureEngineer object with chosen parameters.

3. Load a dataset and an array with its column types for training.

4. Call the fit() method with proper paramters.

5. (optional) Call save_weights() method in order to save learned weights to use them later.

6. (optional) Call save_transformation_graph() in order to save and inspect current transformation graph.

In order to train on multiple datasets, fit() method has to be called for every training dataset for same FeatureEngineer object or when not training on all datasets at once, after creating new FeatureEngineer object with previously calculated weights. Every call to fit() method creates a new graph, but preserves learned weights.

D. Transforming

Transforming process steps:

1. Choose and construct a proper estimator object (machine learning model).

2. Construct a FeatureEngineer object with chosen parameters.

3. Load a dataset and an array with its column types for feature engineering (transforming).

4. (optional) If weights were not provided in the previous step, perform training process using datasets different than the transformed dataset.

5. Split the loaded dataset into train and test rows.

6. Call fit() method using train rows in order to build a transformation graph.

7. Call transform() method for whole dataset.

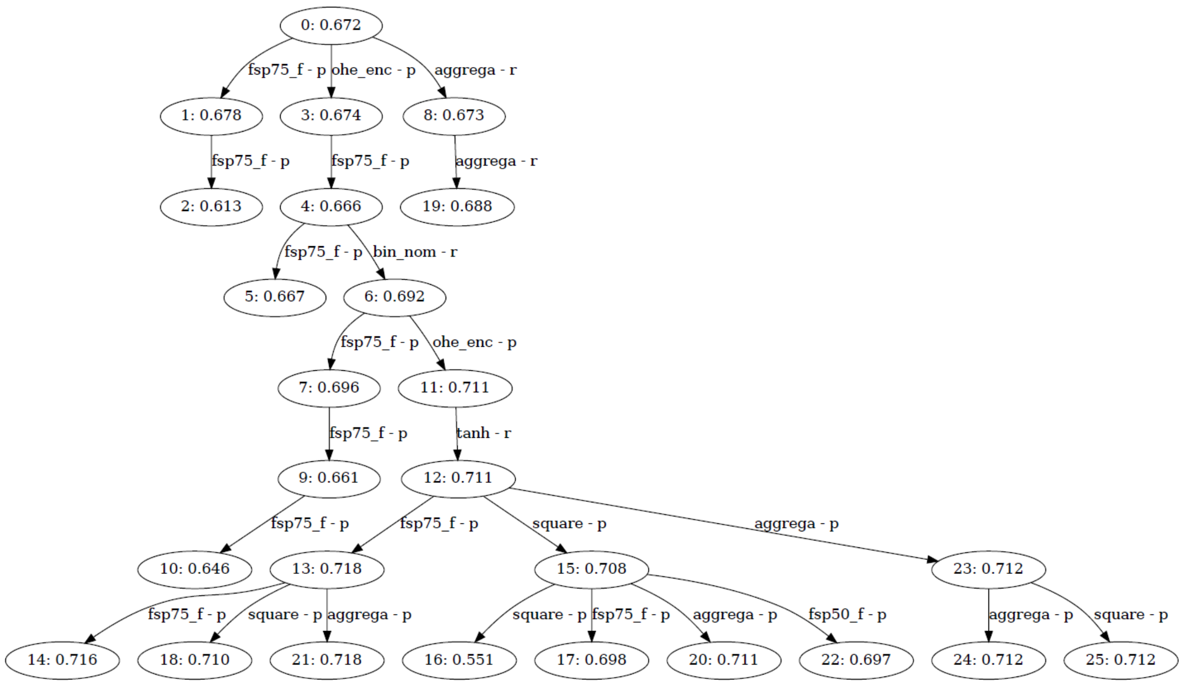
8. Use transformed training data for training estimator and transformed test data to, for example, measure performance of estimator after performing feature engineering.

9. (optional) Call save_transformation_graph() in order to save and inspect current transformation graph.

E. Transformation graph

After calling fit() method it is possible to save current transformation graph using save_transformation_graph() method. Thanks to that user can analyze the transformation pipelines and their influence on the model performance. The results of the analysis can be used as a hint in creating a feature engineering pipeline manually. Every node in the graph represents the dataset and shows its score. The root node contains an initial score and dataset. Every edge is an equivalent of applied transformation to the parent node. Moreover, it contains an information whether child node (the result of applying edge's transformation on the parent) was created using a current policy (p) or by random step (r).

Figure 4.1: Example of a transformation graph



F. State vector

Every state is described by a state vector $f(s, a)$ that consists of fourteen factors. In order to ensure convergence of the algorithm every factor has to be normalized. Otherwise, some of the factors will dominate over another, for example, first factor - node's score is a real number from 0 to 1, however node's depth, ratio of features, transformations' usage count can potentially be an infinite value. Taking into consideration that aspect state vector is constructed in a following way:

1. $score(n) \in [0, 1]$,
2. $avg\_rew(t) \in [0, 1]$,
3. $count(t, n)/depth(n) \in [0, 1]$,
4. $gain(n, n_{parent}) \in [-1, 1]$,
5. $gain(n_{parent}, n_{grandparent}) \in [-1, 1]$,

6. $\text{-}depth(n)/\max_{n'\in\mathcal{N}} depth(n') \in [-1,0]$,
7. $b_{ratio} \in [0,1]$,
8. $n_{ratio} \in [0,1]$,
9. $is\_fs(t) \in \{0,1\}$,
10. $contains\_num(n) \in \{0,1\}$,
11. $contains\_date(n) \in \{0,1\}$,
12. $contains\_categorical(n) \in \{0,1\}$,
13. $contains\_missing(n) \in \{0,1\}$,
14. $is\_imputing(n) \in \{0,1\}$.

Moreover, every feature value was divided by maximum sum of features, that is 14.

In general, every transformation adds new features to parents dataset, except from feature selection and missing value imputation transformations. In order to reduce the number of columns, then thanks to that, reduce the time of training and consumed memory, the feature selection transformations have to be used pretty often. For that purpose, factor no. 9 is introduced, hence that type of transformation has an additional score in Q-value function. It makes that feature selection is used more frequently.

Another enhanced group of transformations is imputing transformations (factor no. 14). Due to the fact that, this type of transformations cannot be used more than once in the path from root to any other node (used once makes the dataset to not contain any missing values, so there is nothing to impute further), it is hard to learn appropriate weights for missing value imputers. There would be not enough training steps using this group of transformations, so in order to produce more examples of using imputers, the factor no. 14 is a part of a state vector.

4.4. Summary

There exist solutions that perform a feature engineering process, however, they do not use the reinforcement learning technique. Instead of RL, techniques such as genetic programming and Bayesian optimization are in use. Contrary to existing solutions, a tool, which is the result of this thesis, uses the reinforcement learning technique to automate the feature engineering process.

5. Experiments

5.1. Preliminaries

The experiments were performed on 36 datasets got from OpenML [Vanschoren et al., 2013] project. The datasets were divided into two groups:

- Group A - 25 datasets used for learning weights (presented in table 5.1),
- Group B - 11 datasets used for testing the behavior of the implemented solution (presented in table 5.2).

Machine learning algorithm used for experiments was *Logistic Reggression* from *scikit-learn* package with the maximum number of iterations equal to 10000 and *saga* solver. Performance during training (building transformation tree) was measured using Stratified 5 fold cross validation strategy with *F1* scoring function.

In order to reduce the time of the experiments, not all of the implemented transformations were used - excluded transformations: time binning, summation, difference, multiplication, division, aggregation.

Definition 5.1 (Optimal node). Let G be a transformation graph, $\theta(G)$ - nodes (datasets) in the graph G, p measure of performance, A machine learning algorithm. The optimal node in graph G is node n^*:

$$n^* = \underset{n \in \theta(G)}{\arg\max}\, S_A^p(n). \tag{5.1}$$

Definition 5.2 (Excessively explored node). Let G be a transformation graph, $\theta(G)$ - nodes of the graph G, Let k be maximum number of graph exploration iterations (budget). Let $n_0 \in \theta(G)$ be a root node of a transformation graph G. Let $n_1, n_2, ..., n_{i-1}, n_i^*, n_{i+1}, ..., n_k \in \theta(G)$ be a sequence in which the nodes were explored in a transformation graph. Let i be an index of the optimal node n_i^*. Let P^* be a set of nodes on a path from the root node n_0 to the node n_i^*. A node n_j is excessively explored when $j < i$ and $n_j \notin P^*$.

The efficiency of the graph exploration is measured in number of excessively explored nodes. The less excessively explored nodes exists in the graph, the more efficient exploration is. Low number of such nodes means that the optimal node was found quicker.

5.1.1. Testing procedure

The behavior testing was performed on 11 OpenML datasets from Group B. Every dataset is randomly split three times using stratified fold with proportions 70% to 30%. The transformation graph is built on 70% of the data. The rest 30% of data is used for validation and calculating e.g., score gain and number of excessively explored nodes. The results of performed experiments presented in the experiments below are averaged over every three folds for a dataset. Thanks to splitting every dataset three times and calculating an average, the results of performed experiments are less dependent on the randomness of the division of the datasets.

5.2. Learning weights analysis

First of all, the reinforcement learning algorithm has to learn the weights for the Q-value function (equation 3.3) in order to explore the transformation graph in an efficient way. The purpose of the experiment is to check if the transformation graph exploration is performed more efficiently as the number of datasets used for learning weights increases.

The learning weights process was done using datasets from Group A in the order presented in the table 5.1 with a budget $B_{max} \in \{25, 50, 75, 100, 150, 200, 300, 500\}$ for each dataset. The process was performed with parameters:

- learning rate = 0.1,
- discount factor = 0.99,
- epsilon = 0.15,
- max height = 8.

Testing, in that case, is done with budged equal to 100 iterations. What is more, to compare learned policies, all actions should be taken according to the current policy, so randomness has to be reduced. Thus epsilon parameter was set to 0. Other parameters were set to the same values as for the learning weights process.

Figure 5.1 shows the exploration efficiency for every dataset from group B depending on the number of datasets from group A used for learning weights.

As we can see, for 6 out of 11 datasets (irish, cholesterol, blood-transfusion-service-center, biomed, eucalyptus, wilt) there are less excessively explored nodes as the number of datasets used for training increases. On average for these datasets, the number of excessively explored nodes decreases by 62% when comparing exploring after learning weights on 25 datasets comparing and exploring after learning weights on 5 datasets. This means that for these datasets, the graph is explored more efficiently when increasing the number of data used for learning weights. Hence the budget can be reduced.

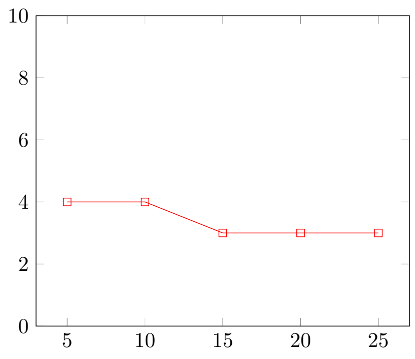
3 out of 11 datasets (car, tick-tac-toe, monks-problems-3) preserves almost not changed efficiency no matter how many datasets are used for training. These three datasets contain only categorical features, and node created after applying one hot encoding (sometimes followed by feature selection) is usually the best one. Even after a short training, only on five datasets, there are required only three nodes of excessive exploration in order to find the optimal node.

However, 2 out of 11 datasets (diabetes, breast-cancer) show the opposite tendency - the number of excessively explored nodes increases as the number of datasets used for learning weights increases. The common part of the best transformation pipelines found for these two datasets is the feature selection transformation selecting 20% of features. The transformation, when more datasets is used for training, loses importance, and is used less frequently. Hence, other transformations are applied at first, causing more excessively explored nodes.

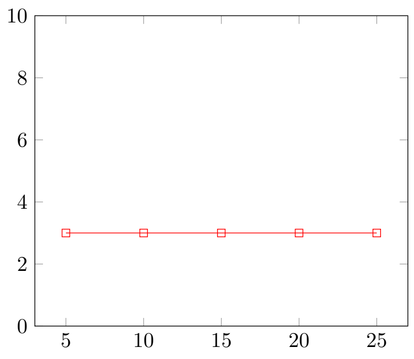
To sum up, plot (l) in figure 5.1 shows an average of excessively explored nodes for all 11 datasets. On average, the optimal node is achieved in 30% less excessive explored nodes after learning weights on 25 datasets comparing to efficiency after learning weights on 5 datasets. The fewer nodes have to be explored, the fewer machine learning models evaluations is needed. It means that computational and time resources are used efficiently. Hence, the maximum number of iterations can be decreased after using more datasets for learning weights. The conclusion is the algorithm using a reinforcement learning technique indeed learns how to perform the feature engineering process. Therefore, knowledge collected during some historical runs can be successfully transferred onto a future feature engineering processes.

5.3. Maximum height influence

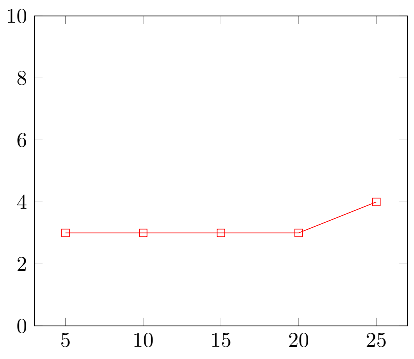
Maximum height is a length of the longest path from a root node to every other node in a transformation graph. In case of a root node, it's height is equal to 1. The deeper the node is, the more complicated the pipeline of transformations should be used in order to create its dataset. Moreover, when on the path, there are no feature selection transformations, every next node has



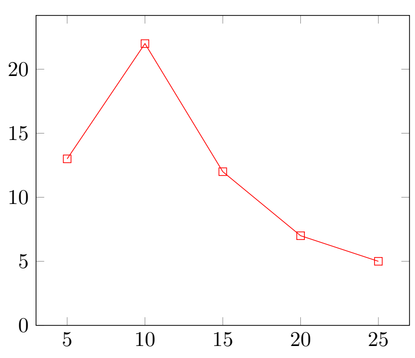
(a) car



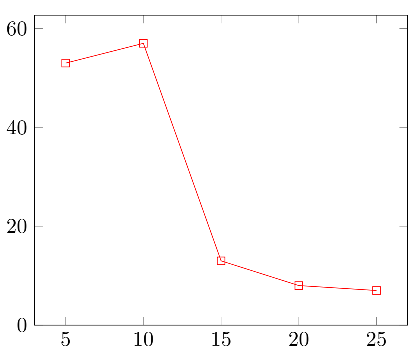
(b) tick-tak-toe



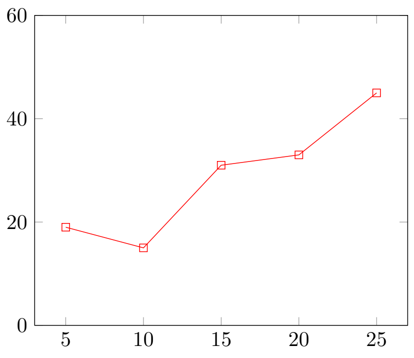
(c) monks-problems-3



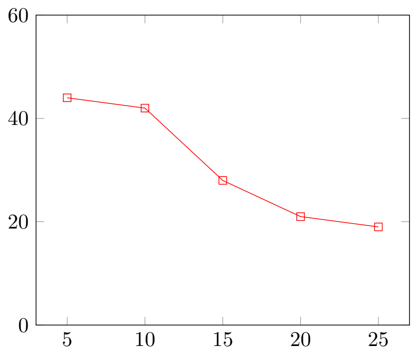
(d) irish



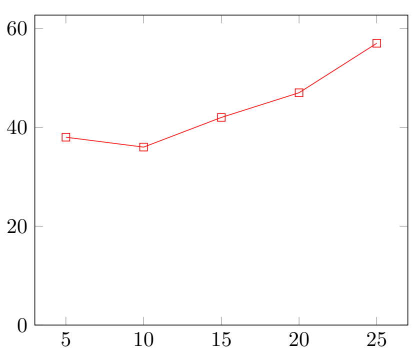
(e) cholesterol



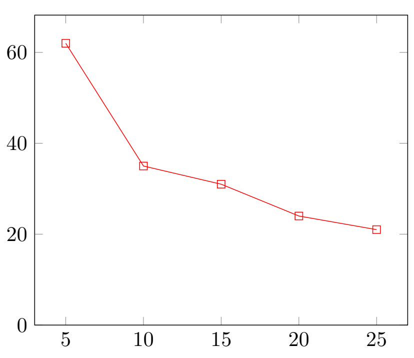
(f) diabetes



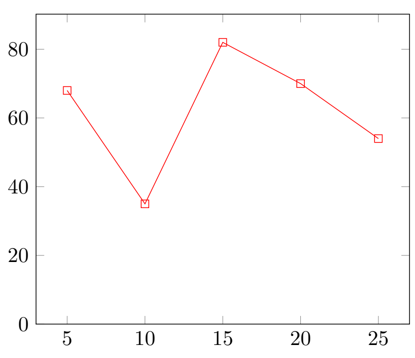
(g) btsc



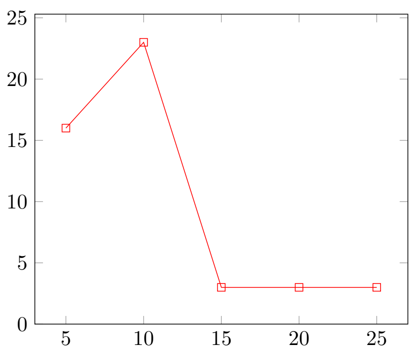
(h) breast-cancer



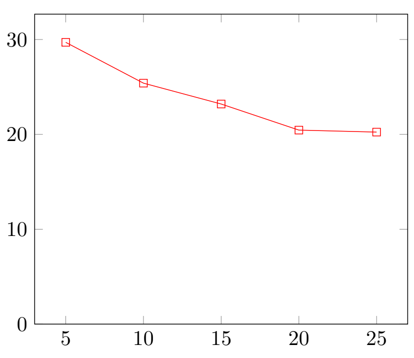
(i) biomed



(j) eucalyptus



(k) wilt



(l) Avg. for all datasets

Figure 5.1: Exploration efficiency - the number of excessively explored nodes (vertical axis), depending on the number of datasets used for training (horizontal axis). Less number of excessively explored nodes means the exploration is performed more efficiently.

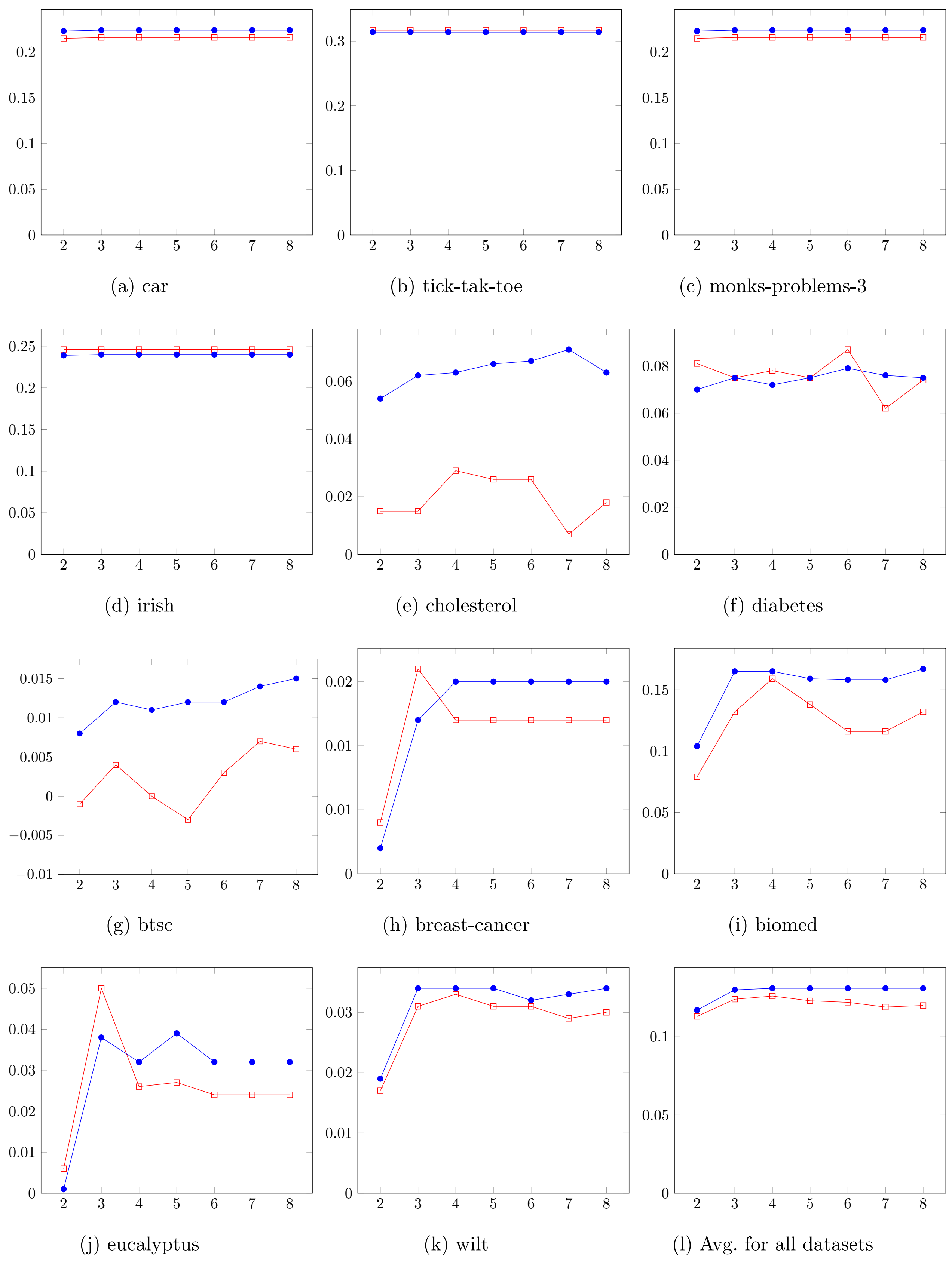
more features than the previous one, which usually slows down the training of the machine learning model. So in order to reduce the complexity of a transformation pipeline and training time of the machine learning model, the maximum height has to be bounded. However, the question is to what extent? The goal of the experiment is to investigate what influence on the score gain (the difference between score after feature engineering and before) has the maximum height parameter. The experiment is performed for maximum height parameter in the range from 2 to 8. Other parameters are the same as for the learning weights process described in section 5.2.

Figure 5.2 shows the average score gains in comparison to score without feature engineering, depending on the maximum height parameter for each of the datasets from group B. The blue plot shows the score gain on training data (data used for building transformation graph). The red plot shows the score gain on testing data.

For datasets that require simple transformation pipelines (car, tick-tac-toe, monks-problems-3, irish), score gain is not dependent on the maximum height parameter. The mentioned datasets require only one hot encoding transformation, sometimes followed by feature selection, in order to significantly improve the dataset score. Applying other transformations does not positively affect the score, so the optimal node is found near the root, and changing the maximum height parameter for higher values does not cause the algorithm to find a different optimal node.

The rest of the datasets (cholesterol, diabetes, blood-transfution-service-center, breast-cancer, biomed, eucalyptus, wilt) require the maximum height parameter set for higher values than 2 in order to find the best transformation pipeline. However, the score gain does not always increase when the maximum height parameter increases - at some point, deterioration can be observed. On average for all datasets (plot (l) in figure 5.2), the optimal value for the maximum height parameter is equal to 4. For higher values, the score gain is lower.

The reason is that increasing the parameter does not always cause lengthening the optimal transformation pipelines. When the parameter is set to lower values, the transformation graph is searched more in breadth than in depth. Higher values allow exploring deeper parts of the graph, causing more nodes that are closer to the root to be unexplored. The optimal node found for the lower values of the parameter can be not visited for the higher values due to the exploration of deeper nodes. The maximum height parameter influences the balance between breadth and depth exploration.



(a) car (b) tick-tak-toe (c) monks-problems-3 (d) irish (e) cholesterol (f) diabetes (g) btsc (h) breast-cancer (i) biomed (j) eucalyptus (k) wilt (l) Avg. for all datasets

Figure 5.2: The maximum height parameter influence - score gain (vertical axis), depending on the maximum height parameter value (horizontal axis). The blue plot is score gain for a training part of data, the red for a testing part of data. More significant score gain means the feature engineering process is performed more effectively.

5.4. Performance after feature engineering

Figure 5.3 is a comparison of F1 scores for eleven datasets without performing feature engineering and after performing feature engineering. The testing process, in that case, was done with the following parameters:

- learning rate = 0.1,
- discount factor = 0.99,
- epsilon = 0.15,
- max height = 5,
- $B_{max} = 50$.

As it is shown in the figure, even for a relatively small budget, the score gain is noticeable. The average improvement is equal to 0.124.

It is worth mentioning that, in the scope of one dataset, the optimal transformation pipeline can differ between splits. Taking three splits of biomed dataset as an example, the transformation pipelines for each split looks as follows:

1. feature selection selecting 75% of features → standardization,
2. quantile binning with encoding bins by a category → standardization → one-hot encoding → feature selection selecting 75% of features → feature selection selecting 75% of features,
3. feature selection selecting 75% of features → square root.

Taking into consideration all splits of all datasets, in optimal transformation pipelines following transformations are used (ordered by usage frequency):

1. one-hot encoding - 21 usages,
2. quantile binning with encoding bins by mean of bins' edges - 18 usages,
3. feature selection selecting 75% of features - 18 usages,
4. feature selection selecting 20% of features - 7 usages,
5. quantile binning with encoding bins by a category - 3 usages,
6. square root - 3 usages,

7. feature selection selecting 50% of features - 2 usages,
8. standardization - 1 usage,
9. logarithm - 1 usage.

The most frequent transformation is one-hot encoding. It usually significantly improves the performance of datasets containing categorical features. Moreover, the transformation can be used in combination with quantile binning with encoding bins by category, so even for datasets initially not containing categorical features, after applying the mentioned type of binning, it is possible to apply one-hot encoding.

Another frequently used transformation is quantile binning with encoding bins by mean of bins' edges. The high number of usages is because the transformation often occurs multiple times in a single transformation pipeline. The result of composing the transformation more than once is not only reducing the diversity of values but also lowering the standard deviation of numeric features.

Also, the feature selection transformations are used frequently. Other transformations add a lot of new features, so it is necessary to prune away not important redundant ones. Thanks to that, the model's performance increases.

On the other side, these transformations are not a part of any optimal transformation pipeline:

- min-max scaling,
- sine,
- cosine,
- hyperbolic tangent,
- sigmoid function,
- all missing value imputing transformations.

What is worth noticing, in any optimal transformation pipeline, there is not used any of the implemented, more complex missing value imputations e.g., feature imputation using regression model or imputation using the KNN algorithm. Simple replacing all missing values by 0 is enough in that case, because datasets used for testing contain a small number of missing values.

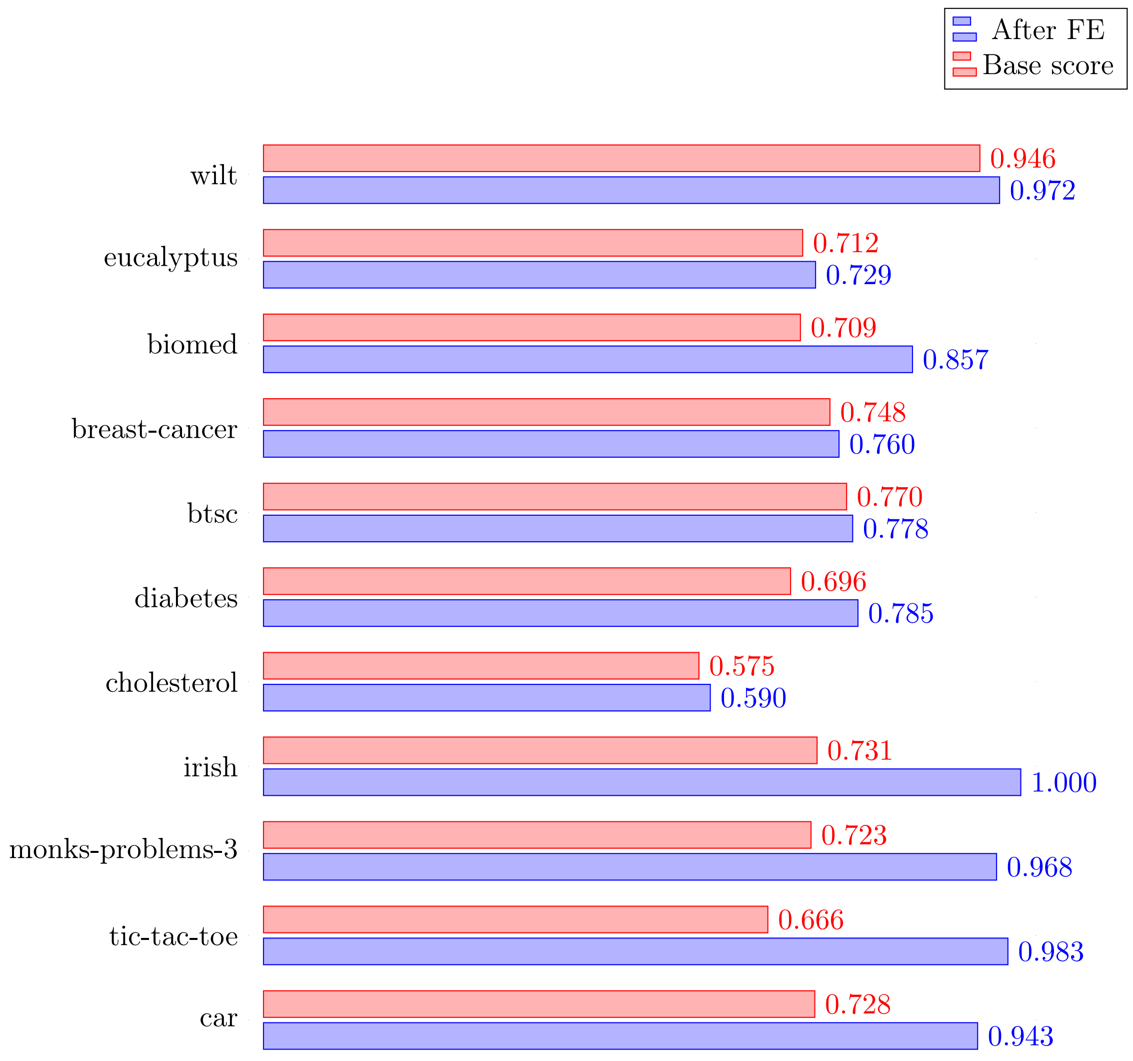


Figure 5.3: Comparison of F1 scores for eleven datasets without performing feature engineering and after performing feature engineering.

Table 5.1: Datasets used for weights learning process

Index	Dataset name	No. rows	No. features	% of missing values	No. num. features	No. cat. features
1	visualizing_environmental	111	3	0%	3	0
2	chscase_geyser1	222	2	0%	2	0
3	heart-h	241	13	24.96%	6	7
4	wdbc	569	30	0%	30	0
5	visualizing_livestock	130	2	0%	0	2
6	breast-w	699	9	0.25%	9	0
7	fri_c3_100_5	100	5	0%	5	0
8	analcatdata_chlamydia	100	3	0%	0	3
9	shuttle-landing-control	15	6	24.76%	0	6
10	colic	368	22	22.69%	7	15
11	transplant	131	3	0%	3	0
12	kc2	522	21	0%	21	0
13	vote	435	16	5.63%	0	16
14	ilpd	583	10	0%	9	1
15	profb	672	9	19.84%	5	4
16	monks-problems-1	556	6	0%	0	6
17	monks-problems-2	601	6	0%	0	6
18	hip	54	7	31.75%	7	0
19	heart-statlog	270	13	0%	13	0
20	credit-approval	690	15	2.35%	6	9
21	heart-c	303	13	0.18%	6	7
22	dresses-sales	500	12	13,92%	1	11
23	banknote-authentication	1372	4	0%	4	0
24	haberman	306	3	0%	2	1
25	cmc	1473	9	0%	2	7

Table 5.2: Datasets used for testing

Index	Dataset name	No. rows	No. features	% of missing values	No. num. features	No. cat. features
1	car	1728	6	0%	0	6
2	tick-tac-toe	958	9	0%	0	9
3	monks-problems-3	554	6	0%	0	6
4	irish	500	5	1.28%	2	3
5	cholesterol	303	13	0.15%	6	7
6	diabetes	768	8	0%	8	0
7	blood-transfusion-service-center (btsc)	748	4	0%	4	0
8	breast-cancer	286	9	0.35%	0	9
9	biomed	209	8	0.9%	7	1
10	eucalyptus	736	19	3.66%	14	5
11	wilt	4839	5	0%	5	0

6. Summary

6.1. Future work

There are a couple of directions for further improvement of the presented methodology and the implemented tool.

First of all, a linear approximation of state and Q-value function was used, which limits finding potential dependencies between state and further actions to some extent. In exchange for that, some non-linear approximation of Q-value function can be used, e.g., neural network. However, in that case, it probably would take more time to train such model (due to the increased complexity of the system).

Moreover, used transformations have their hyperparameters, which can also be tuned. In this thesis, just a fixed set of the hyperparameters was used. Tuning the transformations can potentially improve the feature engineering performance, when it comes to score, but can decrease the performance in terms of computational power and time used for training.

Due to the fact that most of the transformations extend the number of features in the dataset, the feature selection process is an important step in exploring the transformation graph and performing a successful feature engineering process. In this thesis, there was used only a simple type of feature selection transformation. Applying more complex feature selection is an interesting field of research.

The next way of improvement is choosing the best node in a transformation graph. In order to avoid or reduce overfitting, more factors such as the node's depth or the number of features in the corresponding dataset should probably be taken into consideration.

6.2. Conclusions

Feature engineering is a powerful tool when it comes to improving the machine learning model's performance. Thanks to that, data is easier to handle by these models. However, there are a lot of possible variants of transformation pipelines because of the number of possible transformations, the order of applying them, and various transformations' composition combinations. All of these factors make the feature engineering process a hard and time-consuming task.

The reinforcement learning technique can be used to solve many types of problems. The technique is capable of learning the environment in which is acting by trial and error method. It collects the environment feedback of taken actions and makes use of gained experience in similar future tasks.

The results of the thesis show that it is possible to join feature engineering and reinforcement learning. First of all, the feature engineering process has to be modeled as a reinforcement learning problem. Then performing a training process allows the RL technique to learn how feature engineering environment works and which steps take under given circumstances. The more datasets that are used for training, the less time it takes to find a proper transformation pipeline for various datasets that are representing some classification problems.

As a part of the thesis tool which facilitates feature engineering process was implemented. The tool is not only capable of performing automated feature engineering, but it can also be used as a guide showing possible ways of transforming given data. That is because the tool can produce a graphical representation of the feature engineering process - a transformation graph.

To sum up, as the results of the thesis show, the reinforcement learning technique can be successfully applied to feature engineering problems. Thanks to that, historical feature engineering processes can be used in order to improve machine learning pipelines when solving future tasks. The implemented tool can automatize the process of feature engineering, which potentially can save much high-valued time of specialists.

Bibliography

A. Feinberg, E. and Shwartz, A. [2002], *Handbook of Markov Decision Processes: Methods and Applications*, Vol. 40, Kluwer Academic Publishers.

Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M. and Hutter, F. [2015], Efficient and robust automated machine learning, *in* C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama and R. Garnett, eds, 'Advances in Neural Information Processing Systems 28', Curran Associates, Inc., pp. 2962–2970. [Online; accessed 15.12.2019].
URL: *http://papers.nips.cc/paper/5872-efficient-and-robust-automated-machine-learning.pdf*

Gansner, E. R. and North, S. C. [2000], 'An open graph visualization system and its applications to software engineering', *SOFTWARE - PRACTICE AND EXPERIENCE* **30**(11), 1203–1233.

Guo, Y. [2017], 'The 7 steps of machine learning'. [Online; accessed 15.12.2019].
URL: *https://towardsdatascience.com/the-7-steps-of-machine-learning-2877d7e5548e*

Heaton, J. [2017], 'An empirical analysis of feature engineering for predictive modeling', *CoRR* **abs/1701.07852**. [Online; accessed 15.12.2019].
URL: *http://arxiv.org/abs/1701.07852*

Irodova, M. and Sloan, R. H. [2005], Reinforcement learning and function approximation, *in* 'FLAIRS Conference'.

Jones, E., Oliphant, T., Peterson, P. et al. [2001], 'SciPy: Open source scientific tools for Python'. [Online; accessed 15.12.2019].
URL: *http://www.scipy.org/*

Khurana, U., Samulowitz, H. and Turaga, D. S. [2017], 'Feature engineering for predictive modeling using reinforcement learning', *CoRR* **abs/1709.07150**. [Online; accessed 15.12.2019].
URL: *http://arxiv.org/abs/1709.07150*

Kursa, M. B. and Rudnicki, W. R. [2010], 'Feature selection with the Boruta package', *Journal of Statistical Software* **36**(11), 1–13. [Online; accessed 15.12.2019].
URL: *http://www.jstatsoft.org/v36/i11/*

Martinsson, P.-G., Rokhlin, V. and Tygert, M. [2011], ‘A randomized algorithm for the decomposition of matrices’, *Applied and Computational Harmonic Analysis* **30**, 47–68.

McKinney, W. et al. [2010], Data structures for statistical computing in python, *in* ‘Proceedings of the 9th Python in Science Conference’, Vol. 445, Austin, TX, pp. 51–56.

Oliphant, T. [2006], ‘NumPy: A guide to NumPy’, USA: Trelgol Publishing. [Online; accessed 15.12.2019].
URL: *http://www.numpy.org/*

Olson, R. S., Bartley, N., Urbanowicz, R. J. and Moore, J. H. [2016], Evaluation of a tree-based pipeline optimization tool for automating data science, *in* ‘Proceedings of the Genetic and Evolutionary Computation Conference 2016’, GECCO ’16, ACM, New York, NY, USA, pp. 485–492. [Online; accessed 15.12.2019].
URL: *http://doi.acm.org/10.1145/2908812.2908918*

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, E. [2011], ‘Scikit-learn: Machine Learning in Python ’, *Journal of Machine Learning Research* **12**, 2825–2830.

Puterman, M. L. [1994], *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, 1st edn, John Wiley & Sons, Inc., New York, NY, USA.

Shalev-Shwartz, S. and Ben-David, S. [2014], *Understanding Machine Learning: From Theory to Algorithms*, Cambridge University Press, New York, NY, USA.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T. et al. [2018], ‘A general reinforcement learning algorithm that masters chess, shogi, and go through self-play’, *Science* **362**(6419), 1140–1144. [Online; accessed 15.12.2019].
URL: *http://science.sciencemag.org/content/362/6419/1140/tab-pdf*

Somasundaram, R. and Nedunchezhian, R. [2011], ‘Evaluation of three simple imputation methods for enhancing preprocessing of data with missing values’, *International Journal of Computer Applications* **21**.

Sutton, R. S. and Barto, A. G. [2018], *Reinforcement Learning: An Introduction*, second edn, The MIT Press. [Online; accessed 15.12.2019].
URL: *http://incompleteideas.net/book/the-book-2nd.html*

Thornton, C., Hutter, F., Hoos, H. H. and Leyton-Brown, K. [2013], Auto-weka: Combined selection and hyperparameter optimization of classification algorithms, *in* 'Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining', ACM, pp. 847–855. [Online; accessed 15.12.2019].
URL: *https://arxiv.org/pdf/1208.3719*

Van Rossum, G. and Drake Jr, F. L. [1995], *Python tutorial*, Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands.

Vanschoren, J., van Rijn, J. N., Bischl, B. and Torgo, L. [2013], 'Openml: Networked science in machine learning', *SIGKDD Explorations* **15**(2), 49–60. [Online; accessed 15.12.2019].
URL: *http://doi.acm.org/10.1145/2641190.2641198*

Watkins, C. J. C. H. [1989], Learning from Delayed Rewards, PhD thesis, King's College, Cambridge, UK. [Online; accessed 15.12.2019].
URL: *http://www.cs.rhul.ac.uk/$\tilde{c}$hrisw/new$_t$hesis.pdf*

Zheng, A. and Casari, A. [2018], *Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists*, 1st edn, O'Reilly Media, Inc.

List of symbols and abbreviations

AFERL	Automated Feature Engineering using Reinforcement Learning
AutoML	automated machine learning
FE	feature engineering
GPI	generalized policy iteration
KNN	k nearest neighbours
MC	Monte Carlo
MDP	Markov Decision Process
ML	machine learning
PCA	principal component analysis
RL	reinforcement learning
SGD	stochastic gradient descent
TD	temporal difference
TPOT	Tree-based Pipeline Optimization Tool

List of Figures

List of tables