BAR ILAN UNIVERSITY

Faculty of Exact Sciences

School of Graduate Studies

EMPIRICAL EVALUATION OF AUTONOMOUS AGENTS SOFTWARE USING CODE METRICS

A thesis submitted toward the degree of

Master of Science in Exact Sciences

by

Alon Zanbar

2021

BAR ILAN UNIVERSITY
Faculty of Exact Sciences
School of Graduate Studies

EMPIRICAL EVALUATION OF AUTONOMOUS AGENTS SOFTWARE USING CODE METRICS

A thesis submitted toward the degree of
Master of Science in Exact Sciences

by

Alon Zanbar

This research was carried out at Bar Ilan University
in the Department of Computer Science
Faculty of Engineering
under the supervision of Prof. Gal A. Kaminka

2021

Acknowledgments

This research would not have been possible without the careful and knowledgeable guidance of my advisor Prof. Gal A. Kaminka. Thank you for opening the door of knowledge for me and sharing with me the excitement and curiosity of my first thesis. Obviously, to my one and only wife, Tamar and my wonderful kids: Hoshen, Moriya, Tchelet, Noam and Nave, you gave the strength to keep trying and the incentive to achieve this dream. To my parents that made me what I am and pushed me at youth and maturity to explore the world of Math and Computer Science. And lastly, God, Thanks for sharing your wisdom with flesh and blood.

Abstract

E mpirical research based on agent software repositories using commonly used software metrics, which are used in software engineering literature to quantify meaningful characteristics of software based on its source code. In the first part agent software measurements are contrasted with those of software in other categories. Analyzing hundreds of software projects the commonality and uniqueness of the two groups is presented. The second part describes an attempt to use the information extract from the code metrics to identify behaviors that impact program performance.

Table of Contents

List of Figures

List of Tables

1 Introduction

For many years, significant research efforts have been spent on investigating methodologies, tools, models and technologies for engineering autonomous agents software. Research into agent architectures and their structure, programming languages specialized for building agents, formal models and their implementation, development methodologies, middle-ware software, have been discussed in the literature, encompassing multiple communities of researchers, with at least partial overlaps in interests and approaches.

The most important underlying assumption of these research efforts is that such specialization is *needed*, because autonomous agent software poses engineering requirements that may not be easily met by more general (and more familiar) software engineering and programming paradigms. Specialized tools, models, programming languages, code architectures and abstractions make sense, if the software engineering problem is specialized.

A broad overview of the literature reveals that for the most part, the truth of this assumption has been supported by qualitative arguments and anecdotal evidence. Agent-oriented programming [40] is by now a familiar and accepted programming paradigm, and countless discussions of its merits and its distinctiveness with respect to other programming paradigms (e.g., object-oriented programming, aspect-oriented programming) are commonly found on the internet. Agent architectures are commercially available as development platforms and are incorporated into products. Indeed, agent-oriented software development methodologies are taught and utilized in and out of academic [36, 16, 21, 6].

However, there is a disturbing lack of quantitative, empirical evidence for the distinctiveness of autonomous agent software. Lacking such evidence, agent software engineers rely on intuition, experience, and philosophical arguments when they evaluate or advocate specialized methods. The basic data we use for the research are common code metrics. We utilize general source code metrics, such as *Cyclomatic Complexity*, *Cohesion*, *Coupling*, and others. These metrics (see below) are commonly used by researchers and practitioners to assess code quality, estimate work effort, and to quantify other meaningful characteristics of software. The research cover two hypotheses: one that common code metrics of a program holds descriptive information about the program and can be used to identify unique characteristics of AI software. Second, the performance of AI software is impacted by code quality and this quality can be evaluated by common code metrics. The first hypothesis is addressed by comparing AI to other software domains. We quantitatively analyze over 500 software projects: 140 autonomous agent and robotics projects (from RoboCup, the Agent Negotiations Competitions, Chess, and other sources), together with close to 400 automatically selected software projects from github, of various types. For validation of the second hypothesis , unique information of the Robocup programs is used: performance of achieving the task of wining a football competition. The performance of each team with relation to the code metrics of the the program is investigated to reveal patterns of impact.

2 Background

The fundamental questions lay under our research are basically: are there special characteristics of autonomous agent development? what are they? How can we use this unique form in agent software design?

There is vast literature reporting on research that directly or indirectly impacts software engineering and development of autonomous agents: agent architectures, agent-oriented programming languages, formal models and their implementation, development methodologies, middleware software, and more. We cannot do justice to these efforts for lack of space. For brevity, we use the term *agent-oriented software engineering* (AOSE) to refer to the combined research area, With due apologies to all the different threads of work whose unique contributions are blurred by our choice.

AOSE is a thriving area of research, with at least one dedicated annual conference/-workshop and a specialized journal[1]. [40, 25, 42, 36, 38, 44]. For the most part, the arguments for the study of AOSE as distinct from general software engineering are well argued *philosophically*, and *qualitatively* pointing out inherent conceptual differences between the software engineering of agents. To the best of our knowledge, little quantitative empirical evidence—certainly not at the scale detailed below—has been offered to support these important conceptual arguments.

Closely related, pioneering works into software engineering in robotics similarly argue *qualitatively* for distinguishing software engineering in robotics [8]. For important features and design methodologies. Some emphasize specific middle-ware frameworks like JaCaMo [6] or O-MAZE [16] (e.g., [20, 10, 39, 17, 43]), while others focus on critical capabilities or approaches [35, 14, 9]). The underlying implicit assumption is similar to those in AOSE: that robotics software is sufficiently different from general software, that it merits distinct methodologies and tools to ease software development. As agents researchers would (in general) argue that robots are a special case of agents, this should not be surprising.

[1] International Journal of Agent-Oriented Software Engineering

However, researchers have repeatedly found that using methods from software agents in robotics is not trivial, and require significant changes and extensions to the original software agents tools and methods [27, 28, 31]. These findings essentially make qualitative arguments, based on case studies, rather than strong, cross-task empirical studies. Indeed, we report below that robot code is similar in some aspects to autonomous agents code, but is not as easily distinguished from general software. For example, we do not know whether agent-based development methodologies are more suited to robotics software development, than—for example—methods and approaches for software engineering of enterprise server systems, or operating systems.

Code metrics , When coming into the space of common code metrics it is fair to say that the history of quantitative methods for evaluating general software is almost as old as computer engineering itself. Many different researches were conducted on the subject over the years with variate of metrics is well as many different applications for the metrics proposed.

1970s pioneering research on Cyclomatic Complexity [32] and Halstead measures [22] there have been many investigations both proposing quantitative metrics of software constructs, and relating the measurements to software quality, development effort, software type, and other attributes of interest [1, 26, 5]. For example, metrics such as *Cyclomatic Complexity*, *Coupling*, and *Cohesion*—generated from analysis of the software source code and the program control flow graph— have been shown to correlate with defects [32, 11, 24]. Maintaining their values within specific ranges (or below some thresholds) tends to lower the expected *defect creation rate*, and improve other measures of software quality. Development and exploration of software metrics continues today, e.g., for paradigms such as aspect-oriented programming [37].

Applied statistics and Code metrics , In addition to the role software metrics have been playing in evaluating software properties, some researches proven its effectiveness in classifying software, set guidelines for good programming and determine the attractiveness of open source projects. Classification of software domains is also promising approach for revealing software attributes. For an example [29] showed LOC (among other product level metrics) a reliable feature for clustering software for their cost or effort. [15] showed significant difference in values of "coupling" metrics between different software categories. Based on those finding they suggested that weights and values of specific metrics should be enforced with adjustments to the domain of the software. [41] adapted the classification method to Android projects. Another example was found by Meirelles [33], who found linkage between the size and complexity of open source projects to attractiveness of the project for contributors. Investigating the uniqueness or commonality of AI code based on those techniques is just another implementation that have not been looked at.

Those and other techniques that are being used with general software research for many years. In this research we make an attempt to evaluate those general quantitative techniques on agent code analysis to answer questions about agents software architecture as well a verification of validity of the common code researches in this software domain. We think that by addressing the above we are opening the path to expend the tool-set of analysis and researches on AI code and make it less limited to its own architecture specific methodology and measurements.

our research goal, to compare ASOE and general software on the same common metrics prevent us from using those ASOE specific metrics. Furthermore, we were forced to neglect project code portions that used ASOE special architecture as those do not enable general code metrics analysis. We think that the code hold enough data even without the agent specific metrics to compared fairly to general software repositories

3 Code Analysis

There are few components of the AI software development that impact the final attributes and the performance of the resulted program. Among the rest, there are process characteristics, domain knowledge and the quality of the requirements. However, this research is focused on code metrics solely for several reasons: First, code metrics, as opposed to other attributes, has a simple and direct quantitative representation, this is crucial when coming to conduct large scale research that aims to draw general conclusions. For example metrics like domain expertise can only be estimated roughly but can hardly be evaluated numerically. Second, code metrics can be used for different types of software domains. Finally, code metrics preserve information about the code design.

The question about the impact of different code metrics on the maintainability and development effort is a subject to a well established research. Its results which shows relation between code metrics to several effort and maintainability measures is being used for decades by the industry for measuring quality and predicting development effort.

Software code metrics can roughly be divided into three main groups: size metrics, complexity metrics, interrelations metrics. Although those groups might represents different aspects of software analysis, literature reveals relationship between the groups. The applications that were suggested for code metrics are among the rest: quantitative expression of code, quality evaluation, maintenance cost, classification of software, developer identification.

The code metrics used in this research are described below alongside a support for using it in our research.

3.1 Source Code Metrics

3.1.1 Size metrics

Lines Of Code (LOC) A common basis of estimate on a software project is the LOC or Lines of Code. LOC are used to create time and cost estimates. The LOC estimate becomes the baseline to measure the degree of work performed on a project. Once a project is underway, the LOC becomes a tracking tool to measure the degree of progress on a module or project. An experienced developer can make a LOC estimate based upon knowledge of past productivity on projects. The LOC measurement becomes the barometer for the program‘s progress and productivity. There are few ways of counting LOC:

LOC - All lines of Code without any filtering.

eLOC - An effective line of code or eLOC is the measurement of all lines that are not comments, blanks or standalone braces or parenthesis. This metric more closely represents the quantity of work performed.

iLOC - Logical Lines of Code, Logical lines of code represent a metrics for those line of code which form code statements. These statements are terminated with a semi-colon. The control line for the *for* loop contain two semi-colons but accounts for only one semi colon. Lines of Comments - Lines of comments and the relation between it and other size and complexity measures might suggest the level of attention invested in code. Also many comments might lower the maintenance cost of a program. The different ways of calculating LOC are demonstrated in table 3.1

Source code line	LOC	eLOC	lLOC	Comment	Blank
if ($x < 10$) // test range	*	*		*	
{	*				
// update y coordinate				*	
					*
$y = x + 1;$	*	*			
}	*				

Table 3.1: Compression between different Line Of Code metrics

Average Method LOC, Maximum method LOC (AMLOC, MMLOC) The LOC of the different layers of the program indicates if the code is well distributed between the methods or modules How bigger, "heavier" they are. It’s preferable to have a lot of small and of easy understandable operations than a few large and complex operations.

3.1.2 Object oriented design metrics

Afferent Connections per Class (ACC) Measures the connectivity of a class as follows. Let (Ci => Cj) be a one edge of the directed graph defined by a unique call from one class Ci to another Cj is_client(Ci, Cj) = 1, if (Ci => Cj) exists and (Ci => Cj) does not exists is_client(Ci, Cj) = 0, otherwise. So $ACC(Cj) = (sum(is\_client(Ci,Cj)), i = 1toN)$, where N is the total number of system classes. If the value of this metric is large, a change in the class has substantially more side effects, making maintenance more difficult [34].

Coupling Between Objects (CBO) Similar to to ACC but when ACC is directed, meaning it counts only "outgoing" calls, CBO is in-directed thus, it counts outgoing and incoming interactions. Thus, for CBO any call between a pair of class is counted

Average Number of Parameters per method (ANPM) A large amount of parameters per function might indicate a flew in design where function is doing more task.

Coupling Factor (COF) Considers inheritance, polymorphism, method overriding, and direct methods of invocations to identify possible interactions in the system that contribute to the software complexity. Formal definition of this framework is

$$COF = \frac{\sum_{i=1}^{TC}\left[\sum_{j=1}^{TC} is\_client(C_i, C_j)\right]}{TC^2 - TC - 2 \times \sum_{i=1}^{TC} DC(C_i)}$$

where *is_client* defines a relation between two classes that is not inheritance, *TC* is the total number and $2 \times \sum_{i=1}^{TC} DC(C_i)$ defines the maximum number of interactions due to inheritance. The numerator then represents the actual number of couplings not imputable to inheritance. The denominator stands for the maximum possible number of non-inheritance couplings in a system with *TC* [7].

Depth of Inheritance Tree (DIT) This measure might indicate under design when this number is low for a large part of the program and over design when the Depth is very high [37].

Lack of Cohesion of Methods (LCOM4) A class should have one responsibility taken care by its internal method and attribute. When class methods can be separated to unconnected components, meaning there is no common internal parameter they are accessing, it might suggest the class has more than one responsibility [23].

Number of Attributes, Methods, Public Attributes, Public Methods (NOC, NOM, NPA, NPM) Calculates the number of different members of a class. Its minimum value is zero and there is no upper limit to its result. A class with many members may indicate that it has many responsibilities and presents a low cohesion, i.e., is probably dealing with several different subjects. Also the Rate between public and private members might indicate bad object oriented design of the program

Number of Children (NOC) The number of direct subclasses a class has. When NOC is high it might indicate good code reusability. On other hand a large number of Children increase the potential impact of of an error in the class and might suggest bad responsibility design [30].

3.1.3 Control Flow Graph measures

Control Flow Graph The most basic description of a computer program is the its Control flow graph. The CFG presents the statements as nodes and the sequence of executing the statements as edges of the graphs. Statements in the CFG (and generally in software) as divided to two types: state changing statements and control statements. State changing statements are every statement that impact the world state (print to screen, variable value change etc.) And they keep the natural sequential flow of the program. Thus, a state changing statement will be connected with an outgoing edge to the next statement in the program. Control statements potentially change the natural flow of the statement and can connect with edges with multiple nodes / statements in the graph in arbitrary distance. Control statements are among the reset: conditions, jump, loops etc. Describing a program as a Graph enables extracting attributes that describe the flow of the program in graph analysis tools,

3.1 shows the structure describing basic flows in a program. CFG is a powerfully presentation that enables cross language and cross platform analysis as it is generic and language independent. Yet the structure describes same algorithm in different language might have different CFG structure and expose some language specific attributes as shown by [4] in a paper measuring the different graph structure of the same algorithm in different language

Control	Cyclomatic Complexity
Sequential	CC = 1 - 2 + 2 = 1
IF-THEN-ELSE	CC = 4 - 4 + 2 = 2
WHILE	CC = 3 - 3 + 2 = 2

Table 3.2: Cyclomatic Complexity of Basic Flows

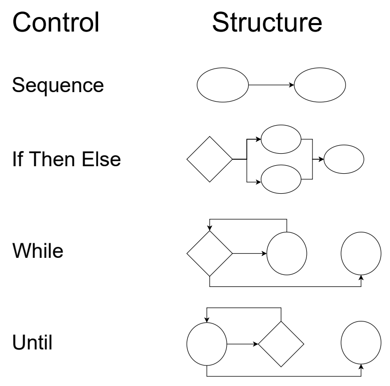


Figure 3.1: Control components

Cyclomatic Complexity and Average Cyclomatic Complexity per Method (ACCM) (from wikipedia) The cyclomatic complexity of a section of source code is the number of linearly independent paths within it. Mathematically, the cyclomatic complexity of a structured program[a] is defined with reference to the control flow graph of the program, a directed graph containing the basic blocks of the program, with an edge between two basic blocks if control may pass from the first to the second. The complexity M is then defined as

$$M = E - N + 2P$$

where

E = the number of edges of the graph. N = the number of nodes of the graph. P = the number of connected components.

For instance the complexity of each flow presented in 3.1 per structure is calculated in 3.2:

Cyclomatic complexity was investigated thoroughly and is being used in research in industry with several implantation in software quality and software architecture. Program CC is proven to be highly correlated with Program size therefore we collect Average CC of methods in the module in order to expose the code standards / code behavior. This research we uses CC as in indicator to the amount of logical trajectories in a code a term that will be later elaborated and show cased.

Collections statistics , The code metrics described above are collected by the analysis tools we used at the module level. For example in C++ programs it will calculate the metrics in the class level while in C programs, files are the calculated level. Since our research is focused on the project level, single module metrics are aggregate to the project level using the following statistics: [min, max, 25%, 50%, 75%, mean], to get the metrics we use the our analysis. For example amloc mean represents a "double" averaging on the length (in lines of code) of the methods in the program: first, averaging method LOC inside each module and than averaging the results of all the individual modules.

4 Data Collection and Curation

The fundamental part of any empirical research is its data. Defining a data harvesting methodology, extracting enough samples and reducing data errors has a critical impact on the results and reliability of data based research like this one. The research described in this thesis attempts to answer questions about attributes of a autonomous agents software by extracting information from samples of software programs. This chapter describes the process of preparing the data empirical analysis. It specifies the following parts :

1. Code repositories used as data sources for the research.
2. Automatic data harvesting process used to allocate and download those repositories.
3. Methodology used to reduce errors and validate the correctness of the extraction process and of the data collecting.

4.1 Data Sources

Data variability is a key element in robustness of a data study. therefore the need of different source of code repository having different attributes that potentially will reduce biases of homogenise data. We begin with an overview of data collected and will be used in the analysis processes described in Section 4.2. We use several sources of software projects, each containing multiple projects. These are described below.

RoboCup. RoboCup is one of the oldest and largest annual global robotics competition events in the world—taking place since 1997. The event is organized in several different divisions (soccer, rescue, junior/educational, and more). Within each division, there are multiple leagues, with their own rules and requirements. For example, within the soccer division, there were over the years up to three different simulation-based leagues (*2D*, *3D*, and *coach*), and several physical robot competitions (standard platform, small-size, mid-size, and two humanoid leagues), restricted to different sizes). In addition to the main world-cup event, there are regional competitions which take place during the year. The competitions themselves are between completely autonomous agents/robots; No human in the loop. In most cases, the agents run in completely distributed fashion, without a centralized controller.

The bulk of the code in the various leagues is written by graduate students and researchers in robotics and artificial intelligence, some from top universities in these fields. The simulation leagues follow an internal rule, which requires all teams to release a binary version of their code within a year following the competition. Source code release is not required, but strongly encouraged. Indeed, we use the source code from many 2D simulation league teams, downloaded from their repository server. In addition, we used source code from other RoboCup soccer leagues, gathered from the internet.

GitHub. GitHub has more than 24 million users and more than 67 million code repositories. It is the largest repository of open source projects in the world. GitHub exposes robust API for finding repositories using extensive query language, which we used to find relevant project for analysis. Repositories in GitHub are categorized by users using tags, which we used to categorize software projects.

Automated Negotiating Agent Competition (ANAC). The annual International Automated Negotiating Agents Competition (ANAC) is used by the automated negotiation research community to benchmark and evaluate its work and to challenge itself. The benchmark problems and evaluation results and the protocols and strategies developed are available to the wider research community. ANAC has similar properties to the RoboCup in the sense of emphasizing autonomous agents. It is a popular competition for software agent researchers, maintains a requirement that all the sources of the agents participating in the competition are made available for research. We collected ANAC software agent projects from the competition web site.

Table 4.1 presents a breakdown of the number and categories of the harvested software projects in the dataset (almost a terabyte). In total, there were 118 projects generally classified as autonomous agents for software or virtual environments.

Classification	Source	Software Domain	Number of projects
Autonomous Agents	RoboCup 2D simulation	Virtual Robots	71
	ANAC	Negotiating Agents	105
	GitHub	Chess playing Engines	60
		Chatbot	65
AI platforms	GitHib	Deep Learning	62
		Reinforcement Learning	59
General	GitHub	Audio	73
		Education	129
		Finance	108
		Games	89
		Graphics	36
		IDE	134
		Mobile Applications	136
		Security	68

Table 4.1: Software categories breakdown

4.2 Data Collection

The data we are using in the research is collected from different sources but is analyzed using a common pipeline. Thus, before making the actual analysis some prepossessing steps were carried out, the transformations the data source has gone through is different depending on the source but the outline of the process is the same. Note, that that process of preparing the data for the research is cyclic by nature: the data should be validated and inspected and then a need for more or different data arises, leading to another cycle of fetching data and reprocessing it is than cried out etc. still, for the sake of simplicity the process described below as if it was executed once. The prepossessing includes the following steps:

1. Data definition: setting the attributes of the data to harvest, source url, Category, size etc.

2. Data retrieval: Fetching the data from external sources to a file system in the lab

3. Restructure: Restructuring the file system to a structure required by the analysis pipeline

4. Meta data: Extracting meta data for each project

5. Static code analysis: harvesting code metrics for each project

The prepossessing phase as well the measurement pipeline are done in a map reduce paradigm. Each project is analyzed separately and the results of each step is saved in the local folder of the project. Later in the process those results and meta data files will be collected to create the common data set. Also, the pipeline implemented a continuation mechanism the eliminating the need to rerun steps that were already executed. This design supported a big data ETL (Extract, Transform, Load) and analysis processes executed by 24 cores in parallel running for over a week.

As explained in the previous chapter LOC is the most basic metric to evaluate the size of a program. This metric is also easy to extract from any kind of a program and gave us ability to filter data before we run the full resource consuming pipeline for extracting more advanced metrics. The filtering based on LOC was done to validated our data is not strictly biased toward domains or categories that are much bigger or smaller in terms of code size. Filtering based on the size is done because programs that are much different in their size might have different properties regardless of its software category. For example a very small program might be consider easy enough to handle or read and not be developed in code standard and care as bigger program that does not have the same simplicity.

4.2.1 Robocup data

In order to overcome the lack of meta data on the project we collected from robocup archive a process of extracting information from the files of the project was executed. In this process we collected the below information for each project: competition name and year, group name, does the project contains source code, code language of the project, etc. At the phase of collecting the data some curation has to be carried in order not to feed the final analysis with wrong data.

Many of projects contains some binaries without clear information which was used for the competition. For that an iterative rule based (partially manual) process was executed. In each iteration a smaller subset of indeterminate executable were retrieved and the rules were adjusted until all projects had one player file and up to one coach file.

4.2.2 github data

The process of collecting and filtering of repositories from GitHub was *automatic*, as described in 4.1. The primary constraint in selecting software projects is comparability. The source code collected for agents uses C++, and Java, and so we restricted ourselves to projects in these languages, to allow meaningful comparison to the agents code. Similarly, we restricted ourselves to software size (measured in lines of code—LOC) in comparable ranges, and belonging to software categories other than agents or AI:

- Programming languages: C++, Java
- high Level of maturity (measured by github stars)
- Distinct classification in github (for github projects)
- Size $>$ 900 lines of code (practically filtered at the retrieval phase by file system size and later at the analysis fine tunes by LOC)

The only exception for this process is repositories from Chess tag which was defined as target repository manually. Chess project where collected as control group to lower the bias we have for competition code in our "agent" group. Chess repositories are part of the "agent" group for the rest of the research.

Furthermore, as main data-source only used repositories that has more than 50 projects that matched the above the criterion, in some cases we also used smaller repositories as secondary data set to support our findings.

4.3 The Measurement Pipeline

The essence of the process is the measurement, i.e., the generation of measurements from applying code metrics to the software. We focus on source code metrics in this paper. our pipeline handles binary data as well as source code. The flow and tools are much different and they will be describe separately below:

4.3.1 Source Code Measurement Pipeline

The source code of each project was processed to extract two different data structures: a control flow graph, and a code statistics database. These, in turn, are used to calculate several different metrics. For Source code we used two different tools, independently, to allow validation of the results: CCCC[1] and Analizo[2].

[1] http://cccc.sourceforge.net/
[2] http://www.analizo.org/

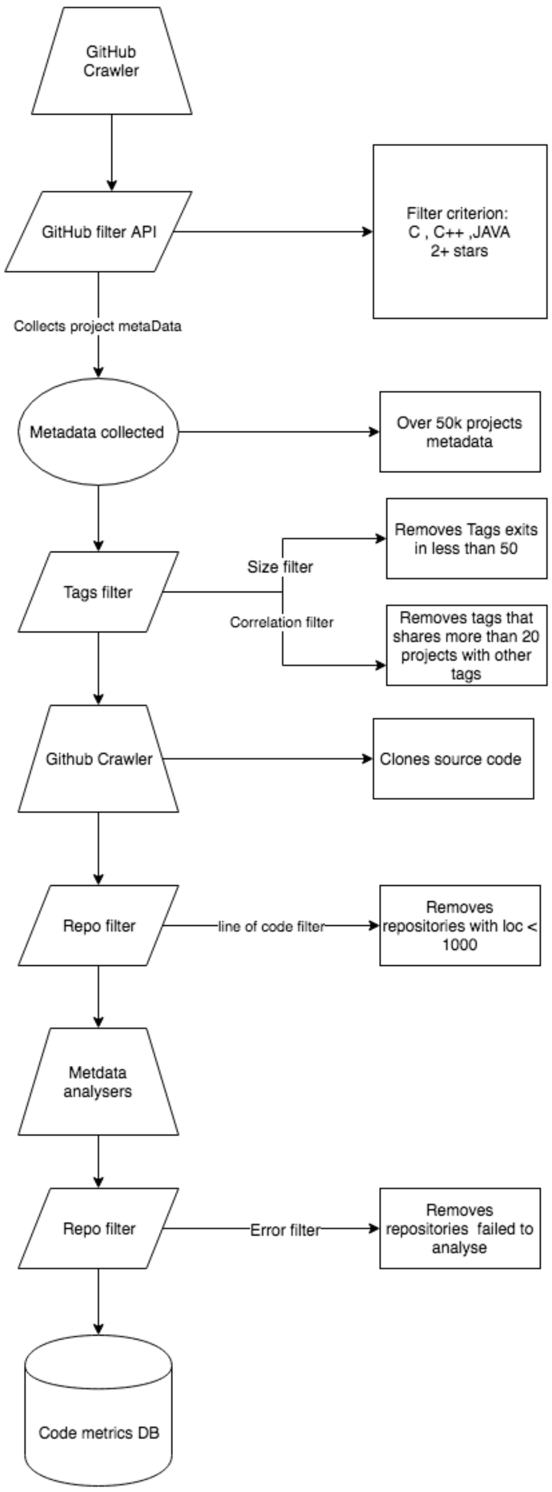

GitHub Crawler
GitHub filter API
Filter criterion: C , C++ ,JAVA 2+ stars
Collects project metaData
Metadata collected
Over 50k projects metadata
Tags filter
Size filter
Removes Tags exits in less than 50
Correlation filter
Removes tags that shares more than 20 projects with other tags
Github Crawler
Clones source code
Repo filter
line of code filter
Removes repositories with loc < 1000
Metdata analysers
Repo filter
Error filter
Removes repositories failed to analyse
Code metrics DB

Figure 4.1: Automatic flow of selecting GitHub projects

The measurement tools provide the following general software metrics, for different level of analysis (see [18] for detailed descriptions). As with the restriction on choice of language, we are restricted to using general metrics as they allow for measuring non-agent code. Otherwise, we'd be able to use code metrics specific to AOSE [19, 3, 2, 12], and specialized languages (e.g., 2/3APL, JASON).

To allow smooth recurrent execution of the automatic analysis process an initial debug and fixing phase was conducted on the some of the repositories to remove files that failed the execution. during the execution the program logs progress in a central and distributed way to enable debugging and rerunning of analysis skipping already analyzed repositories. Finally the metrics were extracted and aggregated for each module and and some general metrics were collected for the project as a whole. The focus of our research is the structure and profile of the code and not the quality aspects so we used the type of metrics that provide such information. The metrics the we collected are described in the background.

Table 4.2 lists the minimum, maximum and median project size in each domain, measured in LOC. Overall, almost a terabyte of project data was collected and analyzed.

	min	mean	max
category			
ANAC	40	3629	205632
Audio	78	25647	182806
Business	44	21630	182806
Chatbot	152	2852	18951
Chess	49	8467	59646
Deep-Learning	276	39958	471213
Education	132	18261	182806
Finance	93	15541	147279
Games	132	20522	182806
IDE	107	27702	182806
Mobile	107	14938	182806
Reinforcement-Learning	375	16207	363987
Robocup-2D	329	38945	153661
Security	132	24476	182806

Table 4.2: min, max, and mean total LOC.

.

4.4 Data validation and curation

In order to achieve reliable results, several cycles of validation of the data are preformed. The first phase is an overview analysis of the metric extraction process: some projects are not successfully analyzed because of technical reasons (bad characters or usage of platform that are not compatible with the extraction tools). Those projects are fixed or replaced with another project in the same domain.

4.4.1 Data Curation

After finalizing the raw dataset that contains all the projects to be analyzed, The metrics of the internal modules are deeper investigated and cleaned to increase data integrity. Log histogram of the different module level metrics presented in figure 4.2 reveals that there are some extreme outliers that dominant the model of this specific metric. Exploring the data exposes that for some metrics there are data records that have invalid values that were probably caused by a wrong calculation. For example in ACCM there some records that have ACCM value of more than 1E9. We reported the bug to the owners of the package and clean the defected data as explain below.

Next, the values collected for each project are scanned to find extremely high values. Few methods for capturing samples with values that are extreme outliers were experimented: n top, upper/lower X percentile, etc. The method we selected for this task is removing samples that has values (in one metric) that are N times IQR higher then the 75% precnetile (Q3) or lower than the 25% precnetile (Q1) minus N time the IQR. This method is chosen for most cases of uni-variant outlier removal in this research as it is robust to scale and it is would not remove outlier if there is not high variance (as apposed to the other listed methods which are always removing samples). The method expressed in equation 4.1: For a given dataset X an extreme outlier is define as

$$EO(x_i) = x_i > Q3 + 100IQR(X) \vee x_i < Q1 - 100IQR(X) \tag{4.1}$$

. The above method was selected and is used in further steps of the research

As explained, one of the reasons for the usage of two different analysis tools is to find metrics that are common between the tools but are not correlated as expected. figure 4.3 presents the initial correlation matrix which exposes unsatisfactory correlation between the module level size metrics of both tools (84%). Calculating the normalized residual on the LOC from both tools reveals projects that one of the tools failed to analyze correctly

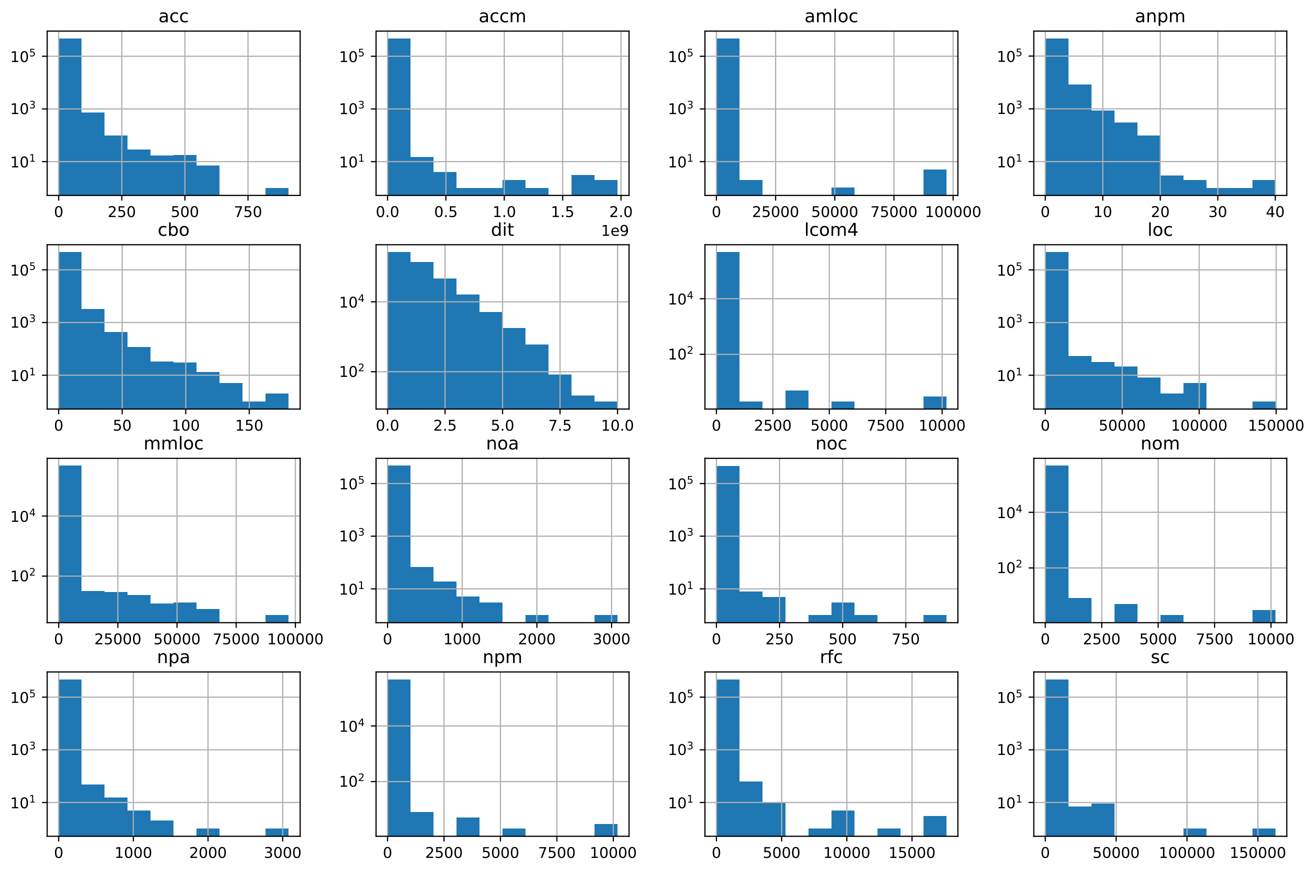


Figure 4.2: Module level code metrics histograms

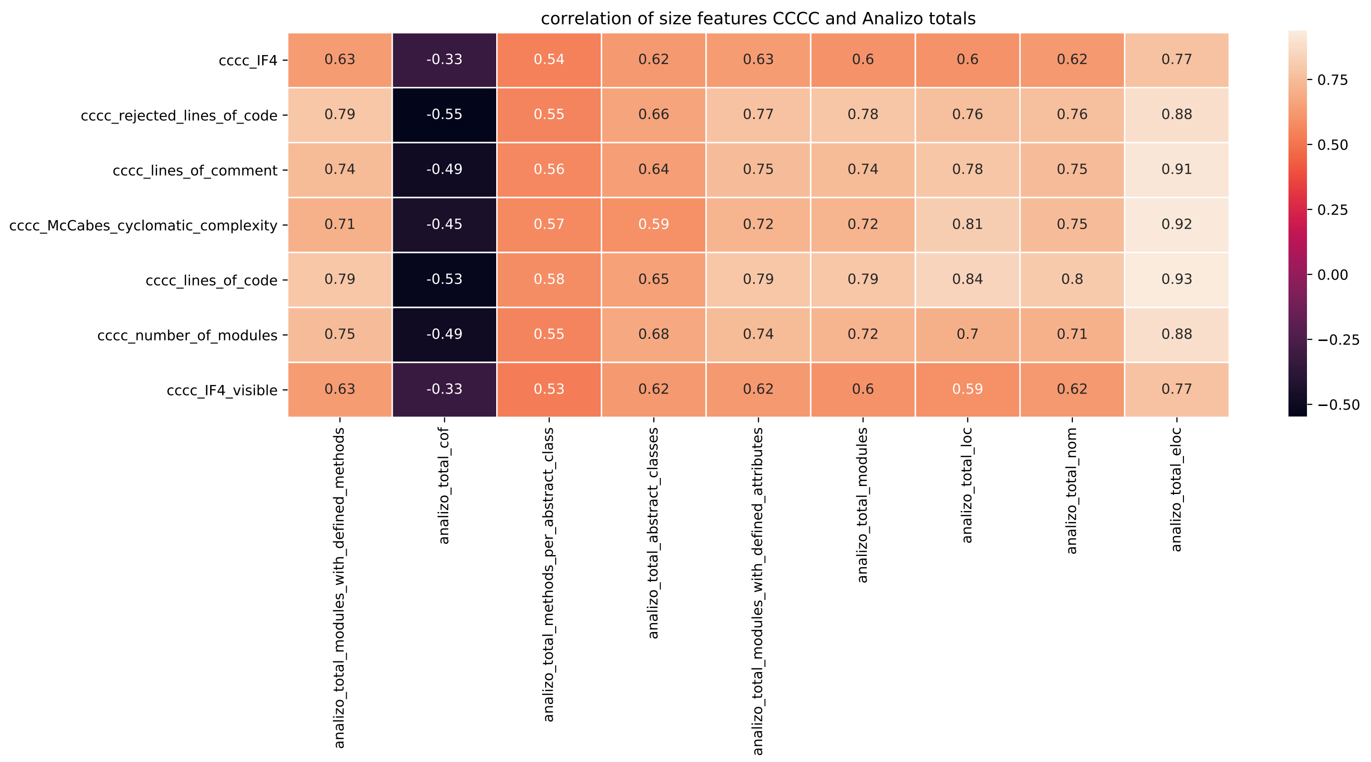


Figure 4.3: Correlation of size features CCCC VS Analizo

4.4. After fixing the projects or replacing them we validate the correlation again, In this particular example we achieve 97%, 99% (Pearson and Spearman respectively) correlation between Analizo and CCCC LOC after the correction.

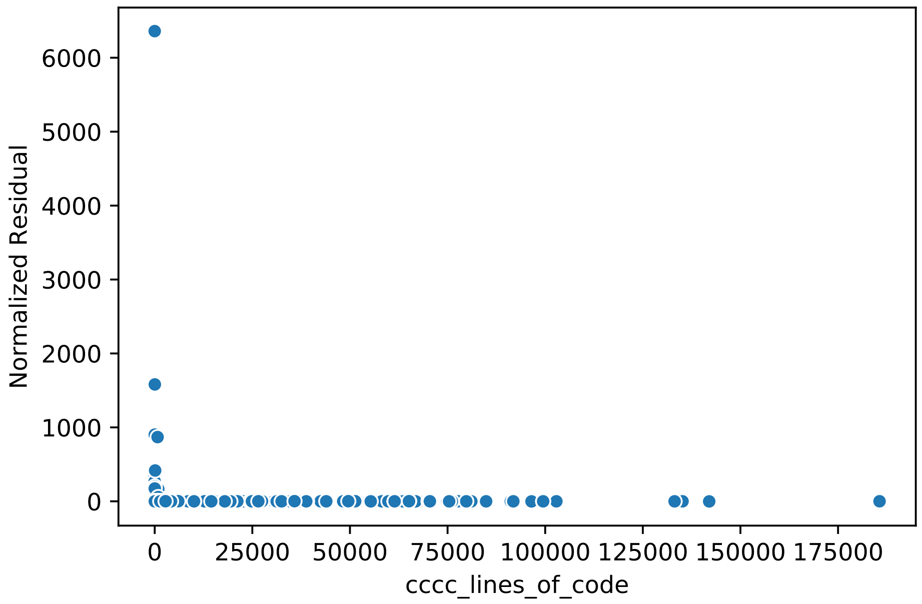


Figure 4.4: Normalized residual between lines of code metrics of CCCC and analizo

5 Is Autonomous Agents Code Unique?

We conducted two separate analysis efforts which had common general goal. 5.1 details the results of a statistical analysis, while the Section 5.2 presents the use of machine-learning analysis. The focus in both is to reveal differences, if they occur, between the different software categories, as expressed in the measurements of different metrics. The analysis was carried out both for code repositories and binary repositories.

5.1 Is Autonomous Agents code Unique? - A Statistical Analysis

5.1.1 Uni variant statistical tests

Every project is represented by approximately 250 different metrics. As such, it is difficult to find differentiating metric by hand. Our goal in this method of analysis is to find individual features that its statistics differ noticeably between different software categories. This task can be described as feature ranking based on the individual feature classification power. This task can be more practically defined as the task to identify code metrics features that can be used as good predictors for a software's category.

Formally we seek for a set of functions g such that $g = p(class|feature)$ and $g > threshold$. Note that this is a uni-variate analysis The methods that used here evaluates each feature individually and do not consider feature interactions. The uni variate methods consist of providing a score to each feature, often based on statistical tests. The scores given by the tests usually either measure the dependency between the dependent variable and the features (e.g. Chi squared , Pearson's correlation coefficient), or the difference between the distributions of the features given the class label (F-test and T-test). Table 5.1 summaries several methods for feature ranking that were examined.

Test	Assumptions	relevancy
Chi-squared	On large samples (n>30) works without normality Since the average value of a distribution converges in distribution to a Gaussian. Data should be strictly positive	
F - test (= one way anova)	For each class, the feature is normally distributed, with same variance.	Not relevant, Our data is not normal
T-test	Assumes equal variance	Different software domains has different variance in the same feature
Kolmogorov–Smirnov test	Less robust that T- test but does not require Normality	Both t-test and KS-test are used for two classes test, A propriety method developed to support multi class.
Mutual Information	Relies on the computation of the feature probability distribution, which is usually done using non- parametric methods (without major assumptions)	It is a measure of the dependency between two random variables. It intuitively measures how much knowing one of the variables reduces the uncertainty about the other

Table 5.1: Methods for feature ranking

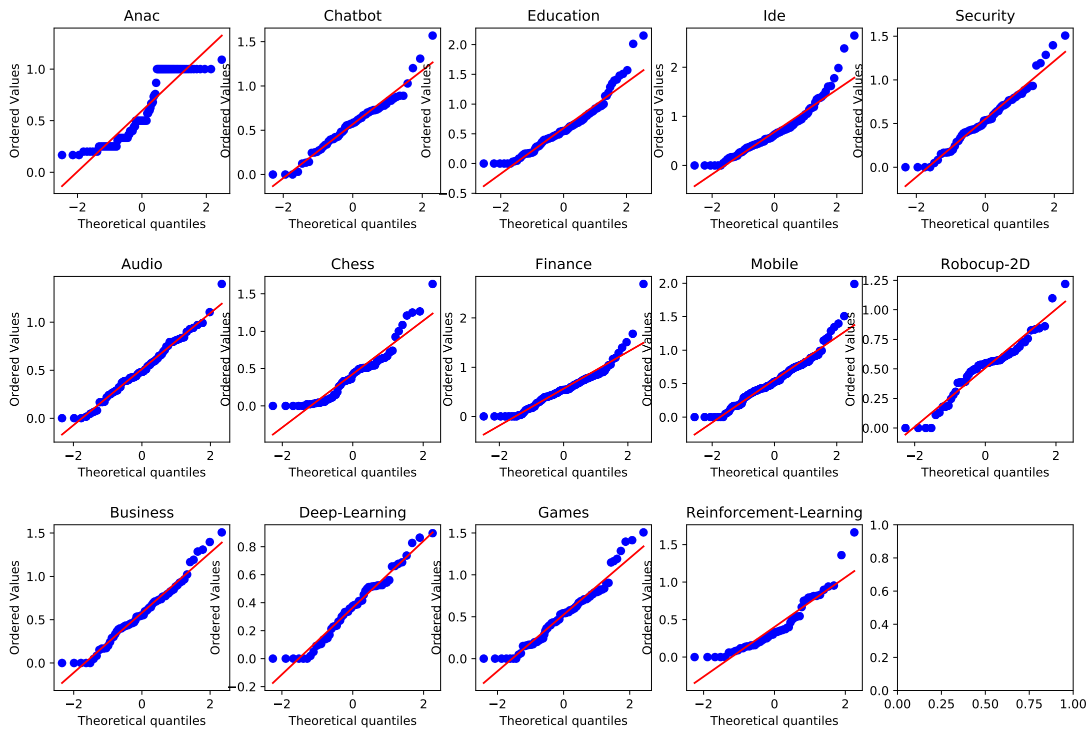


Figure 5.1: Probability plots of DIT mean dots are closely related to the line revealing close to normal behavior

5.1.2 Data distribution

First, Normality tests were conducted on the the data to test that the assumption that the distribution of the values of feature inside each software category is normally distributed. The test were applied using "normaltest" function of the scipy library which implements a methods based on skewness and kurtosis suggested in [13]. The results of the tests exposes that although for some features the tests shows nearly normal behavior, the majority of the data is not normal. Examples for close to normal and non normally distributed features are presented in Figures 5.1 and 5.2. Those finding rejects the assumption that all features are normally distributed inside the categors and suggests that methods that does not require normality should be used.

Therefore, only the two relevant statistical methods were used to find promising features. First we evaluate the feature importance by the mutual information metrics. Intuitively, mutual information means the amount of information (that is, reduction in uncer-

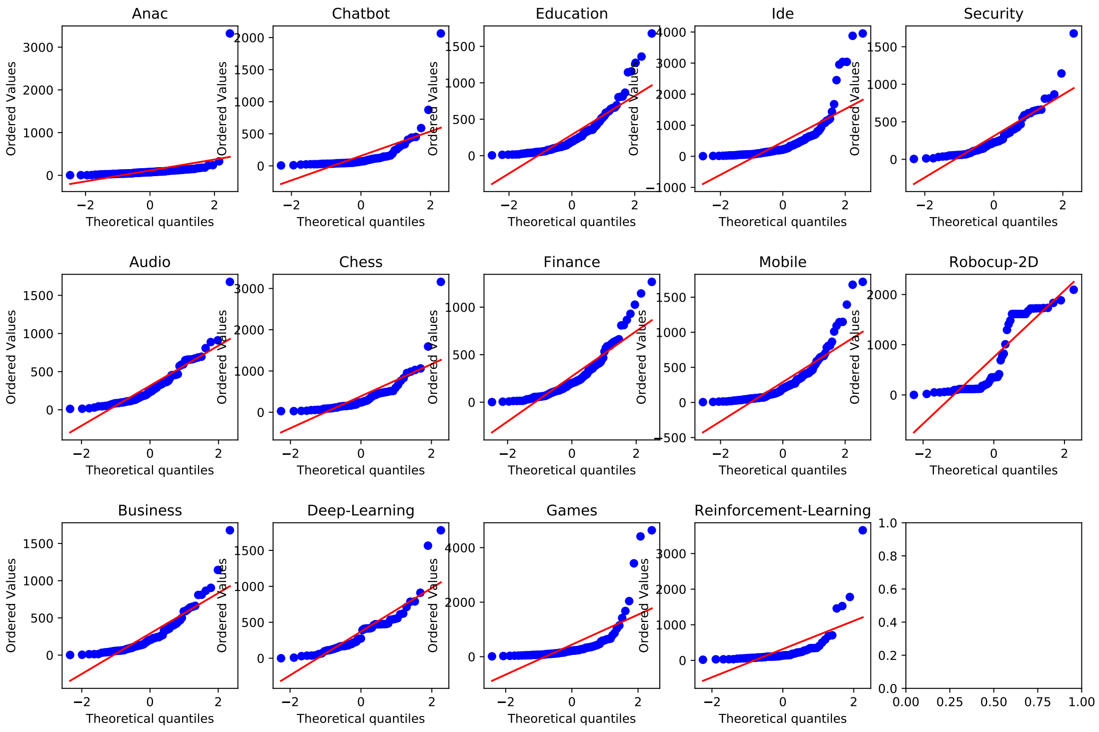


Figure 5.2: Probability plots of RFC max dots are not correlated with the normal line

	feature	mutual infomration score
0	noa_50%	0.23
1	rfc_50%	0.23
2	rfc_min	0.23
3	anpm_max	0.22
4	loc_min	0.22
5	complexity_per_module	0.21
6	rfc_25%	0.21
7	cccc_lines_of_code	0.21
8	lcom4_max	0.21
9	npm_min	0.21
10	cbo_std	0.20
11	accm_50%	0.20
12	amloc_min	0.20
13	accm_min	0.20
14	LOC_per_module	0.20
15	noc_count	0.19
16	lcom4_count	0.19
17	anpm_min	0.19
18	anpm_std	0.19
19	nom_min	0.19

Table 5.2: Mutual information top features

tainty) that knowing either variable provides about the other. In our case we are interested to identify the features that encapsulate more information about the classification to software domain of the specific sample. Practically the test was conducted using the multivariate mutual information calculation function of sklern: "mutual_info_classif" a rank list of the top 20 features is presented in Table 5.2 and will be discussed in the analysis section below.

5.1.3 Statistical tests

The investigation above revealed that some metrics statistics has information that can support software domains classification. Next, the ability to cluster several software categories based on the code metrics is investigated. The goal in this investigation is to highlight features that are not only able to segregate one class from all the others but identify the features that supports separation of several class and potentially reveal some relation between those classes This goal was achieved by running an algorithm that iterate over all features (code metrics statistics) separately and creates two one-dimensional sample distributions on each cycle: one from the repository under test and one from the union of all other repositories.

Next, the algorithm executes two samples Kolmogorov–Smirnov test to determine how likely the sample of this feature in the repository under test is taken from the corresponding feature in "other" repositories distributions. If the result (p-value) of the tests is under a certain threshold it is a added to a set containing all the repositories passed the test on the same feature. Lastly the algorithm returns all those "feature" sets that has less than or equal to 4 repositories. The logic behind that last step is that if for a certain feature too many repositories are "unique" it probably due to a noisy feature. Algorithm 1 describes the procedure. We emphasize that this is a heuristic procedure, to draw human attention to features of interest, not for statistical inference.

Algorithm 1 Common differentiator algorithm

```
1: for all $r \in Domains$ do
2:     $others \leftarrow (Domains - \{r\})$
3:     for all $f \in metrics$ do
4:         if $\textit{2-samples t-test}(r_f, others_f) < 0.05$ then
5:             $CommonSet_f \leftarrow CommonSet_f \bigcup r$
6: for all $f \in metrics$ do
7:     if $|CommonSet_f| >= 3$ then                    ▷ 3 or more clustered together?
8:         $selected_f \leftarrow CommonSet_f$
```

5.1.4 Results

Table 5.3 shows the output of the algorithm for each individual metric, when listed in increasing order of probability (15 first clusters) (i.e., in order of *decreasing* indication of separation power). The p value presented is a the mean value of all p values in the cluster. The table shows that the most frequent cluster is the cluster of [anac, Chess, Robocup-2D] which appears 8 in the from the top 15. The second frequent cluster is [Anac, Chatbot, Chess] which appears 3 times in the top 10. The features that were identified by algorithm to have high separation power: mmloc_mean, loc_mean, loc_75%, mmloc_75%, accm_mean are all related to complexity and are highly correlated. Other separating feature is the rfc which indicates high coupling.

Note that *p* value is used in Table 5.3 as a heuristic indicator for the human analyst. It gives an indication of the strength of the clustering, independent of the content of the cluster. Even if the agent domains could be distinguished from the others, we could easily expect other software domains to be so clustered. However, the fact is that the strongest distinguishing metrics put autonomous agents together, apart from other domains. This hints as to the clarity of the clustering in the data.

'most segregating feature' algorithm executed on the data. The results shows that distributions with lowest p-value are related to ACCM and 'Agent' repositories (RoboCup-2d, anac, chess) and it highlights ACCM as metrics that mostly differentiate the 'Agent' category from other software domains.

var	Repositories	p value ks	relative to mean
mmloc_mean	[Anac, Chess, Robocup-2D]	9.53E-10	above
loc_mean	[Anac, Chess, Robocup-2D]	1.92E-09	above
rfc_mean	[Anac, Chess, Robocup-2D]	6.65E-09	above
loc_75%	[Anac, Chess, Robocup-2D]	2.10E-06	above
mmloc_75%	[Anac, Chess, Robocup-2D]	1.11E-05	above
accm_mean	[Anac, Chess, Robocup-2D]	5.34E-05	above
amloc_mean	[Anac, Chess, Robocup-2D]	7.19E-05	above
cccc_McCabes_ cyclomatic_complexity	[Audio, Ide, Robocup-2D]	2.43E-04	above
anpm_mean	[Deep-Learning, RL, Robocup-2D]	2.69E-04	above
lcom4_max	[Anac, Chatbot, Chess]	3.27E-04	below
lcom4_std	[Anac, Chatbot, Chess]	7.09E-04	below
amloc_75%	[Anac, Chess, Robocup-2D]	8.22E-04	above
cccc_lines_of_code	[Audio, Ide, Robocup-2D]	1.13E-03	above
npm_max	[Audio, Deep-Learning, Ide, Robocup-2D]	1.31E-03	above
dit_max	[Anac, Chatbot, Chess]	2.25E-03	below

Table 5.3: Top distinguishing features in descending order, and the software domains they cluster.

We then moved to examining the results visually, using box-plots and bar-plots to display the distribution of specific metrics of each software domain. First we validate (randomly) that features that has overall high p value really describe the data in an inseparable manner. cbo_50% as showed 5.3 in is an example to such feature.

We seek to find features which, as clearly as possible, distinguish the three classes of domains.

Indeed some metrics clearly are different between domains. For example, Figure 5.4 show the box-plot distribution of the Lack-of-Cohesion (LCOM4) metric, which received generally low rank by the heuristic procedure (i.e., a relative high *p* value). Here, we clearly see that the *anac* group stands out, compared to the other software domains. However, it is the only domain in the cluster, and so it does not stands as a good clustering feature for our needs.

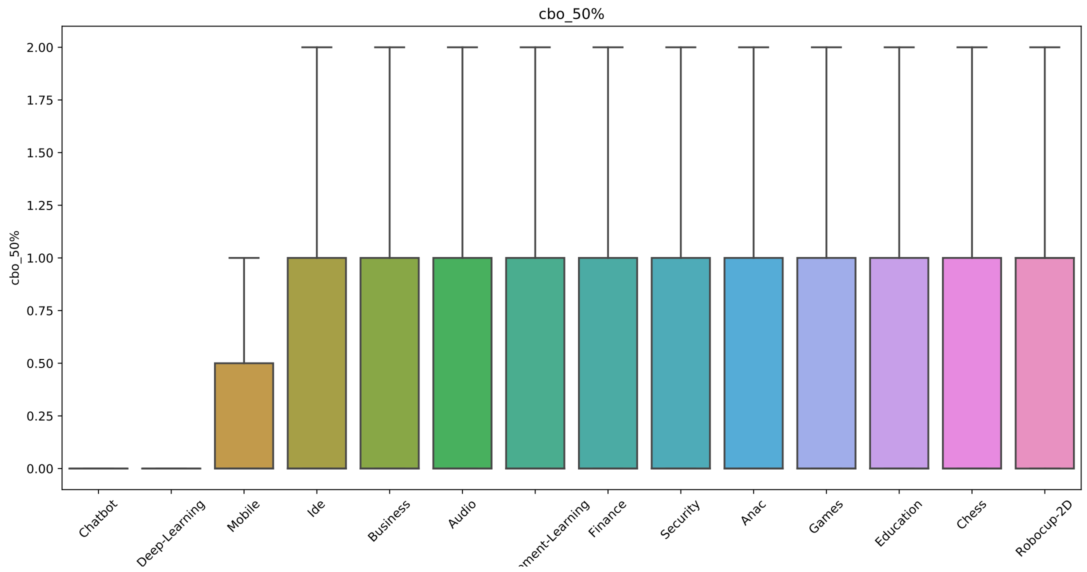


Figure 5.3: Box plot distribution of cbo 50%.

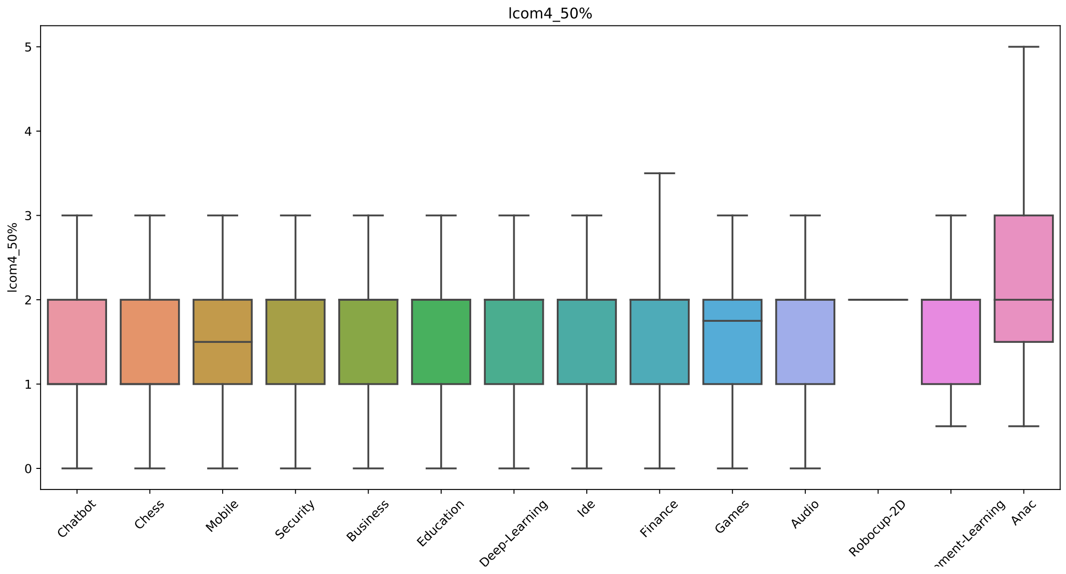


Figure 5.4: Box plot distribution of LCOM4 50%.

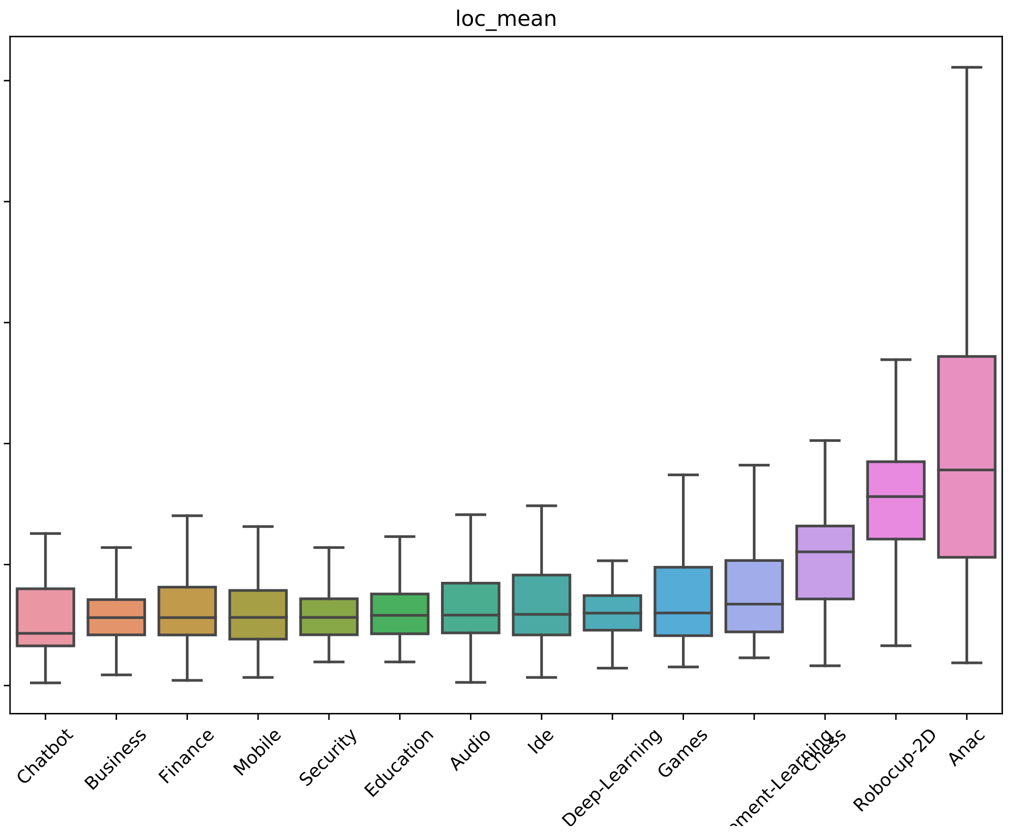


Figure 5.5: Box plot distribution of mean LOC of software domains.

In contrast, inspecting metrics that were ranked high by Alg. 1 visually figs. 5.5 to 5.10 reveals convincing differences between the groups highlighted by the algorithm. Some findings regarding the the differnt software domains stand out from the others. First, The subgroup of {anac,robocup-2d,chess} which are the "agent" categories exist in most of the clustering features. Second, and maybe more interesting is that the feature that are promising, in terms of its their ability to distinguish between agents and non-agent software are all related to complexity: loc (module line of code), mmloc (mean method line of code), accm (Average Cyclomatic Complexity per Method).

5.1.5 Interim Summary.

We defer a discussion of the *meaning* of these findings to Section 7. For now, based on the manual analysis procedure described, we only state the hypothesis that complexity related metrics (LOC, Max method LOC, ACCM) metrics are different between autonomous agents software and general software in other domains. This finding suggest inherent char-

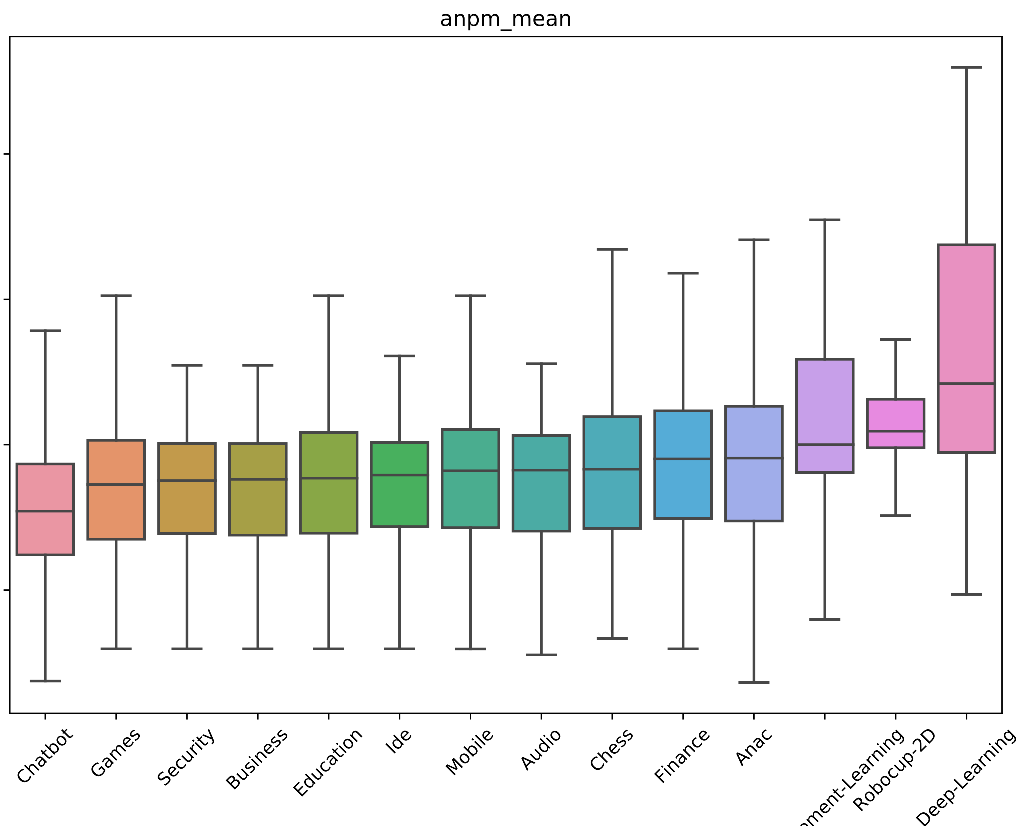


Figure 5.6: Box plot distribution of mean anpm of software domains.

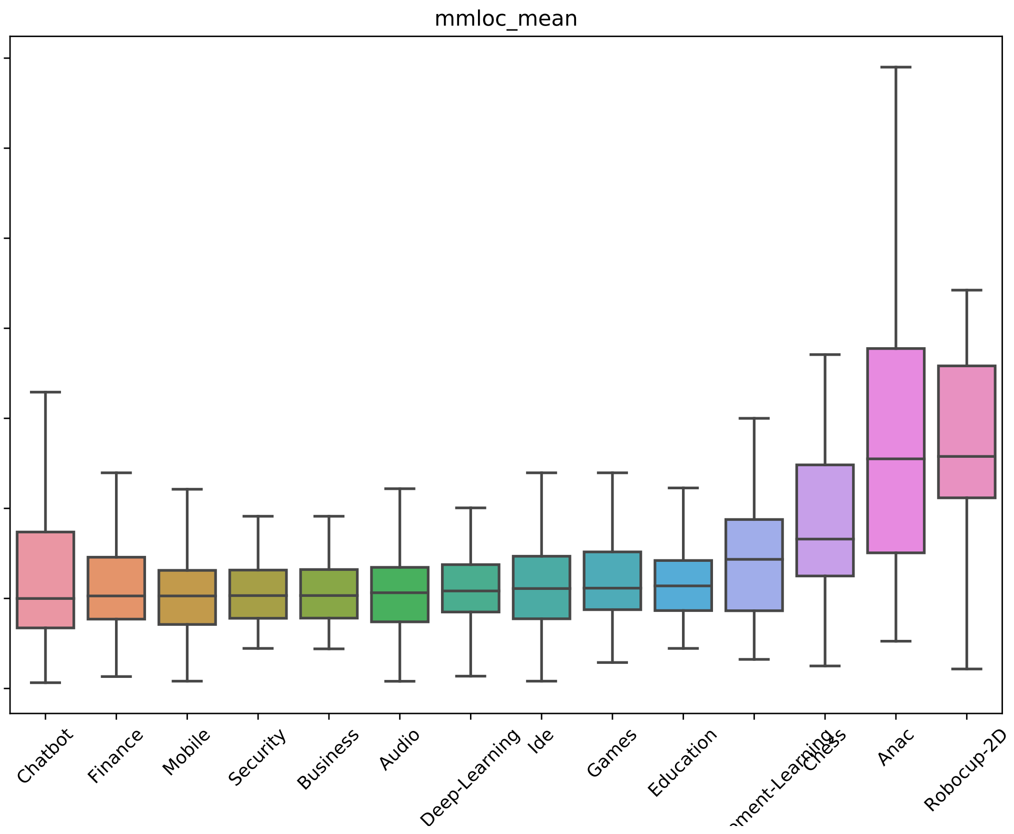


Figure 5.7: Box plot distribution of mean mmloc of software domains.

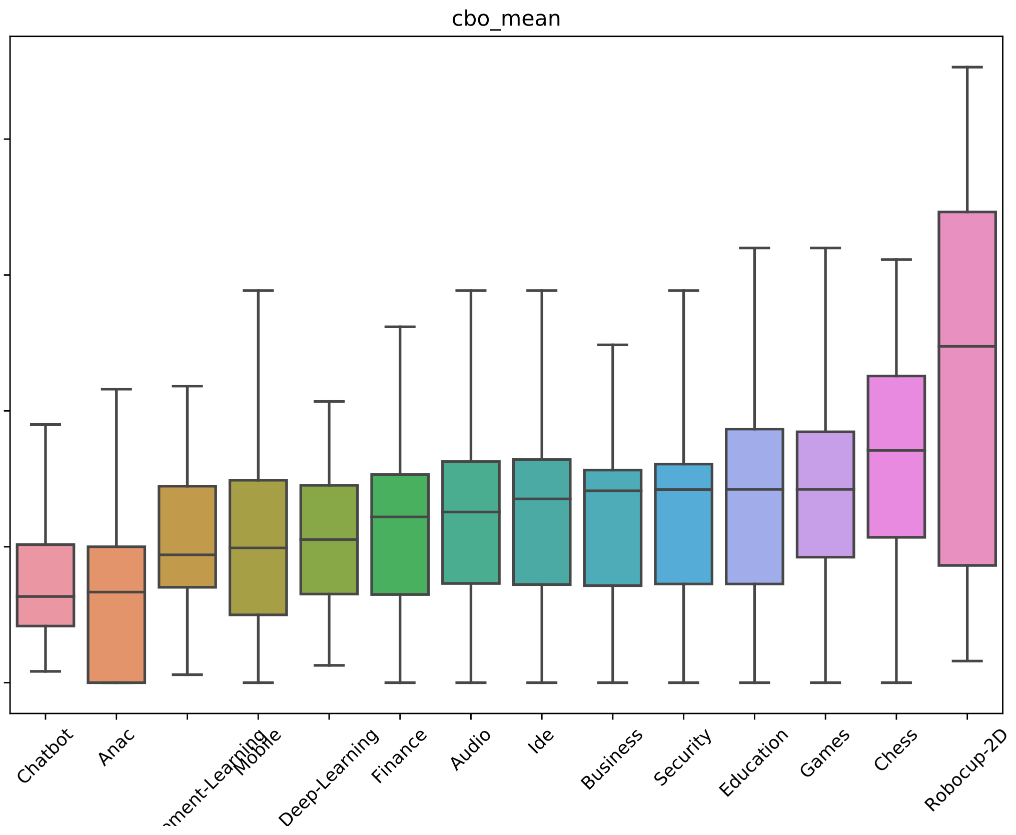


Figure 5.8: Box plot distribution of mean mmloc of software domains.

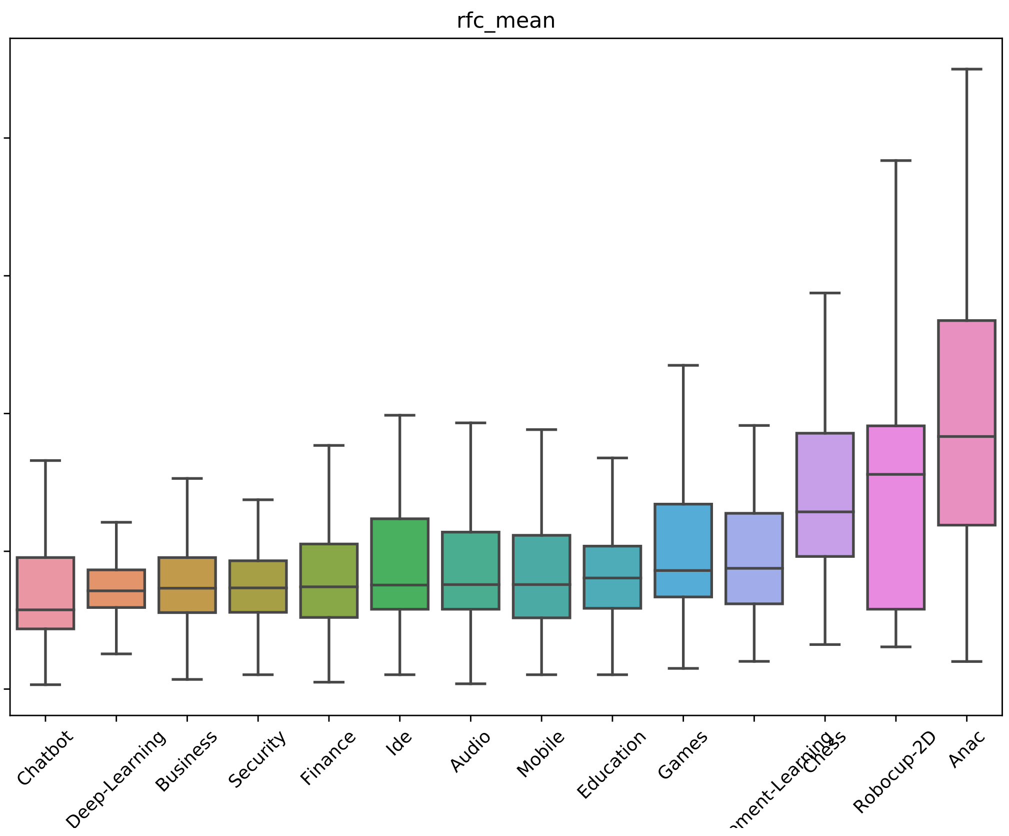


Figure 5.9: Box plot distribution of mean rfc of software domains.

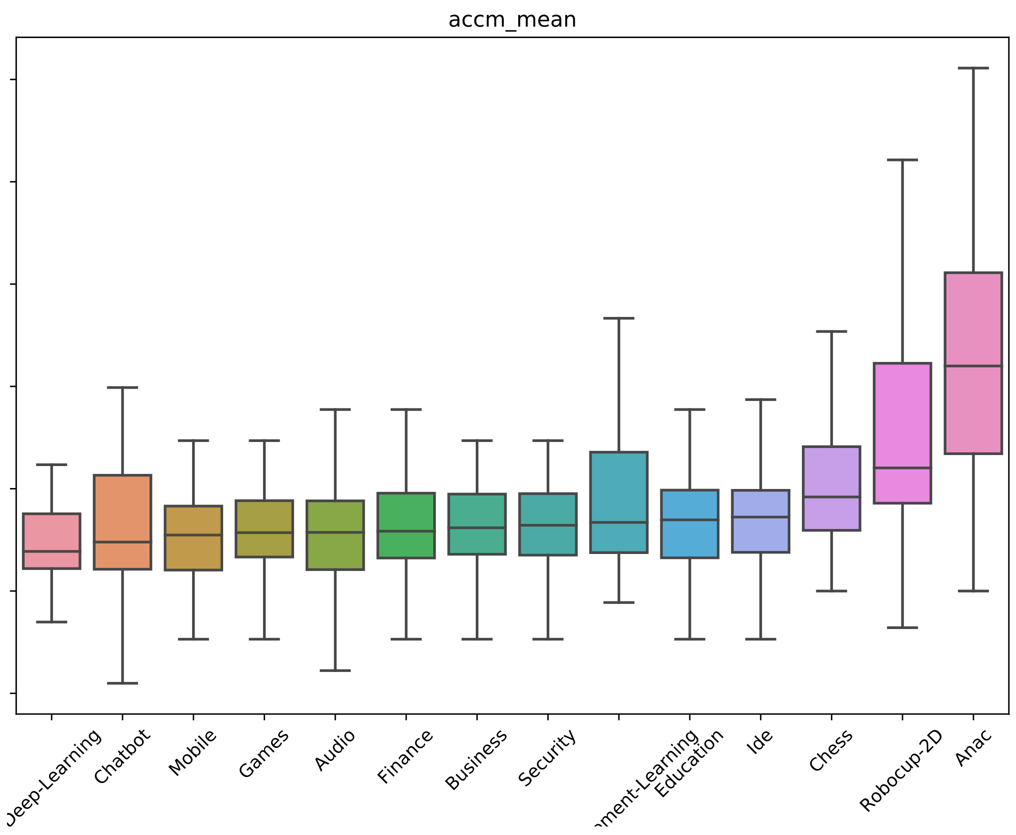


Figure 5.10: Box plot distribution of mean ACCM of software Categories.

acteristics of agent software that makes its development different and impact the way it is coded.

5.2 Is Autonomous Agents code Unique? - Machine Learning Analysis

A second approach for our investigation uses machine learning techniques, to complement the manual analysis. Humans detect patterns in visualizations that computers may miss, yet may also fall prey to misconceptions. Thus an automated analysis can complement the manual process, especially if they agree in their conclusions.

We attempted to use several different machine learning classifiers to distinguish agent and non-agent software domains, with the goal of analyzing successful classification schemes, to reveal the metrics, or metric combinations, which prove meaningful in the classification.

5.2.1 Pre-processing the data.

As the base for the ML based analysis we used the same data-set described in the former chapters while doing the following pre processing actions to adjust to automatic model training. In order to reduce the number of features (standing originally at around 400), highly correlated features (> 95 Pearson correlation) were removed. The remaining features are 104 features of the descriptive statistics of the Analizo tool and all the features from the CCCC tool. Next, all records (project repositories) that has null values in one of those 104 features were removed Finally, to make the classification valid the data set was reduced to contain categories with more than 30 projects 5.4 shows the final amount of projects in each category.

Informally the machine learning based approach tries to run multiple optimizations to that should output the best classifier for each class of software, thus a classifier that has the best score under the relevant evaluation method. If a classifier is able to separate with score above some threshold between the class it was trained on and the rest of the repositories population and suggests that this class is unique in some manner, in this chapter we describe the process and results of this process. Moreover, using techniques to explore the decisions of the model the features that were used by the model of each class are explored and used to suggest reasoning for the model's output.

5.2.2 Machine learning pipeline

We choose one vs many classification strategy, similarly to the manual analysis above. Iterating over all software classes, we trained a binary classifier to differentiate between samples of one software domain (ex. Audio) to all other software classes. In order to adjust our data to to binary classification problem we created a dataset for each category

Category	Projects
Mobile	114
Anac	105
Education	94
Ide	93
Finance	83
Games	79
Audio	64
Business	58
Deep-Learning	57
Robocup-2D	57
Chess	54
Security	53
Reinforcement-Learning	53
Chatbot	53
Development	32
Graphics	31

Table 5.4: Number of projects in each category in the final data-set

such that the input is all the code metrics features selected in the pre-process phase and the target variable (class) was set to be 1 for the underline category and 0 for the all the projects in the other categories. The data in each category data-set was divided into a training (85%) and testing (15%) sets.

For classification, we used the following classification algorithms: *Logistics Regression*, and *Gradient-Boosted Decision Trees*. The implementations are open-source packages (scikit-learn[1] and XGBoost[2]). The performance of classifiers was carried out using two scoring functions, familiar to machine learning practitioners: F1 and AUC (area under the ROC curve). In both, a greater value indicates better performance. Each of the tables below (Tables 5.5–5.6) shows the top classifiers built using the classification algorithms. In each, we list the top classification results of a single domain versus all others. Our interest, however, is not so much on being able to classify a specific domain, but instead in the metrics used as features when classifying Agent software. The last column of each table lists the most informative 3–4 features (metrics) used by the classifier. Frequent recurrence may hint at important metrics.

F1 is a popular scoring function used in many supervised learning tasks. F1 score can

[1] https://scikit-learn.org/
[2] https://github.com/dmlc/xgboost

be interpreted as a weighted average of the precision and recall. the following are formal definitions of those evaluation functions:

$$\begin{aligned} &Accuracy = \frac{TP+TF}{TP+TF+FP+FN} \\ &Precision = \frac{TP}{TP+FP} \\ &Recall = \frac{TP}{TP+FN} \\ &F1 = \frac{2*Precision*Recall}{Precision+Recall} = \frac{2*TP}{2*TP+FP+FN} \end{aligned} \tag{5.1}$$

The second scoring function is the area under curve of the receiver operating characteristic curve, i.e., ROC curve. The latest is a graphical plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied. The auc curve is created based on two evaluation methods:

$$\begin{aligned} &Sensitivity = Recall = \frac{TP}{TP+FN} \\ &Specificity = \frac{TN}{FP+TN} \end{aligned} \tag{5.2}$$

ROC curve is produced by calculating the values of those functions at different decision threshold values ($0 \geq T \leq 1$) and piloting the curve connecting between those values on a 2dim space. Calculating the area under this curve Calculating the area under the ROC curve returns a score that expresses the robustness of the classifier and predicates its performance in future predictions. High ROC curve AUC scores predicts high performance of the classifier.

model parameters all the LR classifier build for this task are using lib-linear optimizer, l2 regularization and stopping criteria of converge or 100 iterations. Next, we used Gradient-Boosted Decision Tree classifiers. The idea in this technique is to use an ensemble of decision trees based on subsets of the samples and features, to lower the risk of over-fitting while maintaining high accuracy. The classifiers were built using the XGBoost package, using the default parameters.

5.2.3 Results

The top performing LR classifiers are reported in Table 5.5. In general their scores are lower than the XGBoost 5.6 reported blow.

Figure 5.11 shows the ROC curves for the XGBoost classifier described above. The ROC curves of the top performers match our understanding of their efficacy.

	Category	AUC	F1
0	Anac	0.97 ± 0.02	0.92 ± 0.05
1	Reinforcement-Learning	0.82 ± 0.02	0.36 ± 0.15
2	Deep-Learning	0.81 ± 0.05	0.39 ± 0.10
3	Robocup-2D	0.80 ± 0.15	0.49 ± 0.29
4	Chatbot	0.78 ± 0.03	0.17 ± 0.01
5	Chess	0.70 ± 0.08	0.22 ± 0.05
6	Audio	0.58 ± 0.04	0.15 ± 0.04
7	Mobile	0.49 ± 0.01	0.15 ± 0.02
8	Graphics	0.45 ± 0.04	0.03 ± 0.00
9	Development	0.45 ± 0.08	0.08 ± 0.05
10	Games	0.43 ± 0.03	0.10 ± 0.03
11	Ide	0.41 ± 0.04	0.09 ± 0.01
12	Education	0.40 ± 0.05	0.10 ± 0.03
13	Business	0.39 ± 0.06	0.05 ± 0.01
14	Finance	0.35 ± 0.03	0.07 ± 0.01
15	Security	0.35 ± 0.04	0.05 ± 0.00

Table 5.5: Performance indicators for one to many classification using Logistic Regression

5.3 Feature importance

Finding the a classifiers that separates agent based software from other types of software is not enough for hypothesizing about the nature of the difference, for that a deep examination of the feature used by the models to make their classification is required. Feature importance analysis is the task of evaluating how the features in a model contribute to prediction. The specific evaluation method is model depended and there are several whys to quantify this impact.

	Category	AUC	F1
0	Anac	0.98 ± 0.02	0.94 ± 0.05
1	Robocup-2D	0.96 ± 0.05	0.74 ± 0.25
2	Chatbot	0.87 ± 0.05	0.33 ± 0.10
3	Chess	0.86 ± 0.06	0.49 ± 0.15
4	Deep-Learning	0.80 ± 0.04	0.45 ± 0.13
5	Reinforcement-Learning	0.80 ± 0.07	0.34 ± 0.16
6	Development	0.68 ± 0.10	0.09 ± 0.01
7	Audio	0.53 ± 0.03	0.11 ± 0.04
8	Graphics	0.49 ± 0.11	0.04 ± 0.01
9	Mobile	0.41 ± 0.02	0.12 ± 0.00
10	Games	0.40 ± 0.06	0.11 ± 0.04
11	Ide	0.36 ± 0.02	0.11 ± 0.03
12	Education	0.35 ± 0.02	0.10 ± 0.02
13	Finance	0.33 ± 0.03	0.08 ± 0.02
14	Business	0.28 ± 0.01	0.05 ± 0.00
15	Security	0.27 ± 0.03	0.04 ± 0.00

Table 5.6: Performance indicators for one to many classification using Gradient Boosting Trees

5.3.1 Logistic regression

Logistic regression classifier are optimize to minimize the following loss function:

$$\begin{array}{l} J(\beta) = 1m\sum_{i=1}^{m} log(1 - e^{-y_i}(w^T x_i + b)) \\ y_i\text{: Label of point } i \in \{-1, 1\} \\ y_i^*\text{: Model prediction } w^T x_i + b \\ w\text{: Weight vector} \\ x_i\text{: input vector} \\ b\text{: intercept} \end{array} \quad l \tag{5.3}$$

When the input vector x_i is normalized to positive values (e.g. $0,1$) large positive values of x_i will push the model toward positive prediction and large negative values will do the opposite. Therefore the importance of a feature for the classification of each of the labels is evaluated naturally based on relevant entry in the wights vector.

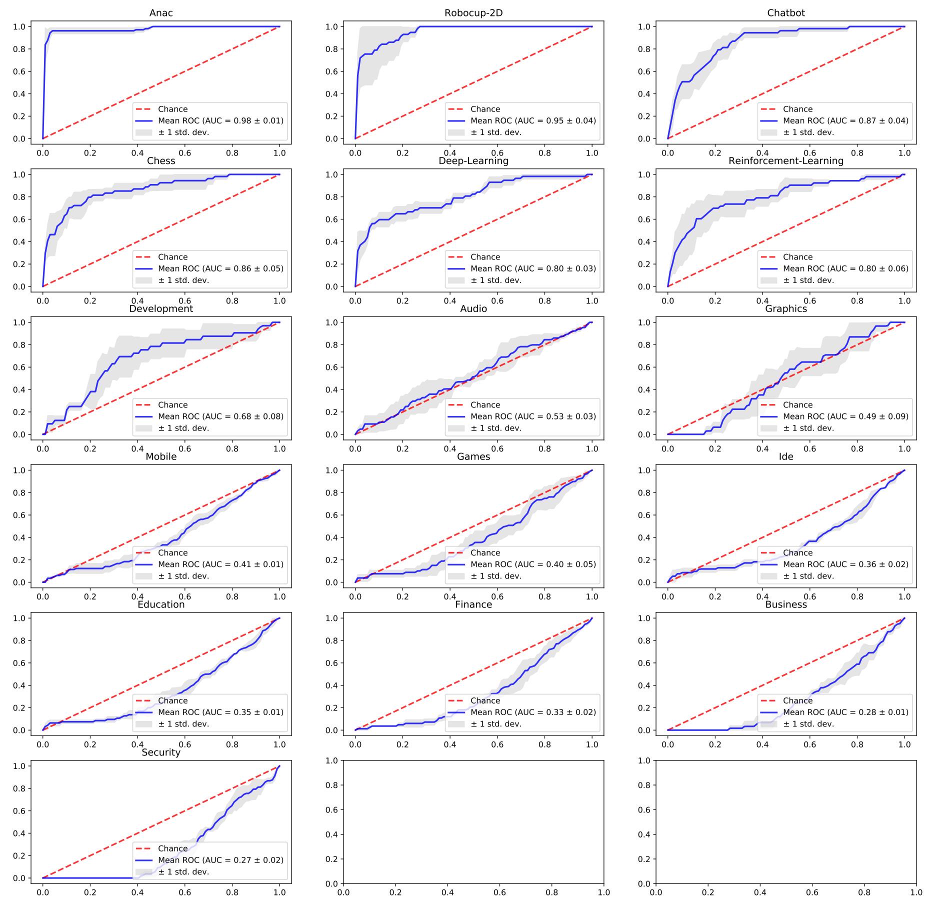


Figure 5.11: ROC plots of Classification of Boosted Decision Trees classifiers.

5.3.2 Boosted trees

Evaluating feature importance for tree based and more specifically extreme boosted classification trees based classifiers is more complicated and and there are few ways to evaluate the importance of each feature. In this task a comparison between different models in different tasks (1 vs many classification for each of the software categories) is required. Therefore the evaluation is done based on the 'Gain' feature added to classification at each tree. 'Gain' is the improvement in accuracy brought by a feature to the branches it is on. The idea is that before adding a new split on a feature X to the branch there was some wrongly classified elements, after adding the split on this feature, there are two new branches, and each of these branch is more accurate (one branch saying if your observation is on this branch then it should be classified as 1, and the other branch saying the exact opposite). The Gain in each specific tree is then averaged between all decision trees in the ensemble to form a single score for each classifier. The Importance of each feature in each classifier for both XGBoost in LR are presented in figs. 5.12 to 5.14. For messuring the important of each feature for all the agents software, each occurrences of a feature in the top 5 important feature in each single classifier is counted. Features with the highest number of occurrences are listed in 5.8, 5.7 The ranks column indicates the number of occurrences of a feature in the top 5 important features of the different classifiers

Feature	Occurrences	Categories
cccc_LOC_per_module	3	[Anac, Chatbot, Robocup-2D]
cccc_CC_per_module	3	[Chatbot, Reinforcement-Learning, Robocup-2D]
anpm_std	2	[Chatbot, Deep-Learning]
lcom4_max	2	[Chatbot, Chess]
mmloc_75%	2	[Deep-Learning, Robocup-2D]

Table 5.7: Features with highest number of occurrences in the top ranked features based on the feature rank analysis of the XGBoost based classifiers

Feature	Occurrences	Categories
noa_50%	3	[Anac, Chatbot, Development]
cccc_number_of_modules	2	[Reinforcement-Learning, Robocup-2D]
sc_75%	2	[Chatbot, Chess]
loc_mean	2	[Anac, Robocup-2D]
noa_75%	2	[Chatbot, Development]
npm_75%	2	[Deep-Learning, Robocup-2D]

Table 5.8: Features with highest number of occurrences in the top ranked features based on the feature rank analysis of the Logistic Regression based classifiers

based on the feature importance analysis results on the top performing classifiers which are those that are able to distinguish agent software from other types of software we can highlight the following: for XGBoost classifier the features with the highest predictive power which are most common between the classifiers are those feature that represent

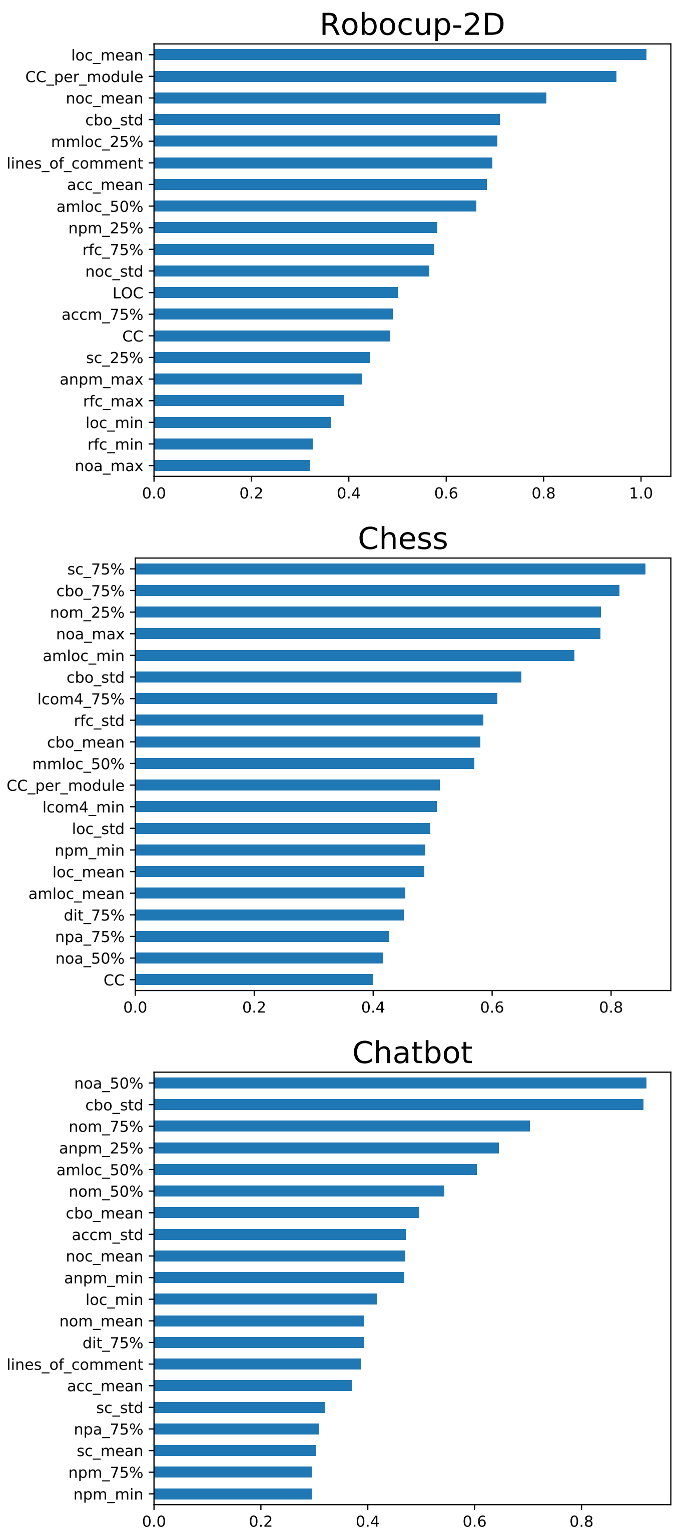
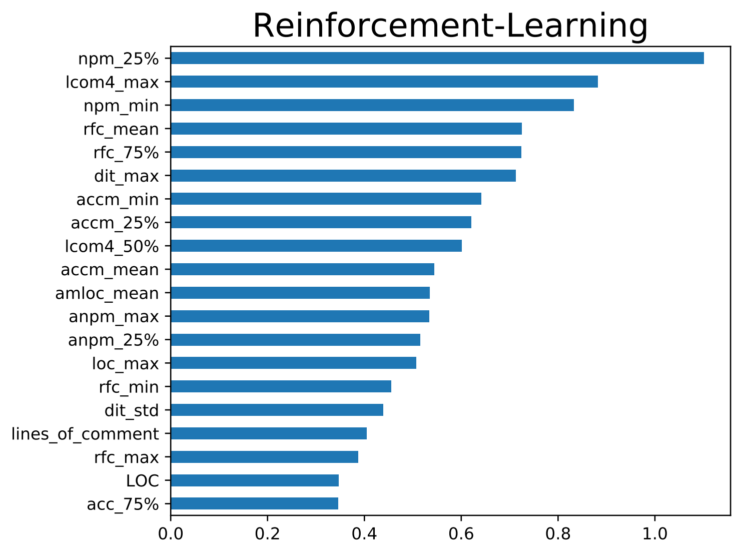


Figure 5.12: Feature Importance - Logistic Regression



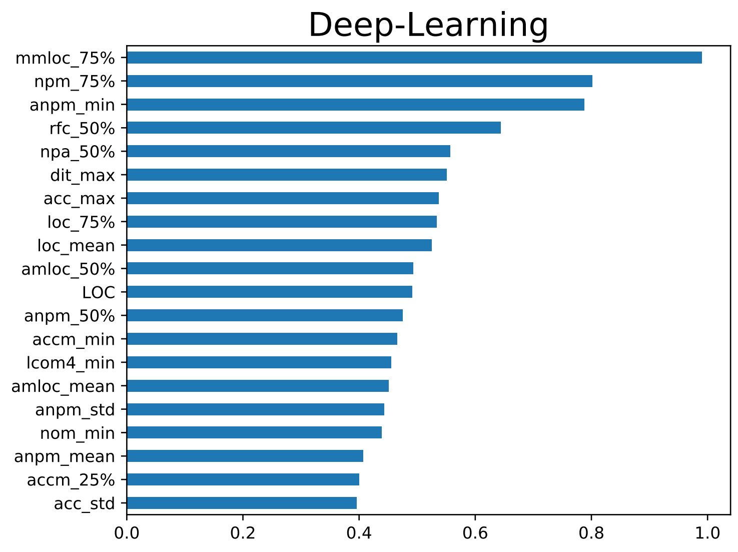


Figure 5.13: Feature Importance - Logistic Regression

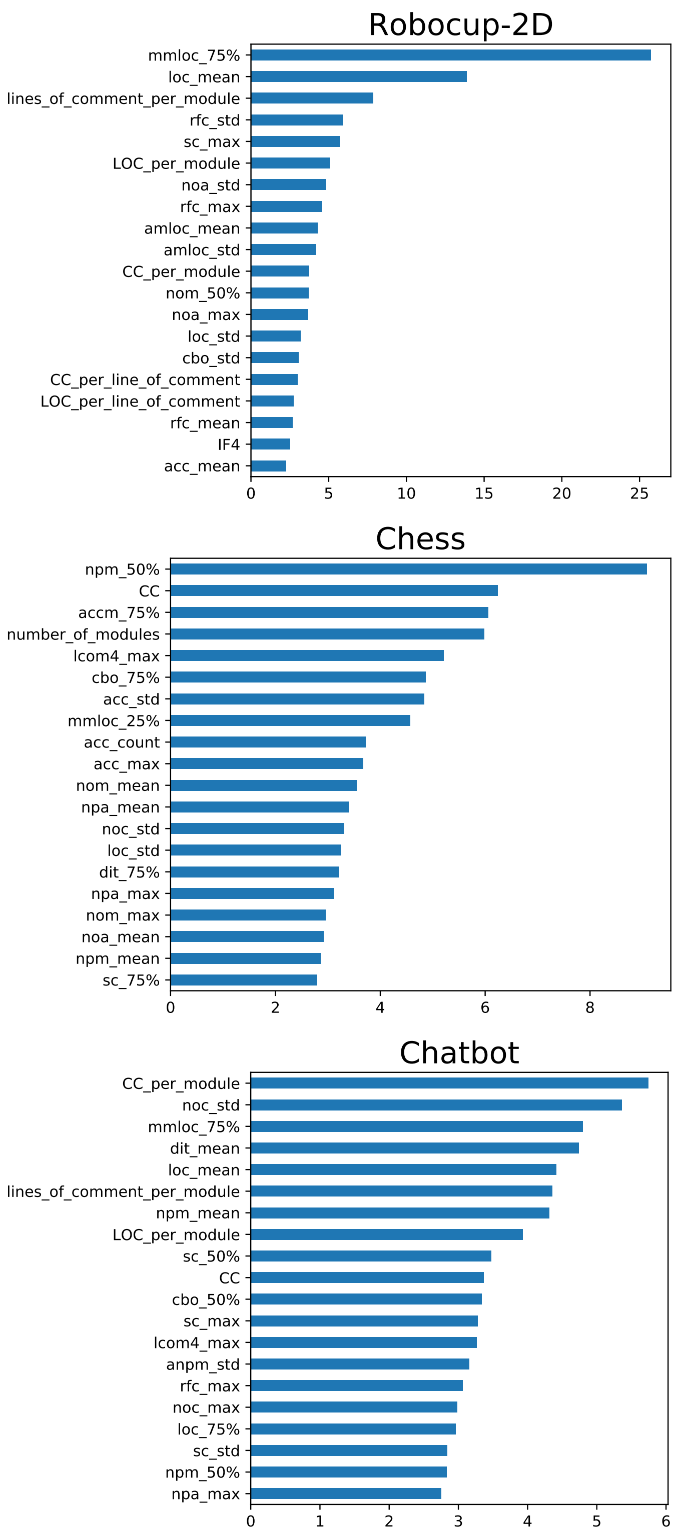
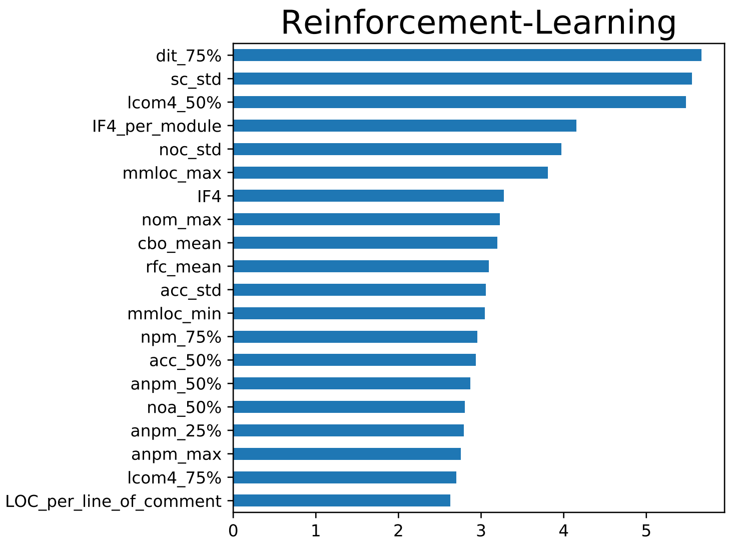


Figure 5.14: Feature Importance - XGBoost



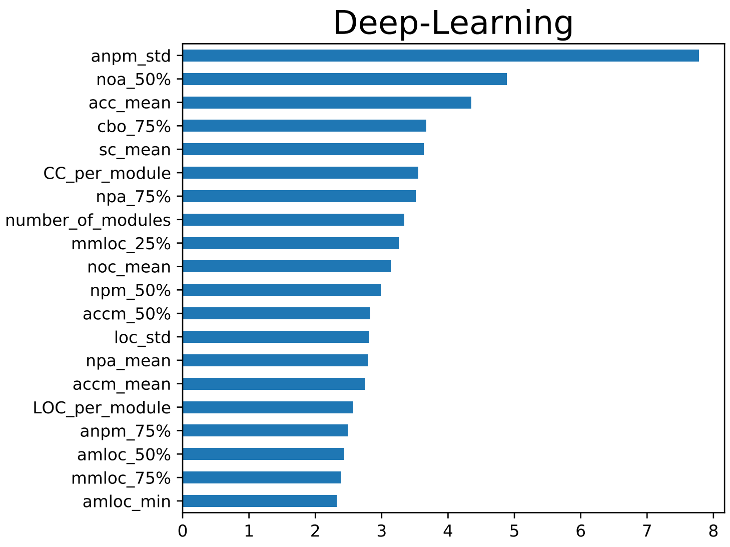


Figure 5.15: Feature Importance - XGBoost

the complexity of code in its wide sense. The "lines of code per module" and "McCabes cyclomatic complexity per module" and also "mean method line of code" suggest that that there is difference between agent / AI modules to other types of software in the amount of code requires to implement functionality and also by the amount of different trajectories this code handles. In Logic regression classifies however, although it also separates AI software better than other types of software, the results of the feature importance analysis are much less descriptive. Looking at the feature importance of the individual charts (5.8 does highlight complexity related features (Anac - mean accm of method, Robocup - line of code and CC per module, Deep learing - Mean method loc) but the commonality of such features is not as clear as in the XGboost classifiers.

5.3.3 Agent software classifier

Based on the above results that suggest unique behavior of agent software we planned another experiment to evaluate the difference between agent code form general software. The idea of this experiment is to build one classifiers to distinguish between agent and non-agent software. Hereby the experiment description: we trained GB Trees classifiers on our basic data-set setting the target variable to be 1 if the project is from an agent category and 0 otherwise. The training was done with 3 folds of 80:20. On each fold the test data was evaluated to get overall performance of classification of Agent software against other software type and also the deferential performance with regards to each specific category. The second set of experiments in the "general autonomous agent" training attempt is to done on repositories taken from gitHub alone putting aside the Anac and Robocup-2D repositories. This configuration although lacks many of the repositories representing pure AI code has the advantage of removing the variability in other parameters such as contest vs no contest code and others and keeping the AI vs non AI as the only difference between the classes of code. The results of this experiment although has slightly worse performance the average performance of XGBoost classifiers train with both experiment configuration are presented in section 5.3.3. The performance of classifying agent software in each category is presented in fig. 5.16 and fig. 5.17.

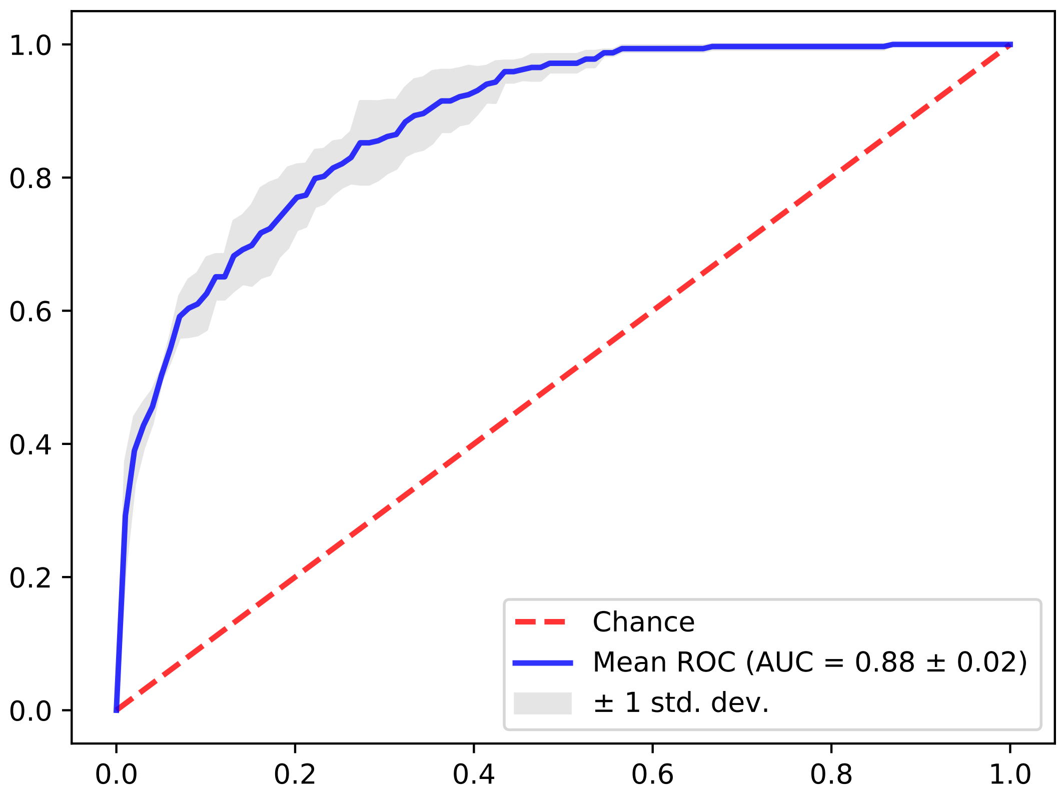


Figure 5.16: Average AUC curve and with error boundaries of classifying Agent against non-Agent software repositories

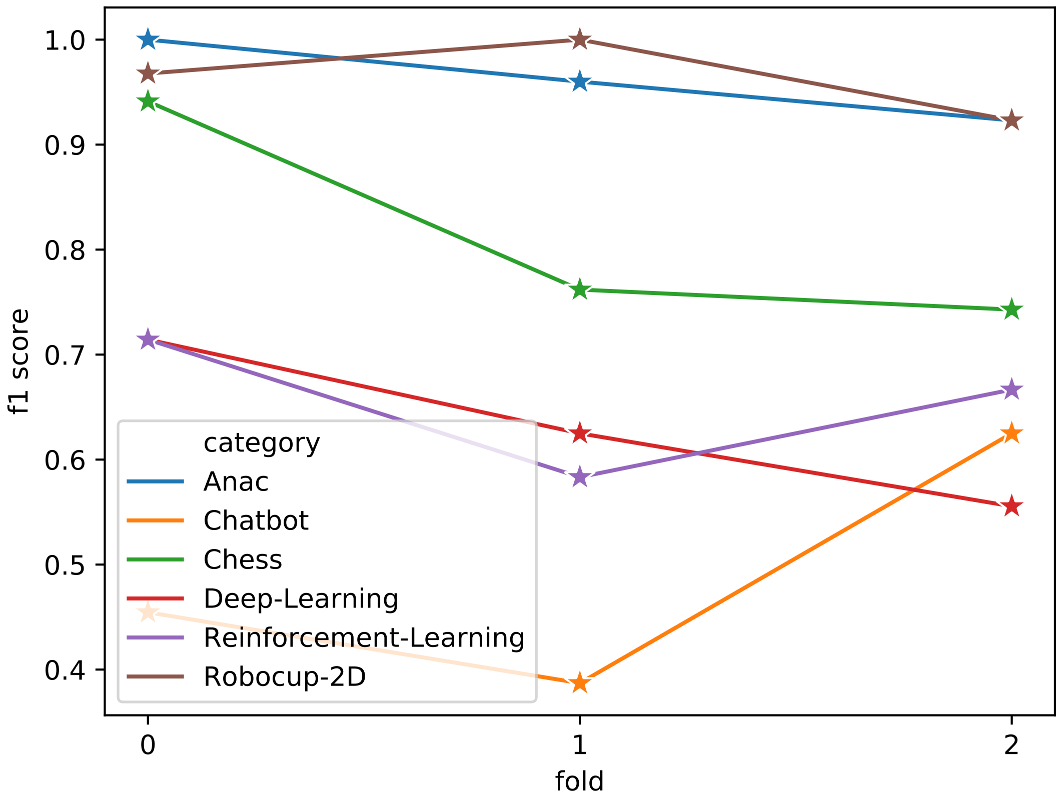


Figure 5.17: F1 scores of predicting agent vs non-agent on each of the agent categories

	Configuration	AUC	F1
0	All categories	0.88 ± 0.02	0.76 ± 0.02
0	GutHub Only	0.8 ± 0.02	0.66 ± 0.02

The results clearly suggests the model is able to generalize Agent software characteristics and apply it on the classification task. Note that although the "GitHub only" setting has lower performance compared to the "All repositories" one, The results of this experiment are even more distinct since it eliminate the inherited uniqueness of the two competition repositories type - Anac, Robocup.

Lastly, we pressed the distinction thesis even further. 5 different classifiers were trained, each keeping aside another subset of projects of the "agent" categories for validation. Also as negative examples we used another data set consists of projects select randomly from github under the same criteria described above. At each cycle of train-validation we trained one classifier and validate its separation power on the validation set that as explained contains the one "agent" category it did not train on and 500 other general software project. The results shown in 5.9 present the different classifiers. Inspecting the results compared to the results of the 1 to many classification might suggest that although each category of the "agent" type has unique characteristics that enables it to be separated from other software, still the training process was able to generalize some characteristics to create an "agent" detector classifier to separate "unseen" agent repositories from general ode repositories

	Category	AUC	F1
0	Anac	0.79	0.37
1	Robocup-2D	0.69	0.33
2	Reinforcement-Learning	0.58	0.27
3	Chess	0.56	0.24
4	Deep-Learning	0.46	0.16
5	Chatbot	0.38	0.21

Table 5.9: Performance indicators for binary classification on "held out" category by classifiers trained on other categories

6 AI Code Metrics as Performance Predictors

The second research question tries to evaluate a different possible usage of AI code metrics: are general code metrics applicable for evaluating AI performance?. For general software it was shown that some code metrics has a direct effect of the quality of a software in general terms like bugs per line of code and maintenance effort. In our research we are more in a specific are of software: AI agents code,Thus, we seek for a relation between code attribute to quality metrics that are unique to this area. As explained in the previews sections our main data source is a set of repositories with programs developed to compete in the Robucup challenge over the year, The same data source also contains logs of the different matches over the years. Each log fie contains an audit log of each action of each player participates of a single match in the RoboCup tournament. Using those logs we were able to pull information about some quality evaluates (number of win, number of goals etc...) for each of the teams. This chapter describes our attempts to find correlation between the code metrics of each team to its quality measures and thus to support the assumption that code metrics holds information that can affect the q"quality" of AI program.

6.1 data preparation

For the internal question research we needed two types of data-sets: 1. Code metrics features data-set based on the same data-set used in the external question research. 2. Quality measures of the differed matches. This part of the dataset, following process was executed: crawl to download logs files segregated by competition and year. a regular expression based extractor to extract the names of the groups from the file names and a final text processor to extract the number of goals each teams scored in the respective match. Note that there might be other quality indicator besides the goals like interaction between

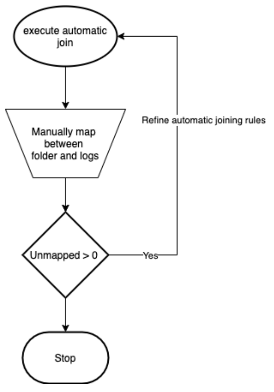

execute automatic join
Manually map between folder and logs
Refine automatic joining rules
Unmapped > 0
Yes
Stop

Figure 6.1: Joining process flow

agents (ball passe) but we focused on the most intuitive and definitive KPI. The second part of the data preparations is joining between the two data-sets described above. Generally the mapping is based on the year, competition name and team name. But since the team names in logs and the folder which contains the code for the program of the team are not synced, there are many misalignment between the two which enforce a more complicated joining process. The iterative process is based on a set of regular expression joining rules (lower, containment etc.) and manually mapped names for cases the automatic rules do not apply. The process is described in fig. 6.1

Performance indicators each row ('x') in the below script represent a summery for one group for a competition in a specific year, i.e. oxsy in robocup 2016. The raw results extracted from each log are:

- games - number of games played by the subject team
- win - number of games resulted in win for subject team
- loss - number of games resulted in loss for subject team
- tie - number of games resulted in tie for subject team
- S - number of goals scored by the subject team
- R - number of goals received by the subject team

based on the above raw values we defined four numeric performance indicators:

```
win diff = (wins - losses) / number of matches
win rate = wins / number of matches
loss rate = losses / number of matches
goals diff = (goals scored - goals received) / number of matches
```

Data validations In order to cross-validate the correctness of the extraction process another data-set is used. This dataset is an extraction of the top ranked teams from each year of the main RoboCup tournament. This validation process is designed with the assumption that team ranked high in the tournament should also have higher scores in the lower resolution performance indicators. i.e a team that has reached the second place in thew 2002 tournament has higher wins rate than a team reached the 20Th place in the same tournament. fig. 6.2 Demonstrate the validation made using the cross validation between the two sources.

6.2 Regression

First attempt to expose the relation between code behavior and AI performance is done by training a regression model to fit a regression function between the code metrics and the KPI calculated from the logs. since code metrics data set contains above 200 dimensions (code metrics and their statistics) and in this internal task we also extracted 11 highly correlated target variables. In order to make performance evaluation feasible a reduction of the two has to be done. although during the research we made a full analysis of several KPIs, for the sake of clarity the analysis in the following sections is based on the win_diff KPI which is highly correlated with some other KPIs and also logically represented a robust evaluator as it is normalized by number of the games a team played during the underlined tournament The straight forward approach to examine the Pearson correlation of the different metrics against the target variable and select those with the highest values as they are must "promising" with regards to their prediction power.

A sample of the features and corresponding Pearson correlation values with regards to the win_diff KPI is presented in table 6.1. The outcome of the correlation analysis shows that there is no single feature that can be used for prediction of the team performance, the model that will be used Eventually there are only 44 teams that has both source code and which we could match logs to. This amount of sample leave us with about 10% of the group in each year. Such a sparse data prevents us from fitting a model the correlates some attributes of the team to it performance In the following section we describe an alternative way to use the metrics for fitting a binary model which eliminate the sparseness effect

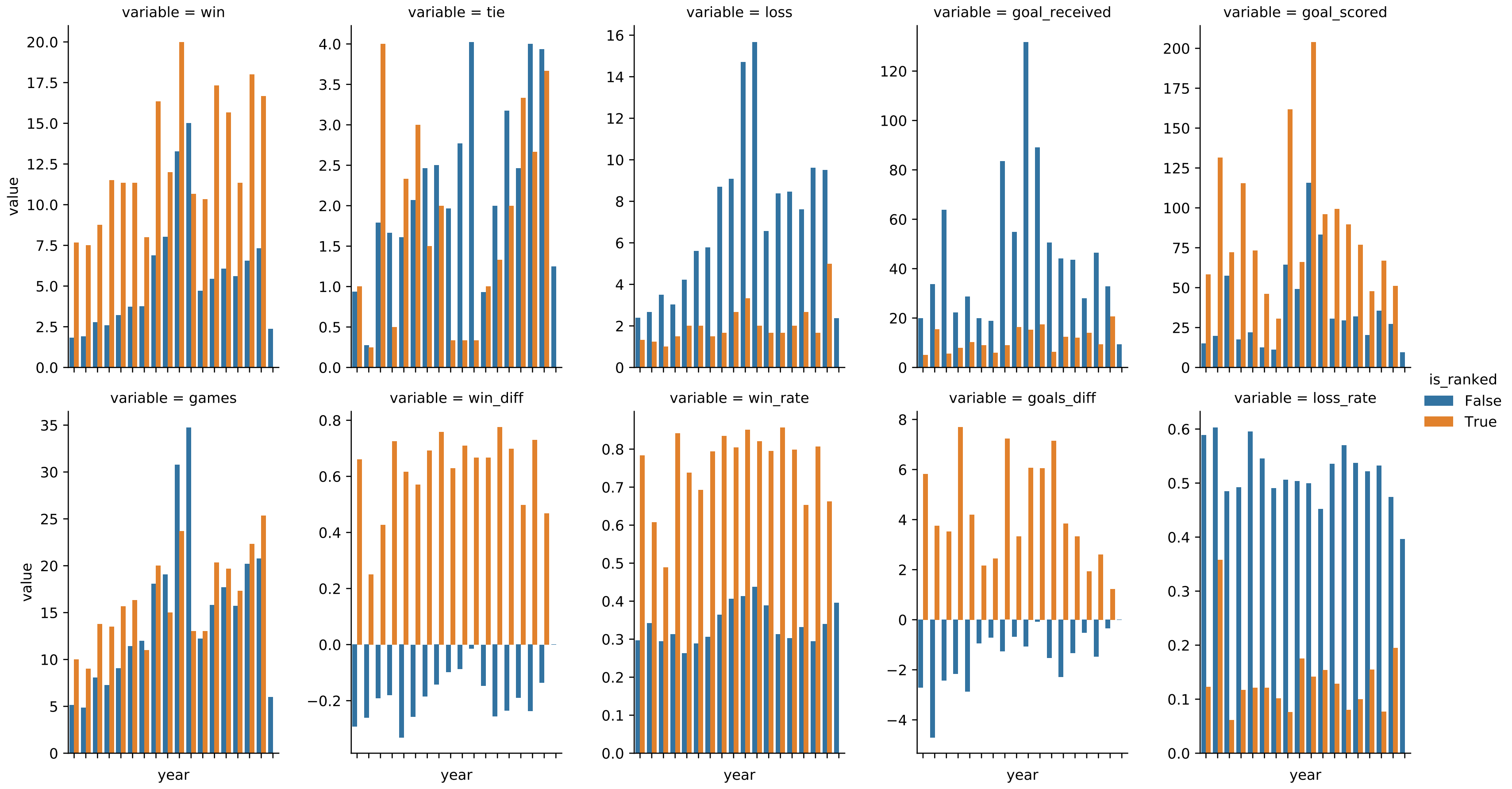


Figure 6.2: Team performance indicators VS Ranks - Clear advantage of the ‘Ranked‘ teams, teams that made it to the semi finals, over the teams that are not ranked

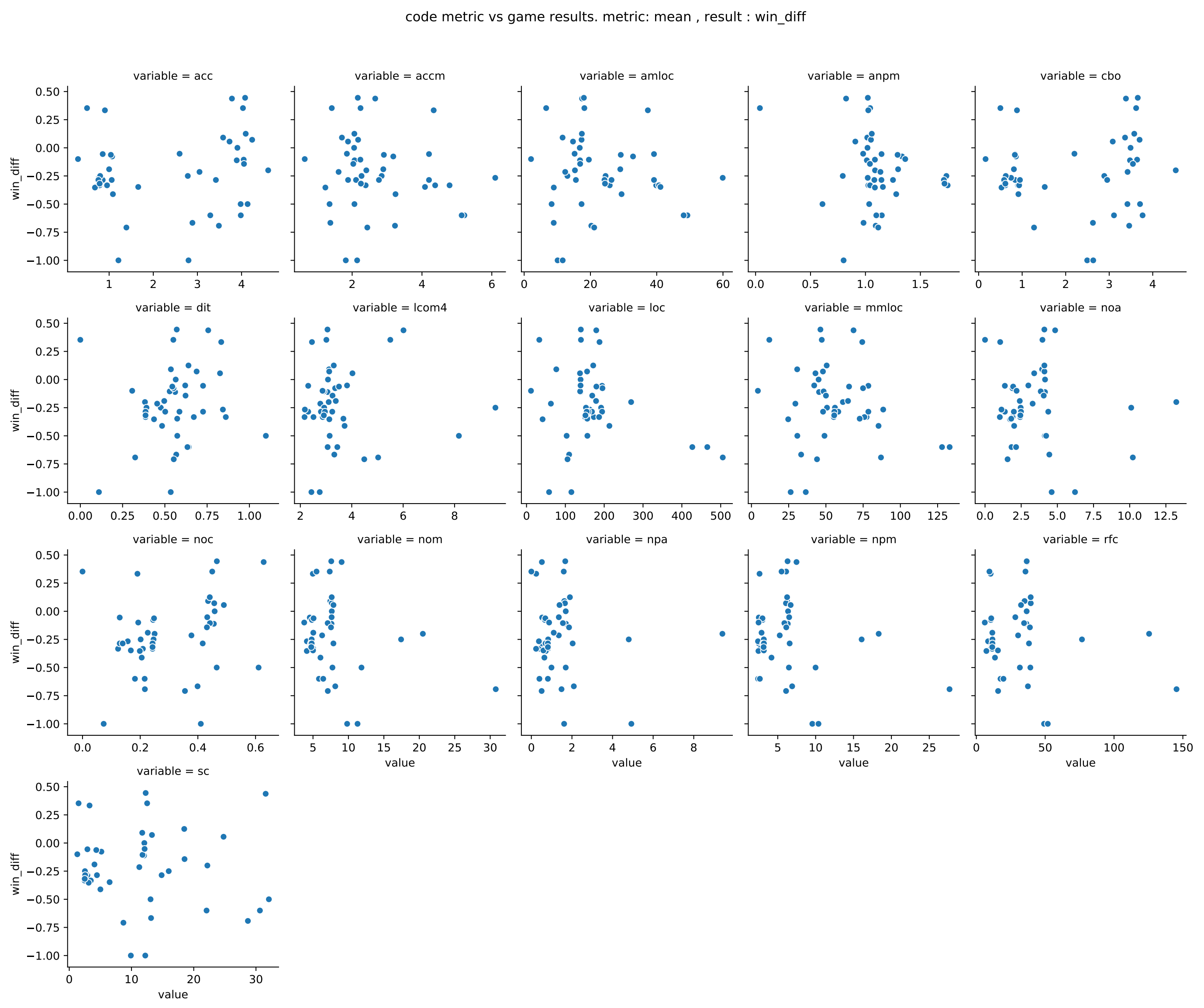


Figure 6.3: Scatter plots of a ‘win diff’ performance indicator vs the modules average of the different code metrics in each the agent repository

	win_diff
(75%, npa)	-0.380407
(25%, lcom4)	0.361000
(50%, npa)	-0.318953
(25%, noa)	-0.317907
(std, mmloc)	-0.301572
(75%, npm)	-0.290096
(max, acc)	0.283206
(std, nom)	-0.281419
(50%, dit)	0.265909
(std, accm)	-0.264787

Table 6.1: Correlation of code metrics vs normalize wins and loses performance indicator

6.3 Winner classification

The goal of the "Winner classification" experiment is to train a model to effectively classify the winner of a single match. The advantage of this model is it can focus on the difference between the data points (groups). Moreover, our data become a bit larger, since each data point represents a match and during a tournament there are few matches between each group. Another advantage of the binary model is it is eliminates the different year bias as it only compares group from the same year. The model is trained to minimize the following loss function : $L(y\_true, y\_pred)$ where $y\_pred = F(WX)$, X is the code metrics vectors of the two groups in a single match and W are the model parameters. Two different splitting methods for "Winner classification" experiments has been tested: 1. Year based folding: training several classifiers each time leaving one year's matches as validation set. 2. Simple Kfold splitting: In this setting each fold split was stratified based on the year to have close to equal proportions of the different years between train and test splits. The results of the cross validation of each setting is shown in table 6.2 and table 6.3 respectively. Although not distinct, the results suggests that code metrics of a robotic football program can be used to predict the winner of match with good success rate.

6.3.1 Feature importance

Next, The classifiers of that were trained in each split were investigated to highlight features that are dominant for the decision. Such features if found might highlight the cause for a team to outperform another team. As presented in fig. 6.4 There is no clear advantage of a single feature across the different classifiers

split	F1	AUC	supports
0	0.6875	0.781250	16
1	0.6250	0.533333	16
2	0.5000	0.698413	16
3	0.4375	0.531250	16
4	0.4375	0.681818	16

Table 6.2: Kfolds split estimated performance: Weighted average AUC: 0.65±0.10, Weighted average accuracy: 0.54±0.10

	F1	AUC	supports	year
0	0.230769	0.472222	13	2008
1	0.705882	0.871429	17	2009
2	0.400000	0.500000	5	2011
3	0.400000	0.895833	10	2012
4	0.625000	0.701389	24	2014
5	0.900000	0.958333	10	2016

Table 6.3: Year based split estimated performance: weighted average AUC: 0.74±0.17, weighted average accuracy: 0.57±0.21

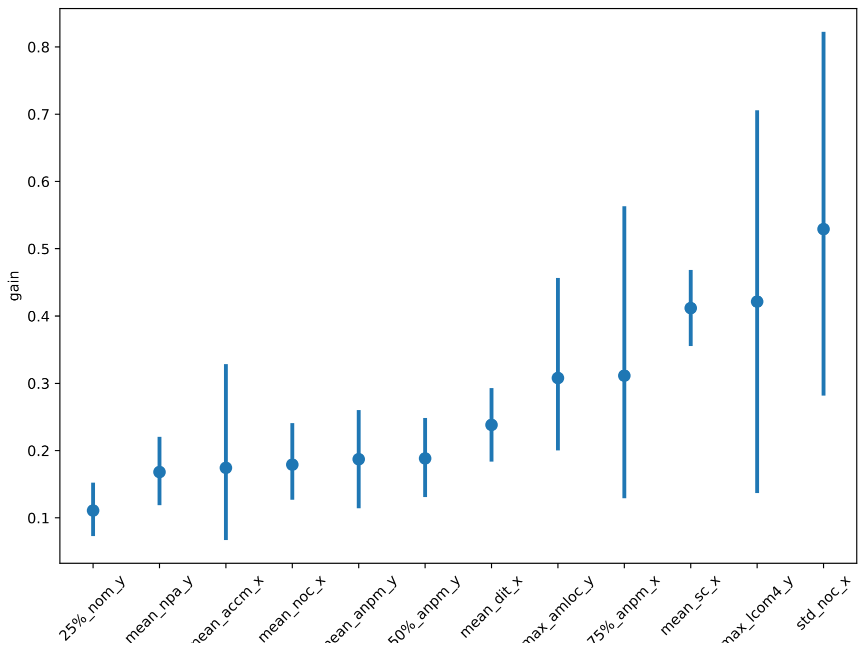


Figure 6.4: Importance values of the features in the different classifiers: Error bars present the variance between the different classifiers on the same feature

7 Conclusions & Future Work

Ultimately, our goal in this investigation is not only finding out if there is a difference between agent or robot software, and other software domains, but also to uncover the nature of this difference. This section discusses the results presented above, and attempts to draw conclusions, lessons, and hypotheses for future investigations.

Uniqueness of agent based software. First, a very obvious fact is that both the statistical analysis and the ML based identified the Autonomous agent category as having unique code attributes when compared to general software. In the manual analysis we presented avoidance that the group of the categories ANAC, Robocup 2d and Chess was the most dominant group among all that has unique behavior (low p value) this is true for 5 of the top "unique" features. However, chatbot as a category belong the autonomous agent domain was not identified as unique on the same features. Thus, features that separate between the other categories in the agents domain like loc_mean and accm_mean do not statically differentiate chatbot from other general software, we relate to this difference later at this chapter.

The uniqueness autonomous agent software becomes even more significant when running machine learning based analysis to reveal how groups of software can be clustered. The ROC curve and F1 performance of binary classifies trained to separate groups of software is proved to be significantly better when trained to separate agent software than any other type of software. Note, that this method also highlights chat-bots as an easily identified software category. Also, Machine learning analysis highlights AI platform software domain as significantly different that general code as well.

Interestingly looking into the features that were identified in the manual analysis as having stronger separability potential reveals the following insight: Agent software seems to have high values, compared to other software domains, in features that are related to size and complexity. Those features (Mean module line of code, mean method line of code, average cyclometic complexity) are presented in figs. 5.5, 5.7 and 5.10. although not as distinct, the feature rank analysis executed on the trained classifier show similar evidence as for XGBoost classifier the highly ranked features are related to LOC and complexity .

is Agent Software Inertly more complex? Both analysis efforts proves uniqueness of the agent based software with regards to their code attributes. But, we also argue that the results point out the agent code is more complex. Complexity is about more branching points, conditional loops, and decision points. In the software chapter A present an example of a module written in C++ as part of autonomous agent program for the Robocup contest. The large amount of branching point like condition inside loops and switch case statements in this code generates a code with high complexity. This code example was selected also for its relative cleanness and its OO structure to prove that there are cases that the complexity is inherited from the required functionality and not from lack of coding standards. In this example being able to implement a logic that support the analysis of the game state is complex and requires to related to many different options

Cyclomatic Complexity has been generally shown to be inversely correlated to code quality and defect frequency. Greater CC is correlated with a greater number of defects in the software, persistent bugs, and other indications of poor design and code quality. Indeed, the correlation is sufficiently accepted, that there exists recommended practices for the maintenance of CC values of new software within accepted *safe range*, below the ACCM measurements we generally see here.

Code Metrics Predicts Performance The second track of this research was done based on the proven impact of code metrics values on functional quality of software. The results of learning to predict the winning team a dual Robocup match presents fair performance based only on the code metrics. Taking into account the limitations of doing statistics on low number of data points we can suggest an evidence that code metrics values can be used for estimating performance of AI code. The lack of better results in performance prediction is explainable by the clear understanding that for winning an AI competition the quality of code itself is not enough, A domain knowledge, algorithmic superiority and other attributes are at least as important. Still, being able to have some evaluation of the quality of an AI program based on generic measurements that are not unique to any AI architecture as we present in this research can be utilize for many purposes in the software engineering of AI and agent based software

7.0.1 Future Work

The results, data and conclusions described above are hopefully a better starting point for several directions of study considering different researches on the same data, extending to other sources of data, using different techniques of analysis than used in this research and applying on practical guidelines and tooling.

Other measurements Optionally evaluating measurements other than code metrics on topics we presented might expose different attributes of the clustering we have presented. We suggest applying analysis of other measurements on the code like code smells, dynamic code analysis, data flow analysis or even other static code metrics that were not included in our research. Reevaluating our data sources and results to detect those attributes can greatly donate to the robustness of the outcomes. For example, running code smells analysis to explore the ability to classify between Agents and other software, can bring the creation of stronger highly preforming classifier that would be able work better in cases where the code metrics classifier were failing. Additionally, the data sources we used can be augmented with different types of data like cost of development, developers experience and other attributes of the data we used to address different aspect of our research question about the unique and common between AI code to other domains of software.

More Data As this research was focusing on the Robocup agents code for his unique characteristics, There are probably other unique repositories we weren't able to achieve and might as well be subject to such a similar research. Another approach of getting larger amount of data and reduce noises is to have one or more repositories in the the data source manually to tagged to differentiate between AI code and regular code. This type of resulting data source is a great way to approve or reject the research suggestion to differentiate. Lastly, the methods, results and data in this research can be used to conduct similar researches on other domains. The knowledge of general software architecture can be extended by revealing the unique characteristics of different software domains, different architectures, development groups etc.

Different analysis techniques The research describes several methods of using clustering and classification to answer the research questions there are other methods available to explore the data further. Running classical clustering algorithms like K-Means or GMM on specific metrics that were highlighted by our research to support the classifier decisions, might further reveal the relation between some metrics and the explored classes (Agents, general software). Furthermore, even though while running the analysis we explored systematically several approaches and algorithm architectures, There are many others known or unique techniques that can be used to improve the results of this research.

Tools and guidelines This research highlights a difference between the unique domain of autonomous agent and other software domains based on a common measurement. As AI and autonomous agents in particular are becoming more and more widespread in many areas of the digital world, we suggest a potential adjustment to the tools being used to measure software quality and development efficiency when it comes to those areas of development. A further research is needed to evaluate the effect of our findings on quality metrics like number of defect, time to repair and others but this foundations for those adjustment are presented in the results presented in this paper. Another direction of practical extension for our research is the usage of the difference we presented between AI code to other domains, is to create automatic method to identify programs or portions of code inside a program that behaves differently that expected and implements a more AI like behaviors. This abilities can be used to detect potential threats in the cyber security front or for creating an automatic indexing of software in code repositories

References

[1] A. J. Albrecht. Measuring application development productivity. In *IBM Applications Development Joint SHARE/GUIDE Symposium*, pages 83–92, Monterey, California, 1979.

[2] F. Alonso, J. L. Fuertes, L. Martinez, and H. Soza. Towards a set of Measures for Evaluating Software Agent Autonomy. In *Mexican International Conference on Artificial Intelligence*, pages 73–78, Nov. 2009.

[3] F. Alonso, J. L. Fuertes, L. Martínez, and H. Soza. Measuring the Pro-Activity of Software Agents. In *International Conference on Software Engineering Advances*, pages 319–324, 2010.

[4] Y. Alsultanny. Using McCabe Method to Compare the Complexity of Object Oriented Languages. *IEEE Transactions on Software Engineering*, Jan. 2009.

[5] B. W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1981.

[6] O. Boissier, R. H. Bordini, J. F. Hübner, and A. Ricci. Unravelling Multi-agent-Oriented Programming. In *Agent-Oriented Software Engineering*, pages 259–272. Springer, 2014.

[7] L. Briand, J. Daly, and J. Wust. A unified framework for coupling measurement in object-oriented systems. *IIEEE Trans. Software Eng.*, 25(1):91–121, Jan. 1999.

[8] D. Brugali. *Software Engineering for Experimental Robotics*. Springer, 2007.

[9] H. Bruyninckx. Open robot control software: the OROCOS project. In *Proceedings of IEEE International Conference on Robotics and Automation*, volume 3, pages 2523–2528, 2001.

[10] D. Calisi, A. Censi, L. Iocchi, and D. Nardi. OpenRDK: A modular framework for robotic software development. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1872–1877, 2008.

[11] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994-06.

[12] M. Cossentino, C. Lodato, S. Lopes, P. Ribino, and V. Palermo. Metrics for Evaluating Modularity and Extensibility in HMAS Systems. In *AAMAS*, 2015.

[13] R. D'Agostino and E. S. Pearson. Tests for Departure from Normality. Empirical Results for the Distributions of b2 and b1. In *a*, volume 60, pages 613–622, 1973. Publisher: [Oxford University Press, Biometrika Trust].

[14] N. T. Dantam, K. Bøndergaard, M. A. Johansson, T. Furuholm, and L. E. Kavraki. Unix Philosophy and the Real World: Control Software for Humanoid Robots. *Frontiers in Robotics and AI*, 3, 2016.

[15] L. B. L. De Souza and M. D. A. Maia. Do software categories impact coupling metrics? In *Proceedings of the Working Conference on Mining Software Repositories*, pages 217–220. IEEE Press, 2013.

[16] S. A. DeLoach. O-MaSE: An Extensible Methodology for Multi-agent Systems. In *Agent-Oriented Software Engineering*, pages 173–191. Springer, 2014.

[17] A. Elkady and T. Sobh. Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography. *Journal of Robotics*, 2012:1–15, 2012.

[18] N. Fenton and J. Bieman. *Software Metrics: A Rigorous and Practical Approach, Third Edition*. CRC Press, 2014.

[19] I. garcia magarino, M. Cossentino, and V. Seidita. A Metrics Suite for Evaluating Agent-oriented Architectures. In *Proceedings of the ACM Symposium on Applied Computing*, pages 912–919, New York, NY, USA, 2010. ACM.

[20] B. P. Gerkey, R. T. Vaughan, and A. Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the International Conference on Advanced Robotics*, 2003.

[21] J. J. Gomez-Sanz. Ten Years of the INGENIAS Methodology. In *Agent-Oriented Software Engineering*, pages 193–209. Springer, 2014.

[22] M. H. Halstead. *Elements of Software Science*. Operating and Programming Systems Series. Elsevier Science, 1977.

[23] M. Hitz and B. Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of International Symposium on Applied Corporate Computing*, pages 25–27, 1995.

[24] R. V. Hudli, C. L. Hoskins, and A. V. Hudli. Software metrics for object-oriented designs. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 492–495, 1994.

[25] N. R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117(2):277–296, 2000.

[26] C. Jones. *Applied Software Measurement: Global Analysis of Productivity and Quality*. McGraw-Hill, New York, 3rd edition, 2008.

[27] G. A. Kaminka and I. Frenkel. Flexible teamwork in behavior-based robots. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, 2005.

[28] G. A. Kaminka and I. Frenkel. Integration of coordination mechanisms in the BITE multi-robot architecture. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA-07)*, 2007.

[29] R. V. Kumar and R. Chandrasekaran. Classification of software projects using k-means, discriminant analysis and artificial neural network. *International Journal of Scientific & Engineering Research*, 4(2):7, 2013.

[30] R. L and H. L. Software Quality Metrics for Object-Oriented System Environments. *a*, 1995.

[31] M. Luck, P. McBurney, O. Shehory, and S. Willmott, editors. *Agent Technology: Computing as Interaction—A roadmap for Agent-Based Computing*. University of Southampton/Agentlink III, 2005.

[32] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[33] P. Meirelles, C. Santos Jr., J. Miranda, F. Kon, A. Terceiro, and C. Chavez. A study of the relationships between source code metrics and attractiveness in free software projects. In *Proceedings of the Brazilian Symposium on Software Engineering*, pages 11–20. IEEE, 2010.

[34] P. R. M. Meirelles. *Monitoramento de metricas de codigo-fonte em projetos de software livre*. PhD thesis, Universidade de São Paulo, May 2013.

[35] M. Montemerlo, N. Roy, and S. Thrun. Perspectives on standardization in mobile robot programming: the Carnegie Mellon Navigation (CARMEN) Toolkit. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 3, pages 2436–2441, Oct. 2003.

[36] L. Padgham, J. Thangarajah, and M. Winikoff. Prometheus Research Directions. In *Agent-Oriented Software Engineering*, pages 155–171. Springer, 2014.

[37] E. K. Piveta, A. Moreira, M. S. Pimenta, J. Araújo, P. Guerreiro, and R. T. Price. An empirical study of aspect-oriented metrics. *Science of Computer Programming*, 78(1):117–144, 2012.

[38] E. Platon, N. Sabouret, and S. Honiden. An architecture for exception management in multiagent systems. *International Journal of Agent-Oriented Software Engineering*, 2(3):267, 2008.

[39] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng. ROS: an open-source Robot Operating System. In *Proceedings of IEEE International Conference on Robotics and Automation*, 2009.

[40] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, Mar. 1993.

[41] M. Stojkovski. Thresholds for software quality metrics in open source android projects. Master's thesis, NTNU, 2017.

[42] A. Sturm and O. Shehory. Agent-Oriented Software Engineering: Revisiting the State of the Art. In *Agent-Oriented Software Engineering*, pages 13–26. Springer, 2014.

[43] E. Tsardoulias and P. Mitkas. Robotic frameworks, architectures and middleware comparison. arXiv:1711.06842 [cs], Nov. 2017.

[44] M. Winikoff. Future Directions for Agent-Based Software Engineering. *International Journal of Agent-Oriented Software Engineering*, 3(4):402–410, May 2009.

A Code Example

```
// basic.cxx
//
////////////////////////////////////////////////////

#include "common.hxx"
#include "basic.hxx"

#ifndef NDEBUG
#  include "basic.inl"
#endif

//////////////////////////////////////////////////
// PlayModeHelper

PlayModeHelper playModeHelper;

const char* const PlayModeHelper::m_playmodeStrings[PLAYMODE_MAZ] =
    {
  "kick_off_l",
    "kick_off_r",
    "kick_in_l",
    "kick_in_r",
    "free_kick_l",
    "free_kick_r",
    "corner_kick_l",
    "corner_kick_r",
    "goal_kick_l",
    "goal_kick_r",
    "goal_l",
    "goal_r",
    "offside_l",
    "offside_r",
    "goalie_catch_ball_l",
    "goalie_catch_ball_r",
    "before_kick_off",
    "time_over",
    "play_on",
    "drop_ball",
};

PlayModeHelper::~PlayModeHelper()
{
}
PlayModeHelper::PlayModeHelper()
{
}


//////////////////////////////////////////////
// RefreeMessageHelper

RefreeMessageHelper refreeMessageHelper;

const char* const RefreeMessageHelper::m_refreeMessageStrings[] = {
  "time_up_without_a_team",
    "time_up",
    "time_extended",
    "half_time",
    0,
    "faul_l",
    "faul_r",
    "goalie_catch_ball_l",
    "goalie_catch_ball_r",
    0,
};


////////////////////////////////////////////
// FlagHelper


......

FlagHelper::FlagHelper()
{
#ifndef NDEBUG
```

```
  for(int i=0; i<FLAG_MAZ; i++) {
    ASSERT(i == m_flagToSymmetric[m_flagToSymmetric[i]]);
  }
#endif
  m_initialized = false;
}
void FlagHelper::initialize(double goalWidth)
{
  ASSERT(goalWidth > 0);
  m_initialized = true;
  int x, y, i;
  for(y=0; y<3; y++) {
    for(x=0; x<3; x++) {
      i = FLAG_LT + y*3 + x;
      m_flagToPosition[i] = Vector((x-1) * server().PITCH_LENGTH()
   /2,
        (y-1) * server().PITCH_WIDTH()/2);
    }
  }
  for(y=0; y<3; y++) {
    for(x=0; x<2; x++) {
      i = FLAG_PLT + y*2 + x;
      m_flagToPosition[i] = Vector((x*2-1) * (server().PITCH_LENGTH
   ()/2 - server().PENALTY_AREA_LENGTH()),
        (y-1) * server().PENALTY_AREA_WIDTH()/2);
    }
  }
  for(y=0; y<2; y++) {
    for(x=0; x<2; x++) {
      i = FLAG_GLT + y*2 + x;
      m_flagToPosition[i] = Vector((x*2-1) * server().PITCH_LENGTH()
   /2,
        (y*2-1) * goalWidth/2);
    }
  }
  for(y=0; y<2; y++) {
    for(x=0; x<11; x++) {
      i = FLAG_TL50 + y*11 + x;
      m_flagToPosition[i] = Vector((x-5) * 10.0,
        (y*2-1) * (server().PITCH_WIDTH()/2 + 5.0));
    }
  }
  for(y=0; y<7; y++) {
    for(x=0; x<2; x++) {
      i = FLAG_LT30 + y + x*7;
      m_flagToPosition[i] = Vector((x*2-1) * (server().PITCH_LENGTH
    ()/2 + 5.0),
        (y-3) * 10.0);
    }
  }
}

static Flag nameToFlagSimple(const char* name)
{
  ASSERT(name[0] != '\0');
  int p1 = name[0];
  int p2 = (name[1] == '\0') ? 0 : name[2];
  int p3 = (p2 == 0 || name[3] == '\0') ? 0 : name[4];
  switch(p1) {
  case 'c':
    switch(p2) {
    case 't':   return FLAG_CT;
    case 0:     return FLAG_C;
    case 'b':   return FLAG_CB;
    }
    break;
  case 'p':
    switch(p2) {
    case 'l':
      switch(p3) {
      case 't': return FLAG_PLT;
      case 'c': return FLAG_PLC;
      case 'b': return FLAG_PLB;
      }
      break;
    case 'r':
      switch(p3) {
      case 't': return FLAG_PRT;
      case 'c': return FLAG_PRC;
      case 'b': return FLAG_PRB;
      }
      break;
    }
    break;
  case 'g':
    switch(p2) {
```

```
    case 'r':
      switch(p3) {
      case 't': return FLAG_GRT;
      case 'b': return FLAG_GRB;
      }
      break;
    case 'l':
      switch(p3) {
      case 't': return FLAG_GLT;
      case 'b': return FLAG_GLB;
      }
      break;
    }
    break;
  case 'l':
    switch(p2) {
    case 0:     return GOAL_L;
    case 't':
      switch(p3) {
      case 0:   return FLAG_LT;
      case '3': return FLAG_LT30;
      case '2': return FLAG_LT20;
      case '1': return FLAG_LT10;
      }
      break;
    case '0':   return FLAG_L0;
    case 'b':
      switch(p3) {
      case '1': return FLAG_LB10;
      case '2': return FLAG_LB20;
      case '3': return FLAG_LB30;
      case 0:   return FLAG_LB;
      }
      break;
    }
    break;
  case 'r':
    switch(p2) {
    case 0:     return GOAL_R;
    case 't':
      switch(p3) {
      case 0:   return FLAG_RT;
      case '3': return FLAG_RT30;
      case '2': return FLAG_RT20;
      case '1': return FLAG_RT10;
      }
      break;
    case '0':   return FLAG_R0;
    case 'b':
      switch(p3) {
      case '1': return FLAG_RB10;
      case '2': return FLAG_RB20;
      case '3': return FLAG_RB30;
      case 0:   return FLAG_RB;
      }
      break;
    }
    break;
  case 't':
    switch(p2) {
    case 'l':
      switch(p3) {
      case '5': return FLAG_TL50;
      case '4': return FLAG_TL40;
      case '3': return FLAG_TL30;
      case '2': return FLAG_TL20;
      case '1': return FLAG_TL10;
      }
      break;
    case '0':   return FLAG_T0;
    case 'r':
      switch(p3) {
      case '1': return FLAG_TR10;
      case '2': return FLAG_TR20;
      case '3': return FLAG_TR30;
      case '4': return FLAG_TR40;
      case '5': return FLAG_TR50;
      }
      break;
    }
    break;
  case 'b':
    switch(p2) {
    case 'l':
      switch(p3) {
      case '5': return FLAG_BL50;
      case '4': return FLAG_BL40;
      case '3': return FLAG_BL30;
      case '2': return FLAG_BL20;
      case '1': return FLAG_BL10;
      }
      break;
    case '0':   return FLAG_B0;
    case 'r':
      switch(p3) {
      case '1': return FLAG_BR10;
      case '2': return FLAG_BR20;
      case '3': return FLAG_BR30;
      case '4': return FLAG_BR40;
      case '5': return FLAG_BR50;
      }
      break;
    }
    break;
  }
  ASSERT(false);
  return FLAG_C;
}

Flag FlagHelper::nameToFlag(const char* name, Side teamSide)
{
  ASSERT(name);
  ASSERT(teamSide == SIDE_LEFT || teamSide == SIDE_RIGHT);
#if 1
  Flag flag = nameToFlagSimple(name);
  ASSERT(0 <= flag && flag < FLAG_MAZ);
  ASSERT(strcmp(name, m_flagNames[flag]) == 0);
  if(teamSide == SIDE_RIGHT)
    flag = m_flagToSymmetric[flag];
  return flag;
#else
  int i = 0;
  for(;i < FLAG_MAZ ; i++) {
    if(strcmp(name, m_flagNames[i]) == 0) {
      if(teamSide == SIDE_RIGHT)
        i = m_flagToSymmetric[i];
      ASSERT(0 <= i && i < FLAG_MAZ);
      return (Flag)(Flag_t)i;
    }
  }
  ASSERT(false);
  return FLAG_C;
#endif
}
```