https://doi.org/10.1142/9789811239922_0004

Chapter 4

Intelligent Agents are More Complex: Initial Empirical Findings

Gal A. Kaminka and Alon T. Zanbar
Bar-Ilan University, Israel
galk@cs.biu.ac.il; atzanbar@gmail.com

4.1 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, middleware software, have been discussed in the literature, encompassing multiple communities of researchers, with at least partial overlaps in interests and approaches.

The fundamental assumption underlying 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 [1] 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 circles [2–5].

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.

This paper provides *the first empirical evidence for the distinctiveness of autonomous agent software*, compared to other software categories. 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. With each, we utilize general source code metrics, such as *Cyclomatic Complexity*, *Cohesion*, *Coupling*, and others used by general software engineering researchers to quantify meaningful characteristics of software (over 250 measures, see below).

We conducted both statistical and machine-learning analysis, to determine (1) whether agents emerge as a distinguishable sub-group within the pool, and (2) whether there are clear distinguishing measures. We find that agent software is *clearly* and *significantly* different from other types of software of comparable size. This result appears both when using manual statistical analysis, as well as machine learning methods. Specifically, autonomous agents software is significantly more complex (in the sense of control flow complexity) than other software categories. We discuss potential implications of these results.

4.2 Background

There is vast literature reporting on software engineering 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, encompassing the collective efforts of the various communities engaged in relevant research. We emphasize no bias in the selection of this name, and 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] [1, 2, 6–9]. For the most part, the arguments for the study of AOSE as distinct from general software

[1]International Journal of Agent-Oriented Software Engineering.

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 (e.g., [10–16] similarly argue *qualitatively* for distinguishing software engineering in robotics. Some emphasize specific middleware frameworks (e.g., [10–12, 17–19]), while others focus on critical capabilities or approaches [20–23]). These are detailed and well-reasoned arguments, however 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. 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.

The rarity of quantitative investigations in AOSE (see below for notable exceptions) is not for lack of quantitative methods in *general* software engineering. Beginning with the 1970s pioneering research on Cyclomatic Complexity [24] and Halstead measures [25] 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 [26–28]. 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 [24, 29, 30]. 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 [31]. See [32] for a comprehensive survey.

Software metrics have been used to classify software, or cluster together software based on measured characteristics, as we do in this paper [33]. For example, De Souza and Maia defined software metric thresholds based on context ([34]). [35] showed this approach is applicable for Android projects. Another example can be found in [36], who found linkage between the size and complexity of open source projects, to attractiveness of the project for contributors.

Surprisingly, despite the prevalence and usefulness of software metrics as noted above, the use of software metrics in intelligent agent and robot

software remains limited, and is not generally reported in relevant literature. Perhaps this is due to lack of data, or the relative novelty of products, which leads to sparse and relative rare expertise in commercial-grade development. AOSE-specific software metrics, specialized to agent programming paradigms and languages, have been proposed in AOSE research circles [37–40], often specific to agent-oriented programming languages (e.g., 2/3APL, JASON). Because of their specialized use cases, which prohibit their use in general software, we were reluctant to use them in this study, which uses general metrics to contrast software from many different categories.

4.3 Software Project Data Collection and Curation

We begin with an overview of the data collection and curation process. The data collected will be used in the analysis processes described in Secs. 4.4–4.5.

4.3.1 *Data Sources*

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. Within each division, there are multiple leagues, with their own rules. 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). 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.

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.

Additional data. We additionally found open source robotics projects from the DARPA Grand and Urban challenges, and from industrial projects where our lab was involved in research.

4.3.2 *Automatic Data Harvesting*

From the sources above, we first collected agent and robot software projects — all we could find and use: 2D RoboCup teams for which source code is available, the ANAC agent projects, robotics software from RoboCup and the other sources described above. We extended the search for relevant software to github projects tagged *chess*, as problem solving is a close AI domain.

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.

The process of collecting and filtering of repositories from GitHub was *automatic*, as described in Fig. 4.1. The primary constraint in selecting software projects is comparability. The source code collected for agents uses C, C++, and Java, and so we restricted ourselves to projects in these languages, to prevent language-specific bias in the metrics. Similarly, we restricted ourselves to software size (measured in lines of code — LOC) in comparable ranges.

- Programming languages : C, C++, Java

- high Level of maturity
- Distinct classification in github (for github projects)
- Size > 900 lines of code (LOC)

As a control group to the software projects above (focusing on AI and agents), we similarly harvested software projects in domains very different from agents or AI. Table 4.1 shows 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, 20 projects classified as autonomous robots, and 377 projects in other categories. Table 4.2 lists the minimum, maximum and median project size in each domain, measured in LOC.

Table 4.1 Software project data breakdown.

Classification	Source	Software Domain	Size	Maturity Indicator
Autonomous Agents	RoboCup 2D simulation	Virtual Robots	64	Qualification for RoboCup
	ANAC	Negotiating Agents	26	Qualification for ANAC
	GitHub	Chess-Playing Engines	28	> 5 GitHub stars
General	GitHub	Audio	54	> 5 GitHub stars
		Education	50	
		Finance	26	
		Games	34	
		Graphics	60	
		IDE	53	
		Mobile Applications	42	
		Security	58	
Robots	DARPA Challenges	Autonomous Car	2	Qualification for Challenge
	RoboCup competition	Soccer Physical Robots	15	
	Applied R&D Projects	Robots	3	

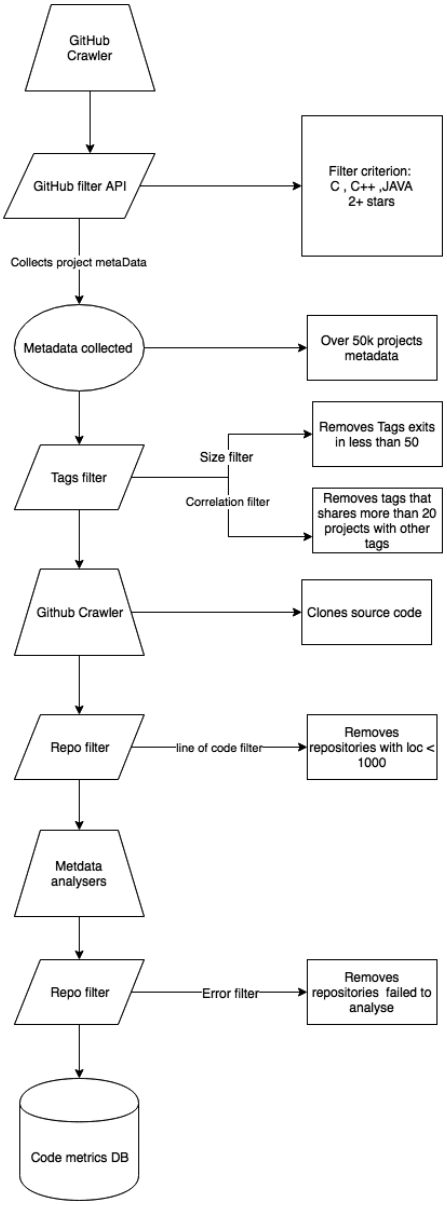

Fig. 4.1 Automatic flow of selecting GitHub projects.

Table 4.2 Software projects min, max, and median LOC.

Software Domain	Min, Max, Median Size in LOC
Virtual Robots	1010, 153661, 23495
Negotiating Agents	1031, 102816, 1352
Chess-Playing Engines	1084, 59108, 5311
Audio	1065, 1912860, 17010
Education	1026, 393360, 6933
Finance	1136, 450524, 10455
34	1393, 185784, 5064
Graphics	1168, 385036, 18769
IDE	1457, 401897, 32486
Mobile Applications	1210, 129366, 4658
Security	1214, 164228, 10341
Autonomous Car	117848, 117848, 117848
Soccer Physical Robots	15335, 793966, 54895
Robots	3131, 64028, 10588

4.3.3 *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. 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. Additionally we save information on the context of the repository (name, location, category) and other information like source code language, competition results, the year in which the code was deployed, etc.

We used two different tools, independently, to allow validation of the results: CCCC[2] and Analizo.[3] The two tools were run on two 24-core XEON servers, each with 76GB of ram. Total CPU time is more than a month.

The measurement tools provide the following general software metrics, for different level of analysis (see [32] 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 [37–40], and specialized languages (e.g., 2/3APL, JASON).

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

Summary & Project Level Metrics:

- Total Lines of Code (total_loc)
- Total Number of Modules (total_modules)
- Total Number of Methods (total_nom)

Module Level Metrics:

- Afferent Connections per Class (ACC)
- Average Cyclomatic Complexity per Method (ACCM)
- Average Method Lines of Code (AMLOC)
- Average Number of Parameters (ANPM)
- Coupling Between Objects (CBO)
- Coupling Factor (COF)
- Depth of Inheritance Tree (DIT)
- Lack of Cohesion of Methods (LCOM4)
- Lines of Code (LOC)
- Number of Attributes (NOA)
- Number of Children (NOC)
- Number of Methods (NOM)
- Number of Public Attributes (NPA)
- Number of Public Methods (NPM)
- Response for Class (RFC)
- Structural Complexity (SC)

We collected not only the raw metrics above, but also their aggregation in various ways, so as to minimize the inherent loss of information. Thus for each metric, we also computed its mean, mode, minimum value, maximum value, quantiles (lower, max, median, min, ninety five, upper), standard deviation, variance, skewness, and kurtosis. All in all, each software project was represented by more than 250 measurements.

4.4 Statistical Analysis

We conducted two separate analysis efforts which had common general goal. This section details the results of a statistical analysis, while the next section 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.

Every project is represented by approximately 250 different metrics. As such, it is difficult to attempt to find differentiating metric by hand. We therefore used a heuristic procedure to assist in finding promising features. 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.

The idea is to iterate over the software domains. For each domain r, we separate it out from the others, and then use a two-tailed t-test to contrast the distribution of the metric values in the domain and in all others. For example, one iteration of the algorithm would run two-sample t-test between the values of accm_mean of projects in the *RoboCup 2D* against the values of accm_mean of all projects in the control group (tagged as 'non agent').

A lower p value from the t-test is used as a heuristic, indicating that potentially a good differentiating feature has been detected. We collect all the domains differentiated by the metric f into a common set indexed by f. We then look for sets larger than two. We use a threshold to avoid distractions from a metric that may distinguish a specific domain from all others, by chance.

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** *2-tailed 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: $selected_f \leftarrow CommonSet_f$
return all $selected_f$

Table 4.3 shows the output of the algorithm for each individual metric, when listed in increasing order of probability (i.e., in order of *decreasing* indication of separation power). The top four metrics are the ACCM mean and its upper and median quantiles, and the Coupling Factor (CoF) metric, which measures coupling between modules. These four metrics clearly distinguish between the agent domains (RoboCup 2D Simulation, Chess, ANAC agents).

Table 4.3 The top distinguishing features in descending order, and the software domains they cluster together.

Metric	Repositories	p Value
accm_mean	[RoboC-2D, Chess, ANAC]	1.18E-04
accm_quantile_upper	[RoboC-2D, Chess, ANAC]	8.78E-04
accm_quantile_median	[RoboC-2D, Chess, ANAC]	1.17E-03
total_cof	[RoboC-2D, Chess, ANAC]	1.19E-03
noa_skewness	[RoboC-2D, IDE, Graphics]	2.59E-03
nom_quantile_upper	[RoboC-2D, ANAC, Audio]	6.27E-03
amloc_quantile_upper	[RoboC-2D, Chess, ANAC, ...]	7.99E-03
nom_mean	[RoboC-2D, ANAC, Graphics, Audio]	1.07E-02
anpm_quantile_upper	[RoboC-2D, ANAC, Graphics]	1.14E-02
noa_kurtosis	[RoboC-2D, Ide, Games, Graphics]	1.28E-02

We use the p value in Table 4.3 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.

We then moved to examining the results visually, using box-plots to display the distribution of specific metrics of each software domain. We seek features which, as clearly as possible, distinguish the three classes of domains.

Indeed some metrics clearly are different between domains. For example, Fig. 4.2 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 *RoboCup-Other-Leagues* group stands out, compared to the other software domains. However, it is the only domain in the cluster, and does not distinguish the agents or robots domains from others.

Other metrics may sometimes cluster together more than one domain, but are not able to distinguish agents from non-agents code. For example, Fig. 4.3 show the distribution of the Structural Complexity metric. We can see that the inner-quantile range and median are similar between *RoboCup 2D* and *Other RoboCup Leagues*, suggesting some commonality in behavior of the structure of classes and objects. However, it reveals no commonality between the different domains of the same class (Agents, Robots, or General Software). In other words, it distinguishes some domains from others (to an extent) but does not cluster together domains that come from the same software class.

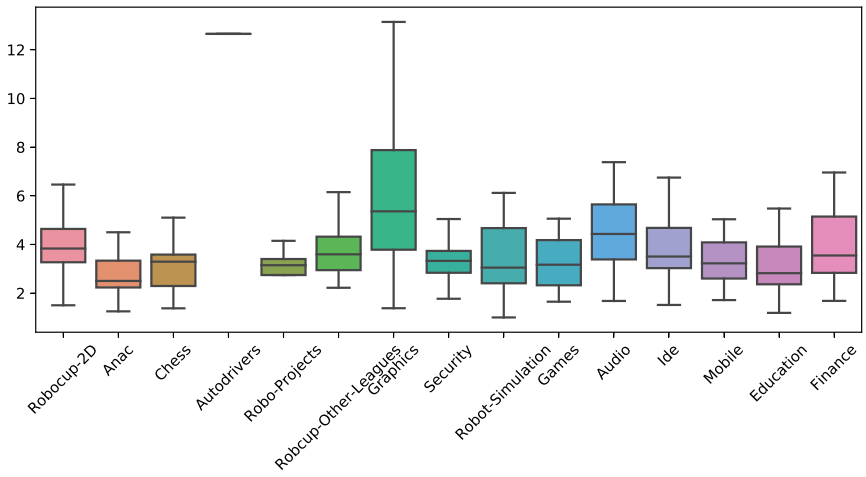

Fig. 4.2 Box plot distribution of LCOM4 means. The vertical axis range is 0–12.

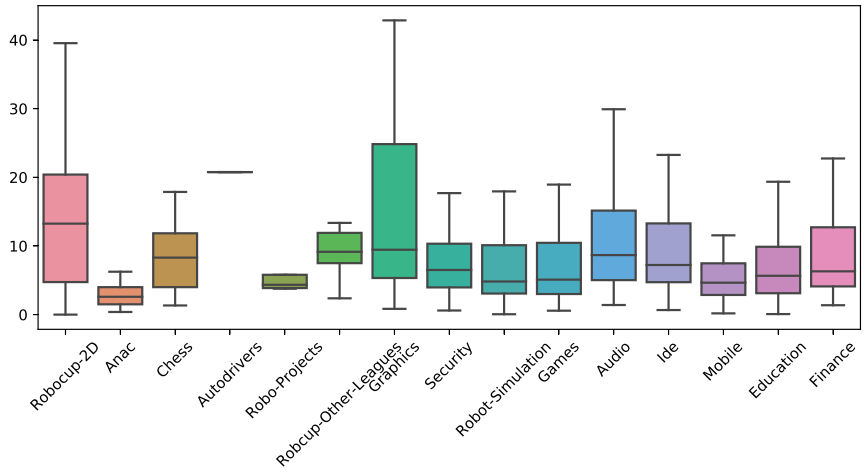

Fig. 4.3 Box plot distribution of the Structural Complexity metric for software domains. The vertical axis range is 0–40. *RoboCup 2D simulation* and *RoboCup-Other-Leagues* have larger variance and higher values than all other software domains.

In contrast, metrics that were ranked high by Alg. 1 visually show much more promise. For example, Table 4.3 suggests the mean ACCM is and the MLOC upper quartile are promising, in terms of their ability to distinguish between agents and non-agent software. Figures 4.4 and 4.5 show the box

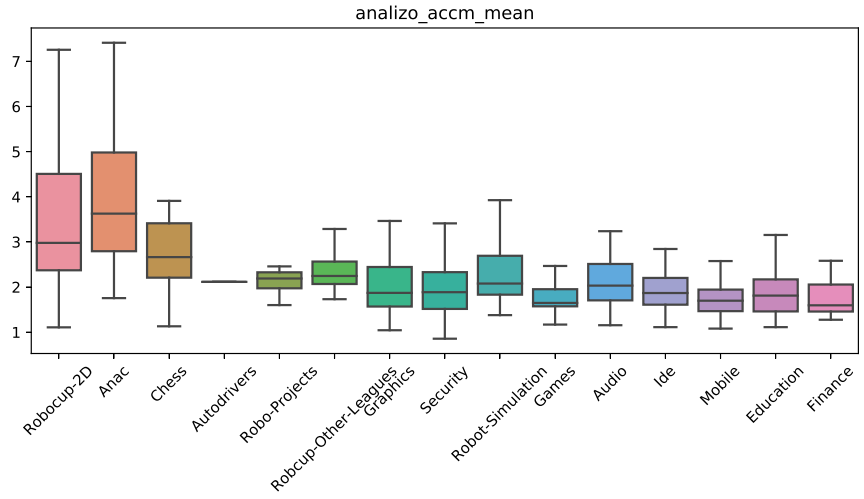

Fig. 4.4 Box plot distribution of mean ACCM of software domains. The vertical axis range is 0–8.

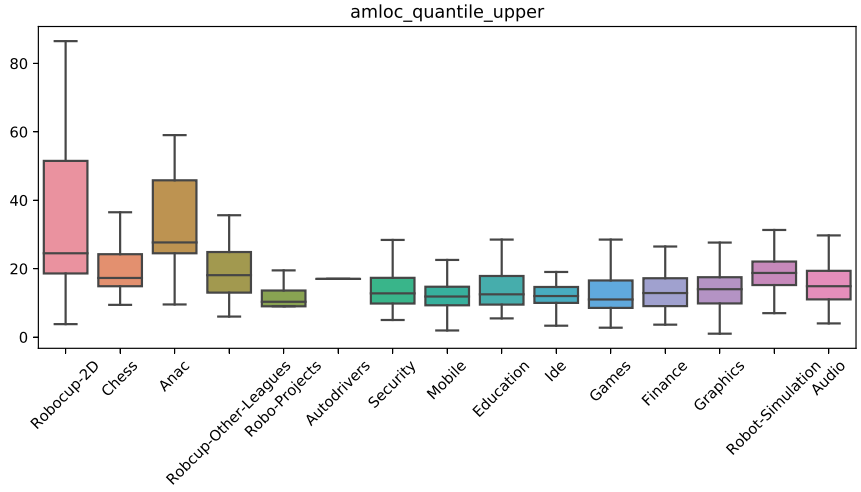

Fig. 4.5 Box plot distribution of AMLOC upper quantile. The vertical axis range is 0–40.

plot distributions these metrics. Visually, in both figures, the box-plots for the Agents class (RoboCup 2D, ANAC, Chess) are clearly prominent relative to other software domains.

Interim Summary. We defer a discussion of the *meaning* of these findings to Sec. 4.6. For now, based on the manual analysis procedure described, we only state the hypothesis that ACCM and AMLOC metrics are different between autonomous agents software and general software in other domains. Moreover, it seems robotics code lies somewhere in between, in terms of these metrics.

4.5 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.

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.

Pre-processing the data. We filtered outliers at the top and bottom 3% of the data (i.e., within the 3–97 percentiles). Aggregated features (total, median, etc.) were removed to minimize the effect of project size on the model, and to reduce the number of features (standing originally at around 250). The data was divided into a training (85%) and testing (15%) sets.

Classification procedure. 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. This creates an inherent imbalance in the number of examples presented, which we alleviated by using random over-sampling of the minority class.

For classification, we used the following classification algorithms: *Support Vector Machines*, *Logistics Regression*, and *Gradient-Boosted Decision Trees*. The implementations are open-source packages (scikit-learn[4] and XGBoost[5]). 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 4.4–4.6) shows the top classifiers

[4]https://scikit-learn.org/.
[5]https://github.com/dmlc/xgboost.

Table 4.4 SVM top five scoring software domain classifiers.

Class	Repositories	F1 Score	Feature Ranking
Agent	Robocup-2D	0.67	[accm_quantile_median, accm_quantile_upper, noc_mean]
Agent	ANAC	0.62	[accm_quantile_lower, noa_quantile_lower, noc_mean]
General	IDE	0.33	[acc_quantile_lower, dit_mean, noc_quantile_upper]
General	Mobile	0.32	[acc_quantile_lower, acc_quantile_median, accm_quantile_lower]
General	Graphics	0.20	[accm_quantile_lower, dit_quantile_lower, dit_quantile_median]

Table 4.5 Logistic Regression top scoring software classes.

Class	Repositories	AUC	F1	Feature Ranking
Agent	ANAC	0.99	0.80	[accm_quantile_upper, noa_quantile_lower, rfc_quantile_lower]
Agent	Robocup-2D	0.97	0.70	[amloc_quantile_upper, noc_mean, npa_mean]
Robot	Robcup-Other-Leagues	0.87	0.50	[amloc_quantile_lower, cbo_mean, nom_mean]
General	IDE	0.77	0.44	[amloc_quantile_median, anpm_mean, npa_mean]
General	Graphics	0.76	0.24	[anpm_mean, lcom4_mean, mmloc_mean]

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.

Table 4.4 shows the top results from the SVM classifiers, in decreasing order of performance. SVM classification output is only "Hard decision" without probability distribution of the different classes and thus the AUC score is not available for it. We used the default SVM parameters in the implementation. The F1 scores in the table are far from indicating great success, yet we note the presence of the mean ACCM metric in the list of features important for classification for the repositories belongs to the "Agent" class.

We next used classifiers built using Logistic Regression (LR). We used L2 regularization, and stopping criteria of 100 iterations. The top LR classifiers are reported in Table 4.5. In general their scores are lower than the SVM classifiers reported above.

Finally, 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. The results are shown in Table 4.6. Overall, the results are much better than the other two classification attempts. Some individual domain classifiers achieve high scores.

Table 4.6 Gradient Boosted Decision Trees top scoring software classes, in decreasing order of F1 scores.

Agent/General	Class (Domain)	AUC	F1
Agent	Robocup-2D	0.97	0.85
Agent	ANAC	0.98	0.67
Agent	Chess	0.84	0.44
Robot	Robcup-Other-Leagues	0.89	0.40
General	Graphics	0.65	0.31
General	Security	0.76	0.27
General	Mobile	0.80	0.22
General	Games	0.49	0.00
General	Audio	0.56	0.00
General	Robot-Simulation	0.66	0.00
General	Education	0.66	0.00
General	Finance	0.73	0.00
General	IDE	0.75	0.00
Robot	Robo-Projects	0.86	0.00

Most importantly, however, we note that the top performing classifiers (1) are those that are able to distinguish agent software from other types of software, and (2) utilize the mean ACCM and AMLOC metrics in their classification decisions. These results concur with the conclusions of the manual analysis described earlier. We also observe that software from physical robots participating in RoboCup (domain: *Robocup-Other-Leagues*) has also been classified successfully, using the AMLOC metric (among other metrics).

4.6 Discussion

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.

ACCM and Control Complexity of Agents. First, it is clear that the ACCM measure is a recurring metric in successful classification schemes distinguishing agent software from other software. This is true both in the manual analysis, as well as in classifiers generated by machine learning algorithms. In general, Agent software seems to have high ACCM measurements, compared to other software domains. Robot software does have

higher ACCM (on average) than non-agent software, but the difference is much less pronounced than between Agent and general software. It is therefore immediately interesting to better understand what the ACCM actually measures.

ACCM — Average Cyclomatic Complexity per Method — is a more modern variant of the Cyclomatic Complexity (CC) metric introduced by McCabe in 1976 [24]. Briefly, the cyclomatic complexity of software is a measure of the number of possible execution paths through its control flow graph. The more branching points, conditional loops, and decision points in the software, the greater its CC. The ACCM measures the CC value at the method level, for all methods within a module. It then computes the mean of these measurements to introduce a single value which represents the complexity of the module as a whole.

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.

Is Agent Software Inherently More Complex? (In short: YES!) There are alternative explanations for the higher ACCM values we observe in agent software: (1) that agent software is just inherently more complex, because the tasks tackled by the software *requires* greater complexity in the control flow of the software. Or, (2) that the agent code is just more buggy, or written by programmers who are not as well-trained, e.g., too academic?

We offer evidence that the first explanation is the correct one, i.e., that agent software is inherently more complex. One benefit of using competition software in this study is that alongside the software metrics, we also have clear quality metrics in terms of the success of the software. Specifically, we show below (Fig. 4.6) a plot of the ACCM measure from a subset of RoboCup software agent, vs the code effectiveness as measured by the mean goal difference of the agents in competitions. We see a clear inverse relation between the two: higher ACCM is associated with poor performance, just as it is in other software domains. However, the ACCM of *winning* agents is still higher than standard practice in software.

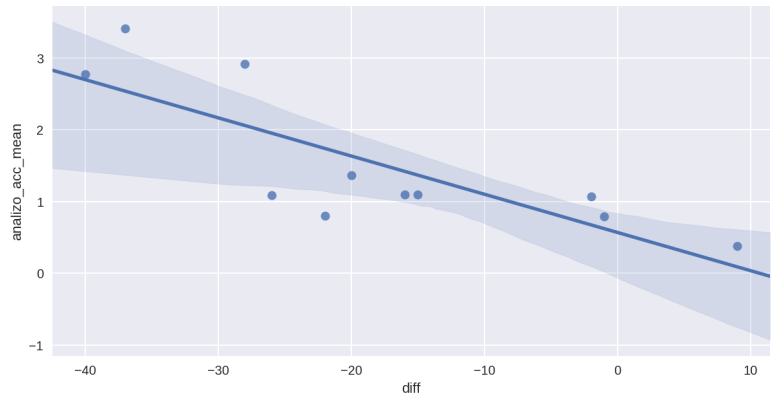

Fig. 4.6 ACCM (Average Cyclomatic Complexity per Module, vertical axis) vs Effectiveness (here, measured by *mean goal difference per game* — horizontal axis, larger is better). The goal difference was extracted automatically from log files of individual games.

What about other measures? A critical look at the results of this study raises the issue of other measures. It is true that ACCM is a clear distinguishing characteristic of agent vs non-agent code. However, it is not so clear that the machine learning classifiers can use it, ignoring other metrics. Indeed, some very successful classifiers do not use ACCM at all. Indeed, we saw also that the AMLOC measure is also a potentially good metric from this respect, as well as the MMLOC measure.

While we do not refute the possibility that other metrics may be as good as ACCM or complement it, we point out that many metrics are known to be correlated in practice (see, e.g., [41]), and thus it may be that a machine learning classifier using a particular metric could have also worked as well with a different one, that is highly correlated. In particular, in our own study here, we found that the Pearson correlation between AMLOC and ACCM is 0.84, and the correlation between MMLOC and ACCM is 0.90. So a preference for one metric over another does not necessarily mean that the other metric was not as useful.

4.7 The Big Picture and Future Directions

This paper offers the first empirical evidence that agent software is indeed inherently different from other types of software, intended for other domains. The empirical evidence was collected by analyzing hundreds of

software projects of comparable sizes, using two different types of analysis. In particular, we find that *agent software has greater control flow complexity* in general, which conjecture to be inherent to the types of tasks agents are deployed to solve — tasks that require autonomy in decision-making, and thus careful deliberation over many possibilities.

Given this conclusion, it becomes clear that agent-oriented software engineering can increase their impact by providing tools, methodologies, and frameworks that directly tackle the issue of complexity. For instance, agent architectures may be so successful because they assist in breaking down the inherent complexity of tasks. We leave this question for future work.

References

[1] Y. Shoham, Agent-oriented programming, *Artif. Intell.* **60**, 1, pp. 51–92 (1993), http://dx.doi.org/10.1016/0004-3702(93)90034-9.

[2] L. Padgham, J. Thangarajah and M. Winikoff, Prometheus Research Directions, in *Agent-Oriented Software Engineering*. Springer, Berlin, Heidelberg, ISBN 978-3-642-54431-6 978-3-642-54432-3, pp. 155–171 (2014), ISBN 978-3-642-54431-6 978-3-642-54432-3, https://link.springer.com/chapter/10.1007/978-3-642-54432-3_8.

[3] S. A. DeLoach, O-MaSE: An Extensible Methodology for Multi-agent Systems, in *Agent-Oriented Software Engineering*. Springer, Berlin, Heidelberg, ISBN 978-3-642-54431-6 978-3-642-54432-3, pp. 173–191 (2014), ISBN 978-3-642-54431-6 978-3-642-54432-3, https://link.springer.com/chapter/10.1007/978-3-642-54432-3_9.

[4] J. J. Gomez-Sanz, Ten Years of the INGENIAS Methodology, in *Agent-Oriented Software Engineering*. Springer, Berlin, Heidelberg, ISBN 978-3-642-54431-6 978-3-642-54432-3, pp. 193–209 (2014), ISBN 978-3-642-54431-6 978-3-642-54432-3, https://link.springer.com/chapter/10.1007/978-3-642-54432-3_10.

[5] O. Boissier, R. H. Bordini, J. F. Hübner and A. Ricci, Unravelling Multi-agent-Oriented Programming, in *Agent-Oriented Software Engineering*. Springer, Berlin, Heidelberg, ISBN 978-3-642-54431-6 978-3-642-54432-3, pp. 259–272 (2014), ISBN 978-3-642-54431-6 978-3-642-54432-3, https://link.springer.com/chapter/10.1007/978-3-642-54432-3_13.

[6] N. R. Jennings, On agent-based software engineering, *Artificial Intelligence* **117**, 2, pp. 277–296 (2000-03-01), http://www.sciencedirect.com/science/article/pii/S0004370299001071.

[7] A. Sturm and O. Shehory, Agent-Oriented Software Engineering: Revisiting the State of the Art, in *Agent-Oriented Software Engineering*. Springer, Berlin, Heidelberg, ISBN 978-3-642-54431-6 978-3-642-54432-3, pp. 13–26 (2014), ISBN 978-3-642-54431-6 978-3-642-54432-3, https://link.springer.com/chapter/10.1007/978-3-642-54432-3_2.

[8] E. Platon, N. Sabouret and S. Honiden, An architecture for exception management in multiagent systems, *International Journal of Agent-Oriented Software Engineering* **2**, 3, p. 267 (2008), http://www.inderscience.com/link.php?id=19420.

[9] M. Winikoff, Future Directions for Agent-Based Software Engineering, *Int. J. Agent-Oriented Softw. Eng.* **3**, 4, pp. 402–410 (2009), http://dx.doi.org/10.1504/IJAOSE.2009.025319.

[10] 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), http://cres.usc.edu/cgi-bin/print_pub_details.pl?pubid=288.

[11] R. T. Vaughan and B. P. Gerkey, Really Reusable Robot Code and the Player/Stage Project, in Brugali, D. (ed.), *Software Engineering for Experimental Robotics*, p. 24 (2006).

[12] 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 *International Conference on Robotics and Automation*, p. 6 (2009).

[13] D. Brugali, *Software Engineering for Experimental Robotics*. Springer (2007), ISBN 978-3-540-68951-5, google-Books-ID: DEpsCQAAQBAJ.

[14] D. Brugali and P. Scandurra, Component-based robotic engineering (Part I), *IEEE Robotics Automation Magazine* **16**, 4, pp. 84–96 (2009).

[15] D. Brugali and A. Shakhimardanov, Component-Based Robotic Engineering (Part II), *IEEE Robotics Automation Magazine* **17**, 1, pp. 100–112 (2010).

[16] D. Brugali, Model-Driven Software Engineering in Robotics: Models Are Designed to Use the Relevant Things, Thereby Reducing the Complexity and Cost in the Field of Robotics, *IEEE Robotics Automation Magazine* **22**, 3, pp. 155–166 (2015).

[17] D. Calisi, A. Censi, L. Iocchi and D. Nardi, Openrdk: A modular framework for robotic software development, in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1872–1877 (2008).

[18] A. Elkady and T. Sobh, Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography, *Journal of Robotics* **2012**, pp. 1–15 (2012), http://www.hindawi.com/journals/jr/2012/959013/.

[19] E. Tsardoulias and P. Mitkas, Robotic frameworks, architectures and middleware comparison, *arXiv:1711.06842 [cs]* (2017), http://arxiv.org/abs/1711.06842, arXiv: 1711.06842.

[20] M. Montemerlo, N. Roy and S. Thrun, Perspectives on standardization in mobile robot programming: the Carnegie Mellon Navigation (CARMEN) Toolkit, in *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003) (Cat. No 03CH37453)*, Vol. 3, pp. 2436–2441 (2003).

[21] N. T. Dantam, K. B⊘ondergaard, 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).

[22] H. Bruyninckx, Open robot control software: the OROCOS project, in *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No. 01CH37164)*, Vol. 3, pp. 2523–2528 (2001).

[23] N. T. Dantam and M. Stilman, Ach: IPC for Real-Time Robot Control, Technical Report GT-GOLEM-2011-001, Georgia Institute of Technology (2011).
[24] T. J. McCabe, A complexity measure, *IEEE Transactions on Software Engineering* **SE-2**, 4, pp. 308–320 (1976).
[25] M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc. (1977), ISBN 978-0-444-00205-1.
[26] A. J. Albrecht, Measuring application development productivity, in *IBM Applications Development Joint SHARE/GUIDE Symposium*. Monterey, California, pp. 83–92 (1979).
[27] C. Jones, *Applied Software Measurement: Global Analysis of Productivity and Quality*, 3rd edn. McGraw-Hill, New York (2008).
[28] B. W. Boehm, *Software Engineering Economics*, 1st edn. Prentice Hall PTR, Upper Saddle River, NJ, USA (1981), ISBN 0138221227.
[29] S. R. Chidamber and C. F. Kemerer, A metrics suite for object oriented design, *IEEE Transactions on Software Engineering* **20**, 6, pp. 476–493 (1994-06).
[30] R. V. Hudli, C. L. Hoskins and A. V. Hudli, Software metrics for object-oriented designs, in *Proceedings 1994 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 492–495 (1994-10).
[31] 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, pp. 117–144 (2012-11), `http://linkinghub.elsevier.com/retrieve/pii/S0167642312000287`.
[32] N. Fenton and J. Bieman, *Software Metrics: A Rigorous and Practical Approach, Third Edition*. CRC Press (2014-10-01), ISBN 978-1-4398-3823-5, google-Books-ID: lx_OBQAAQBAJ.
[33] 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, p. 7 (2013).
[34] L. B. L. De Souza and M. D. A. Maia, Do software categories impact coupling metrics? in *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13. IEEE Press, ISBN 978-1-4673-2936-1, pp. 217–220 (2013), ISBN 978-1-4673-2936-1, `http://dl.acm.org/citation.cfm?id=2487085.2487128`.
[35] M. Stojkovski, *Thresholds for Software Quality Metrics in Open Source Android Projects*, Master's thesis, NTNU (2017).
[36] 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 *2010 Brazilian Symposium on Software Engineering*. IEEE, ISBN 978-1-4244-8917-6, pp. 11–20 (2010), ISBN 978-1-4244-8917-6, `http://ieeexplore.ieee.org/document/5631691/`.
[37] I. García-Magariño, M. Cossentino and V. Seidita, A Metrics Suite for Evaluating Agent-oriented Architectures, in *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10. ACM, New York, NY,

USA, ISBN 978-1-60558-639-7, pp. 912–919 (2010), ISBN 978-1-60558-639-7, doi:10.1145/1774088.1774278, http://doi.acm.org/10.1145/1774088.1774278, event-place: Sierre, Switzerland.

[38] F. Alonso, J. L. Fuertes, L. Martínez and H. Soza, Measuring the Pro-Activity of Software Agents, *2010 Fifth International Conference on Software Engineering Advances*, pp. 319–324 (2010), doi:10.1109/ICSEA.2010.55.

[39] F. Alonso, J. L. Fuertes, L. Martinez and H. Soza, Towards a set of Measures for Evaluating Software Agent Autonomy, in *2009 Eighth Mexican International Conference on Artificial Intelligence*, pp. 73–78 (2009), doi:10.1109/MICAI.2009.15.

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

[41] G. Jay, J. E. Hale, R. K. Smith, D. Hale, N. A. Kraft and C. Ward, Cyclomatic complexity and lines of code: Empirical evidence of a stable linear relationship, *Journal of Software Engineering and Applications* **02**, 3, pp. 137–143 (2009), http://www.scirp.org/journal/doi.aspx?DOI=10.4236/jsea.2009.23020.