Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Predicting Blizzard score in StarCraft II

David Valouch

Supervisor: Ing. Martin Svatoš
January 2019

Acknowledgements

Děkuji Ing. Martinu Svatošovi (Katedra počítačů fakulty Elektrotechnické, ČVUT v Praze), který mi téma této práce navrhl a nasměroval mne k relevantním zdrojům.

Děkuji Vojtěchu Francovi, Ph.D. (Katedra kybernetiky fakulty Elektrotechnické, ČVUT v Praze). Během krátkého rozhovoru po přednášce předmětu *Statistické strojové učení* mě přivedl ke klíčové myšlence použít regresní stromy v kombinaci s *ensamble* metodami.

Děkuji za konzultaci RNDr. Vladimíru Šverákovi, DrSc. (University of Minnesota – Department of Mathematics).

V neposlední řadě děkuji své rodině za trpělivost a psychickou podporu.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for obseving the ehtical principles in the preparation of university theses.

Prague, 7^{th} January 2019

Abstract

In this thesis we attempt to employ inductive logic programming to extract features numeric from a complex domain represented using first order logic language and to use those features for solving a regression taks within the domain.

The particular domain is the RTS game StarCraft II and the predicted value is the player score.

Keywords: StarCraft, RTS, real time strategy, inductive logic programming, regression

Supervisor: Ing. Martin Svatoš

Abstrakt

V této práci se pokusíme využít metody induktivního logického programování k vytěžení numerických hodnot v komplexní prostředí reprezentované pomocí jazyku prvořádové logiky a následně tyto hodnoty využít k řešení regresní úlohy v tomto prostředí.

Konkrétně se budeme zabývat prostředím hry StarCraft II a budeme předvídat hodnotu hráčkého skóre.

Klíčová slova: StarCraft, RTS, realtimová strategie, induktivní logické programování, regrese

Překlad názvu: Predikování Blizzard skóre ve StarCraft II

Contents

1 Preliminaries **1**

1.1 StarCraft II – Game Description . 1

1.1.1 RTS Game Layers 2

1.2 Introduction to Logic 3

1.3 Clausal Logic Syntax 3

1.4 Semantics of Clausal Logic 4

1.5 Inductive Logic Programming: Logical and Relational Learning . . . 5

1.6 Subsumption 7

1.7 Efficiency 8

1.8 Feature extraction from First Order Logic Data Using Techniques of Inductive Logic Programming . . . 9

1.8.1 Conjuctive Query Mining 9

1.8.2 Feature Evaluation 9

1.9 Decision Trees 10

1.10 Ensamble Methods 10

1.10.1 Adaboost 11

1.11 R^2 Metric 11

2 Task Description **13**

2.1 Task Definition 13

2.2 Jusctification for Predicting Player Score/Advantage 14

3 Related Work **17**

3.1 Bots 17

3.2 Simmilar and Closely Related Work 18

4 Proposed Solution **19**

4.1 Data Preprocessing 19

4.2 Feature Mining from First Order Logic Representation 19

4.2.1 TreeLiker 20

4.3 Regression on Attibute Value Data 21

5 Execution **23**

5.1 Data Preprocessing 23

5.2 ILP Feature Mining and First Order Logic Representation 24

5.2.1 Representation 25

5.2.2 Feature Mining 26

5.3 Regression on Attribute Value Data 26

5.3.1 Data Preparation 27

5.3.2 Evaluation 27

6 Conclusion **29**

6.1 Experiment Results 29

6.2 Future Work 29

A Bibliography **31**

B Attachements and Code Documentation **33**

B.1 DVD Contents 34

B.2 Code Documentation 35

B.3 /DataProcessing 35

B.3.1 /ReplayParser 35

B.3.2 /DataExtraction 37

B.4 /AttributeValueLearning 38

C Project Specification **39**

Figures

Tables

5.1 Adaboost and decision tree results 28

Chapter 1

Preliminaries

In this thesis I expect that the reader is familiar with concepts from the fields of formal logic and machine learning. Not all concepts are, therefore, explained in-depth.

1.1 StarCraft II – Game Description

StarCraft II is a real time strategy game (RTS) released in 2010. It is a sequel to StarCraft that came out in 1998 and established the RTS genre. In RTS games players control units in a virtual environment, refered to as a *map*. Unlike in turn based games players can take actions at any time (*in real time*). Individual units fulfill various roles, such as collecting resources (minerals and Vespene gas in StarCraft/StarCraft II), building structures and engaging enemy units in combat. Players have imperfect information about the game state; each unit reveals only a portion of the map in its immediate vicinity, the rest of the map is covered in *fog of war*. Players must actively scout the environment to see what the enemy is doing.

In StarCraft (II) There are three races (*Terrans*, *Protoss*, *Zerg*) each with unique set of units with various abilities to interact with the game world.

The game allows multiple teams of up to 4 people to face each other. However, the most popular game mode is 1v1. I will focus on this type of matches as they the most common and are the simplest form of the game.

There is a simple rule-based AI in the game for the purposes of the single player mode and for new players to play against to learn the basics of the game before playing agains human opponents. It has multiple difficulty levels. At the highest difficulty levels it is cheating by gaining extra resources and access to otherwise hidden information. It is easy, however, for an experienced human player to beat it.

To the concern of reinforced learning there are two reward structures. The game ending condition win/tie/loss and Blizzard score. The Blizzard score is computed in every game frame from values such as the amount of resources, the size of one's army and value of destroyed enemy units or structures. While there is no guarantee the player with higher Blizzard score will win it is the only readily available indicator as to how well the player is doing that is accessible at any time during the game.

1.1.1 RTS Game Layers

RTS games can be broken down into multiple layers depending on the time and spacial scope of the decisions making and actions taken, as recognized widely by players and researchers alike [Syn12].

- **Micromanagement (ultra-short term)** [Syn12, 5] This layer considers the time span of several seconds. It encompasses actions such a control of individual units. While most of it are just routine actions, some level of skill is required to perform well in this scope.

  The human player uses the standard computer interface (mouse and keyboard) to control the game. In order to perform actions quickly and precisely one must acquire certain degree of manual dexterity. AI agents interact with the game via its API.

  Some higher level concepts that fall into this area are actions such as moving units, assigning attack targets to units and/or using special abilities some units posses. One example of a micromanagement layer technique is *kiting* – a technique that has one keeping their mobile units with higher attack range from enemy immobile short ranged units and applying damage by rapidly switching between retreating and attacking. In some cases this technique allows a small group of relatively weak units to dispatch a larger group of more valuable units.

- **Tactics (short term)** [Syn12, 6] Tactics is what takes place on a time scale of under a minute. Considering larger groups of units and larger sections of the map*. Actions such as repositioning one's army or attacking the enemy fall into this category.

  This is the most difficult part of the game. It requires fast evaluation of risks, incomplete information decision making and prediction of opponent's actions.

  I would personally include more long-term decisions such unit production and base expansion (building new resource gathering infrastructure in different parts of the map than one's starting location) into tactics although they are more related to strategy.

- **Strategy (long term and pre-game planning)** [Syn12, 7] Strategy is the specific plan the player has for winning the game. While some

tactical adjustments to one's strategy have to be made during the game, a player usually commits to a certain strategy before the game even begins. Strategy and tactics can be more broadly referred to as *macro-management*.

One example of a common strategy is *rushing*. As the name suggest this is when a player attempts to win the game as soon as possible by quickly generating an army of basic game units and attacking the enemy before they are able to defend themselves. This is often effective with lover level players as they often lack the mechanical skill to set up defenses quickly enough.

Abstract actions on higher level can be deconstructed and solved as individual problems on lower levels. For example a strategy-level decisio to attack an opponent is broken down into tactics-level problems, which can be solved by tactics-level moves such as moving a group of units from location A to location B, which can be further broken down into the individual micro-management actions.

1.2 Introduction to Logic

Predicate logic also refered to as first order logic (FOL) is a strong and elegant tool for knowledge representation. It can model relational databases, graphs and other data structures and is mathematically well-understood. It is therefore natural that one may want to explore its possible usages in the area of machine learning – where logical representation may offer better expressiveness than the feature vector representation used in most common machine learning paradigms.

In logical programming it is common to use a specialized type of predicate logic, a clausal logic.

1.3 Clausal Logic Syntax

This section follows [dR08, 2.2].

A *term* t is a constant, a variable or a compound term $f(t_1, \ldots, t_n)$, where f/n is a function symbol of arity n and $t_1, \ldots, t_2$ are terms. Compound terms can be used to represent structured data. Such data are also known as functors. Terms not containing variables are called *ground terms*.

An *atom* is a formula of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol and $t_1, \ldots, t_n$ are terms. An atom containing only ground terms is a *ground atom*.

A *clause* is an expression of the form $h_1; \ldots; h_n \leftarrow b_1, \ldots, b_m$. h_i and b_j are atoms. $h_1; \ldots; h_n$ is the head of the clause and is a disjunction (or) of it's literals. $b_1, \ldots, b_m$ is the body of the clause and is a conjunction (and). All variables in a clause are universally quantified. (In the syntax of the declarative language Prolog the $\leftarrow$ symbol is replaced by :-.)

There are some special types of clauses:

- *facts* : $n = 1$, $m = 0$ and does not contain variables
- *Horn clauses* : $n \in \{1, 0\}$
- *definite clause* : $n = 1$
- *denial/query* : $n = 0$
- *empty cluase,* $\square$: $n = m = 0$

facts and *definite clauses* are the most common in logical programming. Facts often represent claims about objects (constants) in the described world. Definite clauses generally represent rules by which other facts can be derived – the head represents a tested property and the body describes criteria that the terms in the head must satisfy to have that property. For example `mother(X) :- parent(X,Y), female(X)` represents a rule that `X` is a mother if she is female and if she is parent of some `Y`.

A *database/logical program* is a set of clauses.

A *substitution* $\theta = V_1/t_1, \ldots, V_n/t_n$ is an assignment of terms $t_1, \ldots, t_n$ to variables $V_1, \ldots, V_n$. If all t_i are constants θ is a ground substitution.

1.4 Semantics of Clausal Logic

This section follows [dR08, 2.3].

Semantics is centered around the conceps of *interpretation* and a *domain*. Domain defines a set of objects. *Interpretation* can be understood as a mapping from

- constants to objects in the domain.
- function symbols f/n to n-ary functions mapping n-tuples of objects onto an objec in the domain.
- predicates p/n on n-relations defined on the domain.

In clausal logic we use *Herbrand interpretations*.

A *Herbrand domain* is a set of all ground terms that can be constructed using the given constants and functional symbols.

A *Herbrand interpretation* is a set of ground atoms. All atoms in the interpretation are considered to be true; the others are false. (This is referred to as an closed world hypothesis; anything that is not known or provable is considered to be False.)

A *Herbrand interpretation* I is a *Herbrand model* for a set of clauses C *iff* for all $h_1; \ldots; h_n \leftarrow b_1, \ldots, b_m \in C$ and for all *ground substitutions* θ, $\{b_1\theta, \ldots, b_m\theta\} \subseteq I \rightarrow \{h_1\theta, \ldots, h_n\theta\} \cap I \neq \emptyset$.

A set of clauses C logically entails a clause c ($C \models c$) *iff* all models of C are also models of c.

1.5 Inductive Logic Programming: Logical and Relational Learning

Inductive Logic Programming (ILP) [Mug91][dR08][dR][Plo71] is a machine learning paradigm built around logical programming. Logical programming is a basis for some (even general-purpose) languages; Prolog [End16] being the prime example. Those languages are part of the declarative language family. They generally use a simple proving algorithm akin to resolution method to evaluate queries to a database of facts represented as logical atoms using set of rules represented as Horn clauses. ILP brings a similar approach to the field of machine learning. Mainly the expressiveness of the first order logic (FOL); while most machine learning paradigms represent data as tensors. Since the FOL representation can capture complex relations between object, similarly to a relational database, the relational learning is often used in this context [dR08].

Systems like Prolog, use deductive reasoning to find statements (observations) that are are logical consequence of context given by a program – database of facts and set of rules (background/domain knowledge). ILP attempts to 'reverse engineer' the program by producing facts and/or rules (hypotheses) such that a set of observations follows from it. It is used to find relations and features in data. Background knowledge can be provided by experts in a given field or that itself can be learned by an algorithm, to help guide the learning process. ILP was successfully used in areas of machine learning such as natural language processing and data mining. [dR08] Attempts at using it for regression [KB97] are not so common.

Definition 1.1 (Generality Relation)**.** [Mug91] We define a generality relation in the following manner. We say that formula A is more general than formula B (B is more specific than A) if and only if $A \models B$ and $B \not\models A$. We say that

formula A is more general than formula B relative to formula C if and only if $A \wedge C \models B$ and $B \wedge C \not\models A$.

Now let us describe a general ILP setting as defined by Muggleton [Mug91]

$\mathcal{L}_O$: the language of observations
$\mathcal{L}_B$: the language of background knowledge
$\mathcal{L}_H$: the language of hypotheses

The general ILP task is, given $O \subseteq \mathcal{L}_O$ and $V \subseteq \mathcal{L}_B$, to find $H \in \mathcal{L}_H$, such that

$$B, H \models O \tag{1.1}$$

$\mathcal{L}_O$ is generally required to contain ground literals only (*facts*). And B often consists of Horn clauses (*rules*) It is important to note that it does not necessarily hold that $B, O \models H$ – the difference from deduction. There might be potentially infinite number of hypotheses satisfying 1.1, thus restrictions are usually imposed on H. Often we want H to be only a single clause. We also may require that H is the most/least general formula relative to the background knowledge B. Even then the potential number of hypotheses satisfying the relation 1.1 can be infinite; we may want to impose other non-logical constraints. For example limiting the number of literals in H.

In context of Logical and Relational Learning [dR08]. The observation O corresponds to our data represented as a set of examples. An example E is a FOL clause.

We want to classify the examples as positive/negative by whether a certain property $p(\dots)$ is true in a given example, where $p(\dots)$ is a predicate symbol with some arbitrary arity. An example E is positive if

$$p(\dots) \leftarrow \mathcal{E} \text{ is true within in } O \tag{1.2}$$

Considering the examples as logical rules for the property p, we are looking for hypotheses h as for body of a rule $p(\dots) \leftarrow h$. We say that a hypothesis h covers an example E if h can be satisfied in E ie. $p(\dots) \leftarrow h \models p(\dots) \leftarrow E$. (We represent the example/hypothesis by its body; p is the 'labeling' of the datapoints. Note that in the general setting the hypothesis is the whole rule $p(\dots) \leftarrow h$) If rule t_1 is more general than rule r_2 (according to def. 1.1) then r_1 covers all examples covered by r_2.

The search for hypotheses may be subject to different specifications. We may wish to find all hypotheses that cover at least some minimal percentage of

the positive examples and at most some maximal percentage of the negative examples. Alternatively we may wish to find a hypothesis that is the best classifier of the positive/negative examples. Or hypothesis that 'compresses' the positive examples by covering as much of them as possible while having a short representation. In order to make the search more efficient the hypotheses space is usually structured according to generality. And the search is conducted from more general to specific. Since the hypotheses generally have the form of clauses we specify the hypothesis by literals to the general ones. In this way we are able to prune the search; since creating more specific hypotheses will not cover more positive examples and generalizing hypotheses will not produce rules covering less negative examples.

1.6 Subsumption

In a certain context we may look at logical clauses as on sets of literals.[1]

Definition 1.2 (Propositional Subsumption)**.** [dR08, 5.2] We say about two propositional clauses c_1 and c_2, that

$$c_1 \text{ subsumes } c_2 \textit{ iff } c_1 \subseteq c_2 \tag{1.3}$$

The subsumtion is sound – when c_1 subsumes c_2 then $c_1 \models c_2$, and complete – when $c_1 \models c_2$ then c_1 subsumes c_2.

inpredicate logic we generalize the concept as θ-subsumption.

Definition 1.3 (θ-subsumtion)**.** [dR08, 5.4] Consider clauses c_1 and c_2. We say that

$$c_1 \ \theta\text{-subsumes } c_2 \textit{ iff } \text{ there exists a substitution } \theta \text{ such that } c_1\theta \subseteq c_2\,. \tag{1.4}$$

In this case se write

$$c_1 \preceq_\theta c_2\,. \tag{1.5}$$

The definition of θ-subsumption is sound – when c_1 θ-subsumes c_2 then $c_1 \models c_2$.

The *completeness* property does not hold, however, it only does not hold for 'self-recursive' clauses, ie. clauses that resolve with themselves. This can be caused by function terms.

θ-subsumption forms quasi-ordering of clauses; it is reflexive and transitive; it is not, however, anti-symmetric. In order to make it into an ordering we need to define equivalence classes of clauses.

[1] This is in contrast to usual interpretation of sets of logical expressions (*sentences*). Usually sets are intepreted as conjunctions.

$$c_1 \equiv_\theta c_2 \textit{ iff } c_1 \preceq_\theta c_2 \text{ and } c_2 \preceq_\theta c_1 \tag{1.6}$$

There is a minimal (shortest) clause in each equivalence class. We may obtain a reduction of a clause by simply removing its literals while the reduced class is still subsumed by the original clause ie. $c \preceq_\theta c - \{l\}$.

Subsumption is at heart of logical and relational learning. It embodies the generality relation – c_1 is more general than c_2 if c_1 subsumes c_2, and it is tied to coverage – a hypothesis covers[2] an example if it is subsumed by that example.

In a FOL language without functional symbols the θ-subsumption ordering $\prec_\theta$ – $c_1 \prec_\theta c_2$ *iff* $c_1 \preceq_\theta c_2$ and $c_2 \npreceq_\theta c_1$ corresponds to generality relation and coverage corresponds to $h \preceq_\theta \mathcal{E}$. It can also be used to prun the hypotheses search space, as hypotheses that are equivalent under the subsumption relation are logically equivalent.

1.7 Efficiency

θ-subsumption and coverage testing is as NP-complete problem. There are, however, ways to solve many specific cases quite quickly.

One way to optimise the search for hypotheses is to limit the number of times when subsumption/coverage is computed. For example by keeping a list of hypotheses covering given examples – indeed a specification of a given hypothesis will not cover any more examples than are covered by the the more general original hypothesis.

Also the fact that we separate the examples is a form of optimization. Note that in the general setting for induction we do not talk about examples; rather there is a whole set of observations; examples would be represented as constant arguments added to predicates.

Finally there are more efficient ways to implement subsumption than naive backtracking. Sumbsumtion decision problem is essentially asking a query in a prolog-like database. We can rearrange the literals in the query in a way that the variables are grounded faster of we can split the query into multiple smaller subqueries that can be solved independently.

[2] Only in certain cases – coverage of example ($\mathcal{E}$) by hypothesis h means $p(\ldots) \leftarrow h \models p(\ldots) \leftarrow \mathcal{E}$; the *completeness* property does not hold for all cases

1.8 Feature extraction from First Order Logic Data Using Techniques of Inductive Logic Programming

The following techniques are implemented in the TreeLiker (see. sec. 4.2.1) program refered to me by my supervisor.

1.8.1 Conjuctive Query Mining

Counjunctive Query Mining is an ILP method. The algorithm searches the hypotheses space; the hypotheses have the form of ungrounded headless Horn clauses (queries). The search tree depth corresponds to the number of literal in the clause. New literals in a child clause are connected through common terms, meaning that a newly added term shares a variable with some literal in the parent clause.

Features are being tested agains a dataset of examples classified as either positive of negative. Hypotheses are being selected based on how many positive and negative examples they cover. The criterion for selection and potentially prunning the search tree depends on the specific task. (see. sec. 1.5) The code accepts a contructs called language bias. It is an another layers of filtering where hypothses are selected based on their structure. This allows the user to define a concept of what a sensible hypothesis may look like and thus further prunning the search space.

1.8.2 Feature Evaluation

Feature evaluation is a processs by which numeric values corresponding to each hypoth- esis is produced for each example. The basis evaluating the hypothesis as a query over a given example and counting how many times the hypothesis can be grounded in the example. This ca be done to varying degrees of accuracy. For example: existential – binary value; either the feature can be grounded in an example of it cannot, counting – all possible groundings are counted. Feature values may also be assigned not just on existence or number of substitutions by which they subsume the example. Numeric values assigned via the substitution can also be used to assign value (the Poly algorithm in TreeLiker).

1.9 Decision Trees

This section follows [cita, 1.10.7].

Let us have training data $\mathbf{x}_i \in \mathbb{R}^n$ and their corresponding labels $y_i \in \mathbb{R}$.

Let data at node m be denoted as Q. A split θ is a pair (j, t_m), where j is an index of a field in $\mathbf{x}$ and $t_m \in \mathbb{R}$ is a threshold.

θ defines partitions $Q_{\text{left}}(\theta)$ and $Q_{\text{ringt}}(\theta)$ such that:

$$Q_{\text{left}}(\theta) = (\mathbf{x}, y)|x_j \leq t_m$$
$$Q_{\text{right}}(\theta) = (\mathbf{x}, y)|x_j > t_m$$

Let us have an *impurity function* H. We use it to define G.

$$G(Q, \theta) = \frac{|Q_{\text{left}}(\theta)|}{|Q_m|} H(Q_{\text{left}}(\theta)) + \frac{|Q_{\text{right}}(\theta)|}{|Q_m|} H(Q_{\text{right}}(\theta))$$

We select θ^*

$$\theta^* = \text{argmin}_\theta G(Q, \theta) \tag{1.7}$$

Recurse for $Q_{\text{left}}(\theta^*)$ and $Q_{\text{left}}(\theta^*)$ until maximum depth is reached or until $|Q_m| <$ minimum number of samples.

1.10 Ensamble Methods

This section takes from [citb].

Ensamble methods create several base estimators of a given type and combine them in a single estimator using, usually, a weighted sum. The goal of *ensembling* is to improve generalizabilty.

They are great for combination with decision trees since those are vulnerable to overfitting – in order to produce 'smooth' prediction one needs a complex decision tree, however, complex decision trees can potentially 'remember' the whole training dataset.

There are two general types of ensamble learning:

- **Averaging** – the individual extimators are good by themselves. Used to improve robustness.
- **Boosting** – the individual classifiers are *weak*. They combine to produce a *strong estimator*.

In order for the different estimators not to be all the same, we need to use different data for training each one. This can is generally done in one of the following ways:

- giving data different weights
- randomly splitting the data (special case of the above – weights are from $\{0, 1\}$)

1.10.1 Adaboost

[citb, 1.11.3]

Adaboost algorithm was introduced in [FS96]. It is a *boosting* ensamble. Training data are assigned weights $w_i : i \in 1 \dots N$ such that $\sum_{i=1}^{N} w_i = 1$; initially $w_i = \frac{1}{N}$.

After adding a new estimator to the ensamble the data are reweighted, such that the most incorrectly evaluated data have higher weight in the next iteration.

1.11 R^2 Metric

R^2 metric, also known as *coefficient of determination*, is is used to represent a stength of an estimator. It is defined as:

$$R^2 = 1 - \frac{mean\ square\ error\ of\ estimator}{variance\ of\ data}$$
$$R^2 \in (-\infty, 1]$$

Value 1 corresponds to perfect estimator, value 0 is value of constant estimator outputing the expected value. Othewise the estimator can be arbitrarily worse.

A 'working' estimator should have a value betwenn 0 and 1.

Chapter 2

Task Description

I am going to attempt to apply ILP and test its results in a not so obviously suited environment. Specifically in the real time strategy game StarCraft II.

The goal is to predict score of players or change in score from information about the game state expressed via the language of first order logic. Alternatively, rather than focusing on the score of a specific player, I believe it is better to focus on difference in players' score (*score advantage*).

Me and my supervisor have chosen StarCraft II in particular as the environment because there is a publicly available dataset of anonymised replays (recorded matches) of human players as a part of newly released deep learning environment StarCraft II Learning Environment by Deep Mind. [VEB+17] I also have personal experience with the game, which gives me some level of 'expert' domain knowledge. StarCraft II is also a populat AI research platform. There is a large community of people building AI bots and competitions are held.

If the ILP assisted approach proves to have some predictive power, it could motivate further work. Features found in game data could potentially be used to guide actions of autonomous agents.

To relax the task I will not deal with hidden information and include observation gathered from both players' perspectives in my data.

2.1 Task Definition

Let us have a training set $\mathcal{T} = \left\{\left(\mathcal{E}_{i,j}^{(o)}, s_{i,j}^{(p)}\right)\right\}$ where $\mathcal{E}_{i,j}^{(o)}$ is a set of *facts* ('example'), true statements that describe the knowledge about state of game i up to frame j containing only information accessible to observer o. The observer can be one of the players or it can be an 'all knowing' spectator. $s_{i,j}^{(p)}$ is a natural number, it is the value of the Blizzard score of player p.

Our goal is to use $\mathcal{T}$ to train a predictive model, such that we are able, given a single observation $\mathcal{E}^{(o_0)}_{i_0,j}$ and optionally the current score $s^{p_0}_{i_0,j}$ to predict the value $s^{p_0}_{i_0,j+d}$, where d is the prediction degree. Since the value of the score itself is not that good indicator of player position, we may wish to predict on the difference of the score. (see. sec. 2.2)

In addition to $\mathcal{E}^{(o)}_{i,j}$ it is possible to also consider real-valued vector $\mathbf{v}^{(o)}_{i,j}$ containing values like amount of players' resources, size of players' army, etc.

In our case we focus on games with two players p_1 and p_2. There fore we have observations of the form $\left(\mathcal{E}^{(p_1)}_{i,j}, \mathcal{E}^{(p_2)}_{i,j}, s^{(p_1)}_{i,j}, s^{(p_2)}_{i,j}\right)$.

In order to not have to deal with uncertainty we will consider examples $\mathcal{E}_{i,j} = \mathcal{E}^{(p_1)}_{i,j} \cup \mathcal{E}^{(p_2)}_{i,j}$

Furthermore let us focus on p_1 as a 'studied' player and p_2 as 'enemy' player. Let $p_0 = p_1$ and $p_e = p_2$. Focusing only on p_0's score is not that helpful in my opinion (see. sec. 2.2) – score can change even when players do not interact, for example through unit production and resource mining, and we may need to lower our score (sacrificing units) to eventually defeat the enemy. It is better, I think, to focus on the 'relative advantage of the studied player'; let us define it as $a^{(p_0)}_{i,j} = s^{(p_0)}_{i,j} - s^{(p_e)}_{i,j}$

This leaves us with a working data of a form $\left(a^{(p_0)}_{i,j+d} | \mathcal{E}_{i,j}, \mathbf{v}^{(p_0)}_{i,j}, \mathbf{v}^{(p_e)}_{i,j}\right)$

2.2 Jusctification for Predicting Player Score/Advantage

In his thesis on AI bot architecture for RTS games G. Synnaeve [Syn12] notes that the most complex layer of the game is the tactics (see. sec. 1.1.1); it is in this one where it is the easiest to tell apart human and non-human player. This is mostly because he has to deal with incomplete information and make predictions about the opponent's actions; the speed of decision making is crucial to.

The goal is to decide how to position (attacking is mostly [1] equivalent with positioning one's own units next to enemy units) units and which units. First order logic representation of the game state is well suited for approximating the game-state from this perspective.

As stated above I don't deal with hidden information. Predicting positions of enemy units is a separate (and non-trivial) task. Therefore I assume that this information was already somehow obtained/predicted.

[1] There are exceptions such as spying on the enemy with invisible units

If the representation describes units present in the game and their relative position. The mined features could represent some 'structures' in unit postions. Most simple such feature would be distace between units of a certain type belonging to oposing players. It may, for example, be advantageous to positions one's units that fight with ranged weapons far from enemy units that fight up close.

The advantage of FOL representation is that the features can be interpretable. If we had a predictor of *score advantage* that works, it would be possible to identify features of the game state that lead to increase/decrease in our score advantage and adjust our strategy such that these features are more/less likely to occur in the game.

Chapter 3

Related Work

3.1 Bots

Although StarCraft II is primarily designed for human players, it and its predecessor, StarCraft, have been used as a platform for AI development. The game comes with a rudimentary AI that is there mostly for the purposes of the single player campaign and to fill a role of a 'dummy player' to fill in for humans. Others have tried more sophisticated approaches to the creation of the so called bots. These programs range from simple agents running on a set of hand-crafted rules, similarly to the built-in AI. Others use more sophisticated approaches of artificial intelligence and machine learning.

In his Phd. thesis, G. Synnaeve [Syn12] describes a possible approaches for creating autonomous game agents for RTS games. He uses the breakdown of the game dectribed in section 1.1.1 . Synnaeve solves each layer as a separate problem.

In his approach to micromanagement each unit is controlled individually by an entity called 'Bayesian unit'. The higher level goals are presented as sensory inputs to these units. Each unit tries to achieve these goal individually and uses Bayesian reasoning to cope with the incomplete information. The units are aware of each other, however, the more complex group actions are an emergent property.

The tactics in this approach boil down mostly to different kinds of attacks (ground, air, ...). The map is represented by a graph where the vertices correspond to different areas of the map and edges are the chokepoints connecting these areas. Observations from units are then used to evaluate the vulnerability of the different areas to certain types of attacks.

On the strategy layer the planning is largely about how to spend resources. Between three main classes of goals – expansion of base and economy, army production, and technology. And also what army composition to produce

according to units the opponent is likely to produce.

A similar approach is used in [YSA+12]. There the authors only distinguish between macro and micromanagement. On the macromanagement layer high level goal are selected and they are executed by mostly 'autonomous' units on the micro level.

In 2017 a pair of researchers at the University of Copenhagen [JR17] used deep learning predict actions of skilled human players based on data from replays. Then they integrated the trained neural network into an existing bot in place of the action planner. Although the performance of the bot was not greatly increased it no longer depended on hard coded rules.

3.2 Simmilar and Closely Related Work

In 2017 a pair of researchers at the University of Copenhagen [JR17] used deep learning predict actions of skilled human players based on data from replays. Then they integrated the trained neural network into an existing bot in place of the action planner. Although the performance of the bot was not greatly increased it no longer depended on hard coded rules.

The joint paper [VEB+17] accompanying the learning environment, from which I will take my data, also mentions the topic of predicting the outcome of the game from the current game state using deep neural networks. They note it is a challenging task even for experienced observers such as people commenting professional matches even when they do not deal with partial observability. It may be interesting to compare the accuracy of their prediction with the accuracy of prediction made solely on the score advantage.

In 1997 pair of Slovenian researches came with an approach to adapt ILP for the purposes of regression on numerical values. They published their results in 1997 [KB97]. There had been attemts to modify ILP to handle numerical data, however, their main contribution is that their method does not require negative examples unlike most other ILP approaches. Their algorithm can also deal with time series.

While searching for a way to extract data from the replay files I found a dataset of processed replays for the original StarCraft game. [LGKS17] I planned to used the same standard of recording every third frame of the game, however, that proved time consuming and unnecessary; also on the second thought I was not right to assume that the old StarCraft game would use the same frame frequency.

Chapter 4

Proposed Solution

In this chapter I present the ouline of a solution to taks defined in chapter 2.

4.1 Data Preprocessing

I am using data from dataset released by Deepmind-Blizzard colaboration [VEB+17]. It contains 64,396 `.SC2Replay` files. The main technical difficulty is that the replay files do not allow for direct reading of the game state in a given frame. The file is a proprietary archive format that contains a log of inputs for the game engine. In order to make observation in the game, one must open the replay in the engine, step through the game-states and access the observations through the game's API. This is impractical for repeatedly reading from the same file.

The first task will, therefore, be to run the replay files in the game engine and record game observation data, ideally using an established storage format not prematurely introducing a custom data representation. The selected representaion ought to be easy to work with.

4.2 Feature Mining from First Order Logic Representation

My goal is to attempt to use ILP for solving the given task. Therefore, I need to represent the data using first order logic language. There are lots of possible ways to represent the data, differing in their interpretation and their complexity.

For the purposes of this work the representation should not be overly complex. One problem that will need to be overcome is inclusion of numeric data – like

distance between units.

A step directly tied to feature mining is feature evaluation (see. sec. 1.8.2). Numeric value are assigned to the features in each example. Producing attribute value vector representation.

4.2.1 TreeLiker

For feature mining and evaluation I am going to use algorithms in *TreeLiker* library[citc] (HiFi[KZ08], Poly[KSHZ11] and RelF[KZ10]).

TreeLiker constructs features based on a provided template.

Definition 4.1 (Template)**.** A template τ is a pair (γ, μ), where γ is a ground function-free query and μ is a subset of arguemts of literals in γ. Arguments in μ are *input arguments*; arguemnts not in μ are *output arguemnts*. Let $C \preceq_\theta \gamma$. An occurence of a variable in C is called *input/output occurence* (w.r.t τ and θ) if corresponds to an input/output arguemnt in γ. [KZ08]

Definition 4.2 (Feature)**.** Let $\tau = (\gamma, \mu)$ and $n \in \mathbb{N}$. A query f (containing no constants and functions) is a *correct feature* w.r.t. τ and n if $|f| \leq n$ and $f \preceq_\theta \gamma$ and for every input occurence of a variable in f there is an output occurence.[KZ08]

When defining templates for TreeLiker we use a Prolog like syntax. Eg.

```
hasCar(-c), hasLoad(+c,-l), box(+l), triangle(+l)
```

Arguments are 'typed' – aguemts with the same name may map to the same variable. [citc]

The symbols before the argument signify the argument *mode*. By *mode* we mean *input*(+)/*output*(-) as in definition 4.1. But in addition to those TreeLiker supports several other modes:

- **constant(#)** allows for the feature to contain constants (exception from definition 4.2)
- **ignored(!)** is neither input nor output.
- **aggregation(*)** simmilar to ignored arguments; but signifies that argument contains numeric data which is used by algorithm Poly [KSHZ11].
- **global constant(@)** does not influence feature construction. If template includes unary predicate `<name>(@value)`, the algorithm required such literal to exist in every example from input data. An texttt @attribute of the same `<name>` will then be included in the `.arff` output file.

Algorithms HiFi[KZ08] and RelF[KZ10] function as the ILP feature mining and feature evaluation described in section 1.8.

Poly[KSHZ11] function in a slightly differently in how it assigns numeric values to the features. It does not assign value based on existence ($\{0, 1\}$) of a feature in an example or on the number of groundings of a feature in an example. Poly produces a numeric value b as a function of value mapped to aggregation argumets in the feature. The function is a n-th degree multivariable polynomial. The polynomial is chosen such that the distribution of the feature value is close to normal distribution. In the *grounding counting* mode Poly takes average over all possible grounding of the feature. *Note that a feature cannot be assigned a value if it has no grounding in an example.*

TreeLiker outputs to a `.arff` format file. The file contains the following lines.

- `@relation <relation name>`
- `@attribute <attribute name> <atttibute type>` – in case of conjunction features the `attribute name` is a query in prolog like syntax. Attribute types in this case are either `NUMERIC` or expressed as a set of possible values
- `@data` – followed by lines of comma separated values of the listed features. Lines correspond to examples.

4.3 Regression on Attibute Value Data

The final step is to perform regression on vector data. Several factors should be taken into account when chosing the method.

- Large number of features – the number of features found by ILP is potentially infinite. In reality only a finite number of features are evalueated, however, this number can still be high.
- Small number of data points – this is more of a case specific; in my experiment I will probably perform subselection on the data, making the problem simpler and also for performace reasons, since the feature mining is an expensive step.
- Missing values – values corresponding to certain features may be missing or may be assigned a inconsistent value. The attribute values are result from grounding FOL clauses with respect to Herbrand interpretations corresponding to the individual data points. A lot of features may not be present in some datapoints. Not all feature values are therefore relevant in all data points.

Chapter 5

Execution

This chapter descibes the implemented solution. Sections coresspond to steps outlined in section 4.

5.1 Data Preprocessing

For launching the replay files I use the `pysc2` [VEB+17] Python module (and some of its dependencies). The StarCraft II game engine implemented as client server. The cummunication between texttt pysc2 and the game is implemented in `s2clientprotocol` (`pysc2` dependecy). It is defined using Protocol Buffers, a data exchange format designed by Google.

Protocol Buffers (Protobuf) defines *messages* used to send data between applications (OSI presentation layer). The messages also represent data structures that can be saved to/parsed from byte files.

I defined a simple extension of the `s2clientprotocol` – `replay_dump_protocol`. It defines `ReplayDump` message Which is a wrapper for several messages produced by running the replay.

When launching the replay in the game engine one must specify which player is observed. And the observations received from the engine during the replay are from the perspective of that player. Therefore, to gather all data from the replay, it in necessary to run the game from both players' perspective.

I belive the extendable format of data storage and the `replay_parser.py` script is general enough to be used beyond the scope of this thesis.

In order to simplyfy the task I have only selected matches in which both players play the Terran race – about 15 % of the dataset. I also decided to only include games that take place on just one map. One of the most represented maps was *Acension to Aiur LE*. There are 650 games like that. In

`pysc2` tool `replay_actions.py` a function `valid_replay()` is used to test whether a replay file is not corrupted. It looks at the following:

- Replay is marked as corrupted.
- Game engine build does not match.
- Game duration is too short.
- There are not 2 players in the game.
- One of the players has low apm – suggest one player is inactive.
- One of the players has low MMR rating. This can be caused by a corrupt (according to comment in pysc2 code). Low MMR also indicates a low-skill player.

By further eliminating replays that are not valuable we are left with 433 files.

The game time is dicretized into frames. Each frame is cca. 0.04 seconds. Based on the Deepmind paper [VEB+17, 3.3]; acting every 8 game frames corresponds to 180 AMP (actions per minute).

I decided to step throught the replays in steps of 128 frames, which corresponds roughly to 5 seconds in game time. Initially I planned to use the same standard as in Deepmind paper [VEB+17] which is 8 frames. They were focusing on player actions rather than the state of the game. Player actions can be quite fast, the changes in game state are slower.

The program used to process the replay files is `replay_parser.py`; see appendix B.3.1.

It was launched with the following parameters:

```
python -m ReplayParser.replay_parser -replay_dir
<path_to_replay_files> -output_dir <path_to_output_directory>
-compress -filter tvt_ata_filter -step_mul 128
```

5.2 ILP Feature Mining and First Order Logic Representation

For feature mining I use TreeLiker which is introduced in section 4.2.1

5.2.1 Representation

The most evident information to include into the representation are player-controlled units and their types.

```
unit(<player>, <tag>, <type>)
```

Properly positioning your units is a crucial skill in the game. Unit positioning is relvant on both micromanagement level and on the macromanagement level. Let us then include distance between units to the representation.

```
dist(<unit_tag_1>, <unit_tag_2>, <distance>)
close/far(<unit_tag_1>, <unit_tag_2>)
```

Using the `dist` predicate allows us to include the numeric information about distance in the representation. Alternatively we could use a 'purer' FOL language by grouping the distances into ranges described by different predicates. I find the former approach better since it does not require us make any assumptions as to what grouping would be appropriate. In order to use the numeric data I need to use the Poly algorithm.

The score advantage I am predicting is included in the representation as well and it will be passed through the algorithm using the global constant attribute mode.

```
advantage(player_1_score - player_2_score)
```

For the purpose of splitting data into training and testing sets later, I inlcuded the name of a replay to which an example belongs as a global constant.

```
file(<replay_name>)
```

The representation is stored in *pseudo Prolog* syntax [citc]. Each example is represented by one line. The first word of a line is the example label – for classification tasks.[1] After the label there are comma-separated *Prolog* atoms – *Herbrand interpretation* reprsenting the example.

For creating the representation I used the `repre.py` script. Refer to appendix B.3.2.

Only data between 20th and 60th recorded frame frame were used, because not much is happening in the beginning of the game.

[1] I am doing a regression task, therefore I just use a placeholder string for a label.

5.2.2 Feature Mining

The following template is used with the described representation:

```
advantage(@advantage), file(@file), unit(#owner, +tag1, #type),
    unit(#owner, -tag2, #type), dist(-tag1, +tag2, *dist)
```

The Poly algorithm outputs features of the following type:

```
unit(#owner, T1, #type),
  dist(U1, U2, *dist),
unit(#owner, U2, #type)
```

Therefore the features a correspond to average distance between specific unit types of specific player. This setting of the algorithm theoreticly could produce more complex features grouping larger groups of units together.

Since I did not pose any further restriction on the features this behaviour seemed odd. It is probably caused by the fact that Poly tries to construct features that produce normally disrtibuted values and therefore ignores features that canot sufficiently approximate normal distribution. This is not mentioned the TreeLiker manual [citc], it is, however, implied in the related paper [KSHZ11].

5.3 Regression on Attribute Value Data

As a regression tool I decided to use ensamble regression trees. Trees are well suited for the large number of attributes produced by ILP. [2] A decision tree can also be interpreted as an extension of the logic framework. In the simplest ILP setting, where features are evaluated only existentially, decisions in individual vertices correspond to the truth value of their corresponding clauses in a given example. A path in a tree from the root to a leaf then corresponds to a conjuntion of the features.

Using a tree ensamble instead of a single tree helps dealing with generalizing a model learned on a small dataset. And again, as the decision of an ensamble model is usually a weighted sum of decisions of the submodels, there is a *fuzzy logic* 'feel' to it.

I used methods from python `scikit-learn` module [PVG+11].

[2] Some thoughts about possible future work: In this case features were mined exhautively and used as an input of the regression tree. Tree building and feature construction could possibly be done simmultaneously – only constructing only some features, maybe introducing some heuristic on the feature space.

5.3.1 Data Preparation

Data were prepared from feature values mined by the Poly algorithm (see. sec. 5.2).

First the data are split into groups by their originating replay file. Then those groups are split into a training and training set. The individual data instances in the training/testing set were created from groups of consecutive frames in each game. The feature values in the final vector are mean values of the. The score advantage value is a difference between the value in the first and last frame.

In my setting I used frames spanning 8 frames. Each frame corresponds to 128 game frames and each game frame corresponds to 0.04 seconds. That means that every vector in the final training/testing set corresponds to about 40 seconds of game time. Where the attribute values are mean values of the mined features over this time interval and the labels correspond to difference in score advantage of the 1st player in the beginning and end of the interval.

I had to deal with the problem of missing values. Since Poly uses values in the aggregation arguments, when there is no grounding for the argument in an example there is no way of producing the value.

When computing the mean values I only computed a mean of the present values and substitued for missing values in the next step. Given the context and that the produced values represent distance and that the features are not present in the example if no unit of one of the types is not present in the game, it makes sense to substitute a very large value. This could be interpreted as the given unit type being far outside the map, which would be equivalent to not being present.

I substitued value of `1e+20` for the missing values.

5.3.2 Evaluation

Each of the following experiments was run with training/testing data split roughly 6/4 over 10 random folds.

First set experiments was run using Adaboost with decision tree as the base estimator. We used
`sklearm.ensamble.AdaboostRegressor` and
`sklearn.tree.DecisionTreeRegressor`
from the `sci-kit learn` library [PVG+11]. The results are summarized in table 5.1.

n. trees	max. depth	min. data p. leaf	test R^2	train R^2
400	20	–	0.061	0.997
400	10	–	0.058	0.924
400	5	–	0.025	0.723
400	–	20	0.047	0.809
400	–	10	0.058	0.919

Table 5.1: Adaboost and decision tree results

Chapter 6

Conclusion

In order to manage the StarCraft II complexity I had to limit considerably the scope of information input into the FOL representation. I did so by focusing on relative unit positions. This seems reasonable because the ability to position units well is a crucial skill for players.

6.1 Experiment Results

The R^2 score of my adaboosted tree estimator was around 0.05 for most settings. This is an improvement over a naive extimator using just the expected value, but is not as good as I originaly hoped. However, given the complexity of StarCraft II, those results seem sufficient as a proof of concept.

It is important to note that the classifier performance on the training set was much better, which would suggest overfitting. However, the performance on testing set is not getting worse for the setting with R^2 values near 1 on the training set.

6.2 Future Work

The idea to use decision trees for the final regression came to me relatively late during the work; it was during a discussion with my lecturer[1] for a machine learning course in school.

I believe the decision tree estimator has a great potential working with features produced by ILP.

[1] Thanks to Vojtěch Franc, Ph.D. – Department of Cybernetics, CTU in Prague

- They are great for dealing with data with a large number of attributes and they can be used to select a subset of attributes.
- They can deal with missing data.
- Trees are extension of the logic abstraction – paths in a decision tree correspond to conjunctions.

I see potential in combinig ILP with decision trees and mining features during the tree construction, when ILP would be run in each node. Maybe some heuristic could be found to limit the number of features tested in each node. When applying the final tree on new data, only the features along the path in the tree would have to be evaluated in each example.

Appendix A

Bibliography

[cita] *Decision trees*, URL: scikit-learn.org/stable/modules/tree.html.

[citb] *Ensamble methods*, URL: scikit-learn.org/stable/modules/ensemble.html.

[citc] *Treeliker tutorial*, URL: ida.felk.cvut.cz/treeliker/download/treeliker.pdf.

[dR] L. de Raedt, *A perspective on inductive logic programming.*

[dR08] _____, *Logical and relational learning*, Springer, 2008.

[End16] U. Endriss, *Lecture notes: An introduction to prolog programming*, 2016.

[FS96] Y. Freund and R. E. Schapire, *A decision-theoreic generalization of on-line learning and an application to boosting*, Journal of Computer and System Sciences (1996).

[JR17] Niels Justesen and Sebastian Risi, *Learning macromanagement in starcraft from replays using deep learning*, CoRR **abs/1707.03743** (2017), URL: arxiv.org/abs/1707.03743.

[KB97] A. Kralic and I. Bratko, *First order regression*, Machine Learning (1997).

[KSHZ11] O. Kuzelka, A. Szaboova, M. Holec, and F. Zelezny, *Gaussian logic for predictive classification*, Tech. report, Faculty of Electrical Engieneering, CTU in Prague, 2011, URL: ida.felk.cvut.cz.

[KZ08] O. Kuzelka and F. Zelezny, *Hifi: Tractable propositionalization through hierarchical feature construction*, Tech. report, Intelligent Data Analysis Research Group Dept. of Cybernetics, CTU in Prague, 2008, URL: ida.felk.cvut.cz.

[KZ10] _____, *Block-wise contruction of tree-like relational features with monotone reducibility and redundancy*, Mach Learn (2010), URL: ida.felk.cvut.cz.

[LGKS17] Zeming Lin, Jonas Gehring, Vasil Khalidov, and Gabriel Synnaeve, *STARDATA: A starcraft AI research dataset*, CoRR **abs/1708.02139** (2017).

[Mug91] S. Muggleton, *Inductive logic programming*, New Generation Computing (1991).

[Plo71] G. D. Plotkin, *Automatic methods of inductive inference*, Ph.D. thesis, University of Edinburg, 1971.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, *Scikit-learn: Machine learning in Python*, Journal of Machine Learning Research **12** (2011), 2825–2830.

[Syn12] G. Synnaeve, *Bayesian programming and learning for multi-player video games: Application to rts ai*, Ph.D. thesis, Université de Grenoble, 2012.

[VEB+17] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Kuttler, J. Agapiou, J. Schrittwieser, J. Quan, S. Gaffney, S. Petersen, K. Simonyan, T. Schaul, H. van Hasselt, D. Silver, T. Lilicrap, K. Claderone, P. Keet, A. Brunasso, D. Lawrence, A. Ekermo, J. Repp, and R. Tsing, *Starcraft ii: A new challenge for reinforcement learning*, Tech. report, Deep Mind, Blizzard, 2017.

[YSA+12] J. Young, F. Smith, C. Atkinson, K. Poyner, and T. Chothia, *Scail: An integrated starcraft ai system*, Conference on Computational Intelligence and Games, 2012.

Appendix B

Attachements and Code Documentation

List of Attachements:

- DVD with code and replay file data
- Bachelor's Thesis Assignment

B.1 DVD Contents

```
/
|- AttributeValueLearning
|   |- learn.py
|- DataProcessing
|   |- DataExtraction
|   |   |- arff_reader.py
|   |   |- repre.py
|   |- ReplayParser
|       |- filters.py
|       |- rdump_reader.py
|       |- replay_dump_protocol
|       |   |- proto-source
|       |   |   |- replay_dump.proto
|       |   |   |- s2clientprotocol
|       |   |       |- ...
|       |   |- replay_dump_pb2.py
|       |- replay_parser.py
|- FeatureMining
|   |- binaries_treeliker
|   |   |- lib
|   |   |   |- Jama.jar
|   |   |- TreeLiker.jar
|   |   |- treeliker.pdf
|   |- out_tt
|   |   |- ...
|   |- test.fol
|   |- TLParams.treeliker
|   |- train.fol
|- PredictingBlizzardScoreInStarCraft2.pdf
|- README
|- ReplayDumps-AscensionToAiur-TvT.tar.gz
|- ReplaySelection.tar.gz
```

The directory

```
/binaries_treeliker
```

contains the `.jar` files for TreeLiker.

```
/DataProcessing
```

Contains Python code used for data processing and the learning script.

```
/AttributeValueLearning
```

Contains Python code used for experiments.

```
ReplayDumps-AscensionToAiur-TvT.tar.gz
ReplaySelection.tar.gz
```

These archives contain raw replay files (only the subselection used fot the experiment) and preprocessed replay files in the protobuff format for easier access to data.

B.2 Code Documentation

For TreeLiker documentation refer to the original manual [citc] It is included on the DVD.

```
/binaries_treeliker/treeliker.pdf
```

B.3 /DataProcessing

`/DataProcessing` contains Python scripts used to process the raw replay files and transform them to attribute value data.

B.3.1 /ReplayParser

`/DataProcessing/ReplayParser/replay_parser.py`

contains a program for convertiong `.SC2Replay` files to `.RDump` files; containing selialized `ReplayDump` objects defined by `replay_dump_protocol`.

This program supports the `-help` parameter whose output is:

```
-[no]compress:  Produce compressed files
(default:  'false')
-[no]disable_fog:  Disable fog
(default:  'false')
-filter:  Name of the filter function to use (in filters.py)
-output_dir:  Directory for output files
(default:  '.')
-parallel:  Number of processse
(default:  '1')
(an integer)
-replay:  Replay file
-replay_dir:  Directory with replay files
-rndseed:  Specify random seed
(an integer)
-sample:  Number of replays to compute stats
(default:  '-1')
(an integer)
-[no]shuffle:  Shuffle files
(default:  'false')
-[no]stats:  Get stats for replays
(default:  'false')
-step_mul:  Game step multiplier
(default:  '8')
(an integer)
```

`/DataProcessing/ReplayParser/filters.py`

contains definitions of filter functions that can be used to select replay files based on metadata.

The filter I used is `tvt_ata_filter()` accepts replays where both players play *Terran* and that take place on *Ascension to Aiur LE map*.

`/DataProcessing/ReplayParser/rdump_reader.py`

contains definition of function `read_replay_dump(<file_path>)`, which loads a `ReplayDump` object form a `.RDump` file.

`/DataProcessing/ReplayParser/replay_dump_protocol/`

this directory contains Protocol Buffers source files and the `ReplayDump` message compiled to Python (`replay_dump_pb2.py`).

B.3.2 /DataExtraction

`/DataProcessing/DataExtraction/repre.py`

Creates FOL representation for TreeLiker from `.RDump` file/s and outputs it to standard output.

Usage:

```
python -m DataExtraction.repre <path> <first> <last> <sample>
```

- `<path>` – path to a dump file or directory with dump files.
- `<first>` – index of the first [1] game frame to process.
- `<last>` – index of the last game frame to process.
- `<sample>` – determines how many files to process if `<path>` is a directory.

`/DataProcessing/DataExtraction/arff_reader.py`

Defines function for loading data from `.arff` files.

- `load_arff(<path>)` – loads data from file in `<path>` as `{'attributes': [(<attribute_name>,<attribute_type>), ...], 'data':[[...], ...]}`
- `example_vector(<array>, <span>)` – returns a masked numpy array of attribute value vectors. `<array>` is array of values for each frame in on game. `<span>` determines how many consecutive frames to use to create each row in the return array.
- `get_numpy(<data>, <span>, <test_fraction>` – Creates a numpy (masked) array with rows as attribute value vectors from `<data>` – return value of `load_arff()` function.

[1] frame number depends on step size used do create the dump

B.4 /AttributeValueLearning

`/AttributeValueLearning/learn.py`

A script used for learning from the attribute value data. The code loads (see. sec. 5.3.1) data from an `.arff` file given as an only argumet to the sript.

The rest of the code used `scikit-learn` library and `numpy`. It is meant to be freely edited by user.

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Valouch David** Personal ID number: **460514**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Branch of study: **Computer and Information Science**

II. Bachelor's thesis details

Bachelor's thesis title in English:

Predicting Blizzard score in StarCraft II

Bachelor's thesis title in Czech:

Predikování Blizzard skóre ve StarCraft II

Guidelines:

Familiarize yourself with the computer game of StarCraft II and review approaches to bot creation for the game.
Formulate the problem of learning the Blizzard score in the game as a symbolic regression task.
Propose a symbolic representation for the game state and create or collect an appropriate dataset.
Propose a method with usage of Inductive Logic Programming to predict the score.
Experimentally evaluate the proposed representation and your method on the dataset.

Bibliography / sources:

[1] VINYALS, Oriol, et al. StarCraft II: a new challenge for reinforcement learning. arXiv preprint arXiv:1708.04782, 2017.
[2] DE RAEDT, Luc. Logical and relational learning. Springer Science & Business Media, 2008.

Name and workplace of bachelor's thesis supervisor:

Ing. Martin Svatoš, Intelligent Data Analysis, FEE

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **11.01.2018** Deadline for bachelor thesis submission: **08.01.2019**

Assignment valid until: **30.09.2019**

Ing. Martin Svatoš	doc. Ing. Tomáš Svoboda, Ph.D.	prof. Ing. Pavel Ripka, CSc.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

. _____
Date of assignment receipt

_____
Student's signature