Journal of Computer Sciences and Applications, 2015, Vol. 3, No. 3, 61-66
Available online at http://pubs.sciepub.com/jcsa/3/3/1

DOI:10.12691/jcsa-3-3-1

Transplanting Binary Decision Trees

Eli M. Dow[1,*], Tim Penderghest[2,*]

[1]IBM / Clarkson University, Potsdam NY, USA
[2]Clarkson University, Potsdam NY, USA
*Corresponding author: dowem@clarkson.edu; pendertj@clarkson.edu

Received April 16, 2015; Revised April 29, 2015; Accepted May 04, 2015

Abstract In this paper, we describe a means of compiling binary decision trees as generated by the C4.5 binary decision tree classifier into high-performance, reusable, stand-alone, run-time classifiers. We demonstrate the memory savings and run time characteristics of a compiled tree as compared to the traditional use of a C4.5 runtime. We demonstrate 100% correctness over every input we have available for testing as compared to our own enhanced version of the classic C4.5 run-time classification routine, *consultr*. In addition, this work provides a framework for comparing decision tree classifiers to more in vogue classifiers such as support vector machines as demonstrated within.

Keywords: *binary decision tree, classifier, code-generator, AI*

Cite This Article: Eli M. Dow, and Tim Penderghest, "Transplanting Binary Decision Trees." *Journal of Computer Sciences and Applications*, vol. 3, no. 3 (2015): 61-66. doi: 10.12691/jcsa-3-3-1.

1. Introduction

Decision Tree Classifiers (DTC's) are a commonly used approach for machine learning with application in many diverse areas such as signal classification, remote sensing, medical diagnosis, character recognition, expert systems, and speech recognition. One of the most compelling features of decision tree classifiers is their capability to break down a complex decision-making process into a collection of simpler decisions, thus providing solutions that are easier for human observers to interpret than those solutions derived by support vector machines.

This work presents a novel technique that considers the perspective of application developers who wish to embed decision tree classifier logic in their own programs without having a runtime dependency on the classifier itself. Our main contribution to the art is the presentation and evaluation of a novel system for embedding the decision trees generated by one of the most popular decision tree classifiers, C4.5, into other high-performance applications, thereby increasing the maximum frequency of run-time classifications in a fixed duration.

2. Background

The imitable Ross Quinlan, the author of C4.5, has stated in his own writing, that Carbonell, Michalski and Mitchell [ML] identify three principal dimensions to consider when discussing which machine learning systems:

- learning strategies used by the classification algorithm;
- application domain of the system
- representation of knowledge acquired by the system

This paper addresses the latter consideration in a new way by condensing knowledge representation into its purest and most succinct form.

While decision tree classifiers are not as in vogue as more modern classification approaches such as support vector machines [VAPNIK1][VAPNIK2][LIBSVM], they are still regarded highly for their effectiveness [EFFECTIVENESS] and adaptability to various problem domains.

The C4.5 Binary decision tree classifier is one of the most commonly known binary decision tree classifiers among practitioners and researchers of supervised learning [QUINLAN][IODT]. One strong advantage decision tree classifiers have over support vector machines is that they output plain-text decision trees that can be followed fairly easily even by novices. There is some satisfaction that comes from understanding the decision algorithm by following the decision tree manually, through visual inspection. This is something that is lost when using SVM, which is a powerful tool but encapsulates decision logic in ways that are difficult for many to intuit.

Yet one shortcoming of the C4.5 system goes unaddressed in the literature. While the C4.5 system performs well when the data being classified is static, and during situations where classification times are not incredibly performance sensitive (classifications may take a few seconds or minutes), it is not suitable for use in high frequency or high-volume classification systems. In addition the tool used to make use of the decision tree logic is interactive and operates in human interaction time.

For high performance classifications system that may make hundreds of classifications a second, the status quo is suboptimal. Consider as an example, systems which classify network traffic behaviors for intrusion detection, or other mission critical datacenter tasks. At the extreme, one can envision safety or control systems which need high performance classifiers to make safety decisions involving human beings. These systems may be running on large servers, or on embedded devices.

In light of those considerations, the authors of this paper believe that research into low-overhead, high-performance, runtime classification methods is warranted. Having said that, it makes little sense to attempt to build a completely new classifier when the actual classification algorithms in C4.5 are perfectly suitable. In this work, we present our efforts at encapsulating decision trees, as generated by an unmodified C4.5 instance, into embeddable, high-performance classifiers.

3. The C4.5 Code Generator

In computer science, code generation is the process by which a compiler's code generator converts some intermediate representation of source code into a machine executable format.

Sophisticated compilers typically perform multiple passes over various intermediate forms. This multi-stage process is used because many algorithms for code optimization are easier to apply one at a time, or because the input to one optimization relies on the processing performed by another optimization. This organization also facilitates the creation of a single compiler that can target multiple architectures, as only the last of the code generation stages (the backend) needs to change from target to target.

The input to a code generator typically consists of a parse tree or an abstract syntax tree. A parse tree (also called a concrete syntax tree)is an ordered and rooted tree that represents the syntactic structure of a string instance that belongs to some formal grammar, G of a language L. The tree represents the syntax of the grammar by treating all non-terminals of the grammar as interior nodes, while the terminals of the grammar are treated as leaf nodes. Parse trees are distinct from abstract syntax trees, which are colloquially known as just “syntax trees”, in that the structure and elements that comprise a parse tree more concretely resemble the syntax of the input language L.

The input language of the system described here is the output of the C4.5 binary decision tree classifier, which, after some preprocessing consists solely of a decision tree. The terminals of the grammar consist of strings conforming to the regular expression of the form “(N) | (N/E)” where N and E are both represented by the regular expression “([0-9]+).[0-9])”. The N is the sum of training data values, used to train the classifier, which resulted in that leaf state. While E represents the number of training data values used to train the classifier which, when input into the generated rule represented by the ancestor path through the decision tree, would result in classifications other than the nominated class.

Rather than implementing a machine specific instruction emitter ourselves, we opt instead to generate source code emitter that generate language specific embodiments of the decision tree that can then be compiled into machine instructions. We have implemented emitters for C, C++, and Go programming languages[C][C++][GO]. The C and C++ embodiments compile with the venerable open source c compiler GCC [GCC] while the code emitter by the Go emitter can be compiled with the 6g compiler created by Google.

Table 1. Comparing an unmodified C4.5 rule set to the C source code embodiment of the same logic

Pruned Rule Set As Generated by C4.5	Generated C Language Classification Logic Embodiment

Pruned Rule Set As Generated by C4.5:

```
Final rules from tree 0:
Rule 9:
        synfuels corporation cutback = y
        duty free exports = y
        -> class democrat [97.5%]
Rule 11:
        water project cost sharing = y
        physician fee freeze = u
        -> class democrat [70.7%]
Rule 5:
        physician fee freeze = y
        synfuels corporation cutback = n
        -> class republican [94.8%]
Rule 7:
        physician fee freeze = y
        education spending = y
        duty free exports = n
        -> class republican [94.0%]
Rule 3:
        adoption of the budget resolution = n
        education spending = u
        -> class republican [82.0%]

Rule 13:
        physician fee freeze = u
        mx missile = u
        -> class republican [50.0%]

Default class: democrat
```

Generated C Language Classification Logic Embodiment:

```
char* binary_decision_tree_classifier(char* handicapped_infants,
                                      char* water_project_cost_sharing,
                                      char* adoption_of_the_budget_resolution,
                                      char* physician_fee_freeze,
                                      char* el_salvador_aid,
                                      char* religious_groups_in_schools,
                                      char* anti_satellite_test_ban,
                                      char* aid_to_nicaraguan_contras,
                                      char* mx_missile, char* immigration,
                                      char* synfuels_corporation_cutback,
                                      char* education_spending,
                                      char* superfund_right_to_sue,
                                      char* crime,
                                      char* duty_free_exports,
                                      char* export_administration_act_south_africa)
{
        if ( (strncmp(synfuels_corporation_cutback, "y", strlen("y")) == 0)  &&
             (strncmp(duty_free_exports, "y", strlen("y")) == 0) )
        {
                return "democrat  [97.5%]";
        }
        else if ( (strncmp(water_project_cost_sharing, "y", strlen("y")) == 0)  &&
                  (strncmp(physician_fee_freeze, "u", strlen("u")) == 0) )
        {
                return "democrat  [70.7%]";
        }
        else if ( (strncmp(physician_fee_freeze, "y", strlen("y")) == 0)  &&
                  (strncmp(synfuels_corporation_cutback, "n", strlen("n")) == 0) )
        {
                return "republican  [94.8%]";
        }
        else if ( (strncmp(physician_fee_freeze, "y", strlen("y")) == 0)  &&
                  (strncmp(education_spending, "y", strlen("y")) == 0)  &&
                  (strncmp(duty_free_exports, "n", strlen("n")) == 0) )
        {
                return "republican  [94.0%]";
        }
        else if ( (strncmp(adoption_of_the_budget_resolution, "n", strlen("n")) == 0)  &&
                  (strncmp(education_spending, "u", strlen("u")) == 0) )
        {
                return "republican  [82.0%]";
        }
        else if ( (strncmp(physician_fee_freeze, "u", strlen("u")) == 0)  &&
                  (strncmp(mx_missile, "u", strlen("u")) == 0) )
        {
                return "republican  [50.0%]";
        }

        // We did not hit a C4.5 specified base case, so return the default value…
        return (DEFAULT_CLASSIFICATION);
}
```

An example of a relevant portion of the source code emitted can be seen in Table 1, which compares the decision tree logic from one test case, to the emitted source code (which compiles with strict warnings and errors and should therefore be suitable for incorporation into larger bodies of source code, or used as a stand alone run time classifier). This core logic is encapsulated within a single function whose signature is always of the form binary_decision_tree_classifier() with arguments of type char* corresponding to each of the measured values from a C4.5 data file that are in turn named from the values in the C4.5 names file. The emitted sources also contain code suitable for loading a correspondingly named data file for evaluation as a complete stand-alone program.

The design approach to generate well-formed, high-level language embodiments of the binary decision tree classification function is beneficial in two ways. The first is that this approach allows savvy users, such as the type who are likely to be using C4.5 in the first place, to embed the classification function directly into their own software. The second benefit is that the user may opt to continue processing the generated c-source-code embodiment into a stand-alone high-performance, re-usable, binary classification executable. Both the stand-alone executable and source code are beneficial as they can be distributed to systems that have no instance of C4.5 installed. In addition there are no legal encumbrances on the generated code. This aspect makes our system an attractive option for those who wish to embed logic into larger systems.

3.1. Validating Code Generator Correctness

To ensure the emitters worked properly we downloaded a large collection of machine learning samples made available from the SGI mirror [SGI] of the machine learning data from the University of California at Irvine [UCI].

Though several of the samples needed some adjustment to work with our C4.5 installation, we soon had sample data from over 110 problem domains. Adjustments typically included removing newlines from the middle of comment blocks, which our compiler was not designed to handle.

In order to validate that no errors had been introduced in the classification logic by way of our transformations, we programmatically created an interactive wrapper around the C4.5 *consultr* tool, which is shipped with the C4.5 runtime. The largest drawback of the C4.5 runtime is that is purely interactive and cannot be used in batch mode. The *consultr* tool uses a pruned rule set to interactively prompt a user for values and ultimately makes a classification based on those inputs and the rules. To make the interactive *consltr* tool work in a non-interactive way we used the “empty” library [EMPTY] from a python environment [PYTHON].

We then used each line of input from the training files as input to the interactive *consultr* utility in parallel with the compiled embodiment of the decision tree logic generated by our compiler. We found 100% agreement in classifications for identical inputs across all training data sets we used.

We should note that in order to make our wrapper around the interactive *consultr* tool match the compiled form of the plaintext rules file, we needed to make a one line change in C4.5 rule printing algorithm in order to increase the decimal precision of the numerical values printed into the resulting the rules file by C4.5. We found on particularly enlightening data set that required more precision than the unmodified human readable rules encapsulated. If one were to follow the unmodified rules by hand, to the letter, you would obtain unexpected results. With the precision enhancements in place, after recompiling the more precise rules, 100% agreement was achieved.

3.2. Runtime Performance Analysis

To analyze the runtime performance of our executable classification embodiment, we compare it to the default C4.5 runtime to analyze time-to-classification-decision using wall-clock time.

Before beginning with our performance characterization, we needed to first evaluate whether our decision trees used in testing are generally useful in broader contexts beyond our single data set. By analyzing the results of classifying each data set across the problem domains contained within UCI machine learning repository, we concluded that the average size of decision trees, as measured in lines, is around 41 lines. The median size of all binary classification trees from the same data set was measured to be 34, with a standard deviation of 26. For the purposes of analysis we can say that 34 line rules make a good average case estimation for all data, while some of the larger data sets help us evaluate the empirically measured worst case performance of our system.

Next we analyzed the relationship of training data size to the size of the classifiers decision tree output. As one might expect, there is some evidence to suggest that longer training data sets yield longer decision trees. It is evident from Figure 1 that the relationship is clearly not proportional in nature. This is based on the variation between the shape of the blue line, which represents training data set size, has a shape which is substantially different than the shape indicated by circular points (which indicates the length of the plaintext rules which were generated). We therefore conclude that the relationship between training data size and generated rules is likely governed by other factors that are not yet understood.

As we can see in the same graph, the number of generated rules has very little to do with the size of the resulting stand alone classifier program as generated by our code generator and compiled by GCC. Here we can say that the resulting binary size is, in a small way, affected by the type of data being used inside the logic statements of the decision tree. This is because the code generator causes additional checks, which are necessary for correctness, when individual conditionals test numerical data.

Due to the current code emitter design, numerically heavy files will generate marginally larger executable due to the extra error checking that is used to prevent issues when using the C4.5 “?” value. The “?” value when in C4.5 indicates that a given value’s measurement is not available. As such, during any arithmetic comparison we must special case and check for that the numerical value is not in fact the “?” value, as the inadvertent treatment of

this special "?" value as a float or double type breaks the logic as stated in the rules file. In fact, the rules files do not specifically state what to do when encountering unknown data values, and the proper behavior was only determined through trial and error during regression testing against training data with the *consultr* tool shipped with C4.5.

The relationship between training data set size (measured in lines), the length of the generated decision tree (measured in lines as generated in the pruned rules files created by C4.5, and the file size of the resultant binary capable of encapsulating the classification logic as a stand alone batch style program.

It should further be noted that all source code examples generated during our exhaustive testing of available data have yielded C4.5 rule sets which are in a degenerate type of conjunctive normal form (those boolean statements which are composed of clauses which have been exclusively logically AND'ed together). Strictly speaking, CNF allows clauses to be Boolean literals or disjunctive clauses, but in practice we have only seen the conjunction of Boolean literals. This is highly advantageous as the c embodiment implementing this logic allows for fast short circuit evaluation of statements of this type.

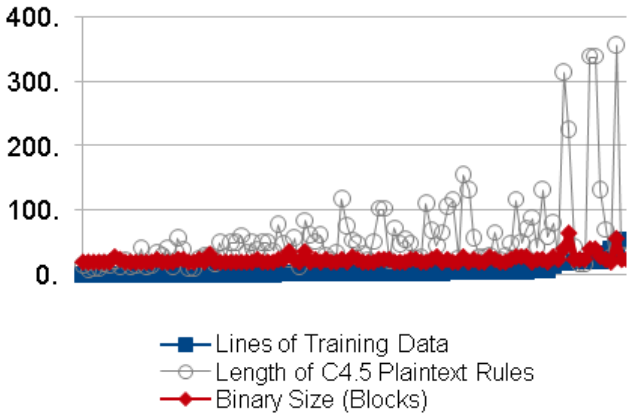

Figure 1. Showing the relationship between training data set size (measured in lines), the length of the generated decision tree (measured in lines as generated in the pruned rule files created by C4.5), and the file size of the resultant binary embodiment of the classification logic as a stand-alone batch style program. The X-Axis of this chart are the training data sets spaced evenly

This allowed us to get a sense of typical sizes of the decision trees generated across these domains. We have concluded that for the data set studied (which we assert is likely to be representative of the set of all possible data used with the C4.5 binary classifier) the average and standard size of the rules generated. We use these figures in our runtime analysis comparison below. All development and experments were performed on a a 4 core AMD server with 8GB of RAM running Ubuntu Linux (Version 11.04).

In addition we applied various techniques to optimize the performance of our executable embodiment such as creating a static binary, altering the GCC compilation optimization levels, and using branch prediction optimizations to increase the time-to-classification, though none of these strategies proved effective enough to warrant the additional work. Since our executable embodiment stands alone, it requires no additional file overhead for reading the names file or data input file.

One potential approach for speeding up C4.5 would be to include a standard input interface to the command line tool which would reduce latency caused by the rotational media I/O required. To attempt to provide the fairest comparison possible, we considered the possibility of showing an additional set of time values for the interactively wrapped C4.5 run time classifier using files backed by a ramdisk that does not suffer from rotational latency, however we realized that this would be overkill when you consider that C4.5 runs in human scale time for classification when using *consultr* without our expect based python wrapper. In addition, any advantage seen by going this far, would be equally advantageous when loading data into our compiled embodiments, and therefore unlikely to prove especially advantageous to either approach.

Furthermore we examined the memory overhead of running concurrent instances of the classifiers in parallel. We examined the cumulative memory usage of N parallel executions of the default C4.5 classifier against the compiled classifier embodiment, as well as a single program that uses N threads, each of which uses the C-source code embodiment as the thread payload directly. As you might expect, parallel invocations of the C4.5 runtime classifier demonstrate the largest cumulative memory usage. This is because the executable size is larger for the C4.5 classifier as compared to the binary size of the stand-alone classification function. In addition, the C4.5 classifier needs a *names* file at run time to describe the second data file which contains the tuple to be classified.

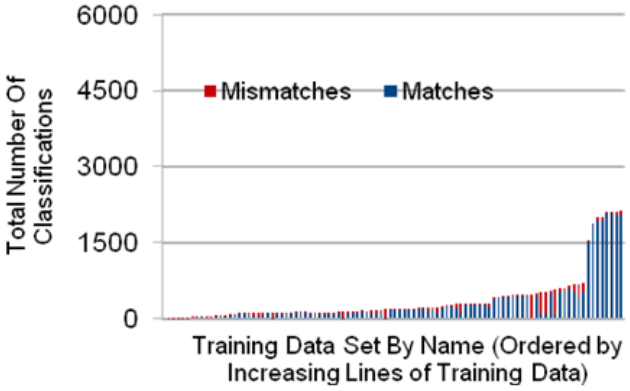

Figure 2. Agreement Between SVM and Compiled C4.5. The blue segments at the bottom of each line indicate agreement in classification for the same data. The red lines that constitute the top of each bar indicate conflicting answers for the same inputs

Now that a rapid, batch, means for evaluating data with a C4.5 derived classifier exists, we set out to compare characteristics of support vector machine derived classifiers as well. We began by building support vector machine models for each of the data files in the C4.5 data set archive. A simple script was written to translated data sets from C4.5 into the data file format needed by libsvm, an open source and freely available support vector machine classifier. Initial results show that while we used a naïve approach to creating numerical only data files (by mapping enumerated types in the C4.5 data into numerical constants for the SVM input file) the libsvm-trained classifiers did fairly well. In fact over 50% of the 104 data sets examined produced better than 90% accuracy with the SVM classifier when tested with all known data values

(for that data set) as test cases on their first run. Furthermore, 67% of the support vector machine models worked in more than 80% of the time during exhaustive testing.

We expect that our work will enable us to further deduce strategies for converting C4.5 data sets into more effective SVM models. We should note, that due to time constraints we were not able to perform any SVM scaling in efforts to boost success of the SVM models.

4. Conclusions

As we can see from the diagrams, our solution effectively embodies the logic generated by C4.5 while outperforming it in terms of memory usage and processor efficiency. We conclude that our solution is an effective tool for systems designers who want to include classification decisions into high performance or massively parallel applications.

5. A Novel Application

In addition to testing this technique on previously constructed data sets, the authors of this paper have constructed a training data set consisting of performance metrics derived from virtual machines running on a cloud hypervisor. As a practical validation of this approach to generating lightweight runtime classifiers that can be easily embedded into other software projects, we replicated and extended an experiment taken from a paper where researchers were able to determine the health and status of a virtual object based solely on measures available externally to that object based on information provided by the hypervisor [Vigilant]. Given the set of data that can be collected is limited in this problem domain, it can be quite difficult to discriminate between ideal and non-ideal behaving virtual objects. The system used in Vigilant monitored the hypervisor (an operating system which multiplexes other operating system instances) resource requests and utilization data collected from a virtual machine instance. A decision tree classier machine learning method was used to analyze the readings at run time and detect problems in situ. The authors note that the choice of decision tree was one of numerous potential classifiers. They selected a decision tree for their work principally because of the training simplicity while also noting that the generated trees are easy to interpret by human observers. They also found their trained tree to be easy to implement with efficient runtime characteristics. They authors did note that decisions trees may not be ideal however. They remark that in terms of classification power, a decision tree is generally considered to be a crude precursor to more modern tools such as support vector machines described in the text “Pattern Classification” by Duda et al [PATCLASS]. They go on to specifically state, that based on their experience, a decision tree is sufficiently powerful for analyzing virtual machine run time metric data. For further information about decision trees and their learning we refer the interested reader to Mitchell’s “Machine Learning” [MITCHELL].

The experimental results from Vigilant show that problems conditions in virtual machines could be detected out-of-band with high accuracy while avoiding the pitfalls associated with in-band monitoring. The efforts in vigilant were specifically targeted at detecting extremely high CPU utilization in kernel space. The metrics analyzed by the Vigilant team include the following:

- The number of times a guest VM was scheduled. Utilization of CPU by the guest VM.
- Time spent in the “runnable” state. Time spent waiting on blocked events.
- Amount of running time allocated to the guest virtual machine by the scheduler. I/O count

These very same metrics seem ideal starting points for detection of virtual machine over-constraint at runtime by a prototype IaaS cloud management system developed by our research group. Interestingly, the authors of the Vigilant paper experimented with several approaches to the problem of out-of-band detection. In one early experiment, they deployed several virtual Linux instances under the QEMU emulator [QEMU] with each running a different type of workload (web service, mail service, etc.) started at various times (to vary the overall load on the host system). In these experiments, the authors were able to classify, using a simple decision tree, the case where the workloads from different machines strain the host machine’s resources, as opposed to the case where only one of the virtual machines was under load. Though they omitted the details of that experiment, they indicated that their approach was applicable in diverse settings.

Our team was able to construct a data set of virtual machines under light-to-moderate load for web servers through the use of HTTP-bench. We then over consolidated the virtual machines onto a common host and were able to collect data at the crossover point where individual http serving virtual machines were experiencing dropped connections or unacceptable latencies in response. Data beyond this point of inflection were collected and tagged as “over-constrained” while the previously collected data under idealized or lightly loaded consolidation levels were tagged as “nominal”.

Using the technique described herein, the data was classified using SVM and C4.5 and converted into a C language embodiment which was further compiled into a shared object and loaded into a runtime over-constraint detection system. Due to the exceptionally lightweight data collection framework devised by our research team along with high performance C embodiment produced by our code generator was able to correctly classify virtual machines as nominal or over-constrained with less than one second delay between increasing the synthetic workload and application alerting on the operator console. Furthermore the system was running continuously while the shared object consisting of the learned decision tree embodiment was dynamically swapped out, at runtime, using an updated instance of the training logic with no negative impact to the running cloud management system. It is worthy of note that during these tests, the C4.5 and SVM trained data sets agreed 100% of the time. The further details of this application and the data collection framework mentioned are to be presented in a forthcoming paper.

6. Future Work

In the near future, the authors intend to apply high frequency runtime classifications to various problems from smarter water sensors to data center optimization. We also see value in extending the emitters to generate other languages in order to make this work more easily consumed by various projects seeking to use machine learning as a component of their software, without learning the training nuances of machine learning.

Just as we have begun to examine head to head comparisons of C4.5 with support vector machines, we intend to extend this work to also incorporate testing of naive Bayesian approaches, Rocchio-style classifiers, k-nearest neighbor methods [KNN] and others as highlighted in the excellent survey by Wu et al [SURVEY] when trained on the same data. This work will provide a framework for integrating other supervised machine learning systems into a cohesive comparative system for novel classification system testing. In addition such a framework would allow an additional level of support when making decisions by comparing the decisions made by multiple independent classifiers and making a final consensus based on some form of quorum (perhaps weighted).

In addition, we would be remiss if we did not mention our desire to attempt this work with the c5.0 decision tree classifier as well, due to the claims of increased performance, which would only benefit our system.

Acknowledgments

The authors of this paper would like to thank the IBM Corporation for generous support of this research. We would also like to thank the Clarkson University Open Source Institute for hosting the development and test servers used for this research.

References

[1] GCC - http://gcc.gnu.org/.

[2] LIBSVM - Chih-Chung Chang and Chih-Jen Lin, LIBSVM : a library for support vector machines. ACM Transactions on Intelligent Systems and Technology, 2:27:1-27:27, 2011. Software available online: http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[3] VAPNIK1 - Vapnik, V. N. (1995). The nature of statistical learning theory. New York: Springer.

[4] VAPNIK2 - Vapnik, V. N. (1998). Statistical learning theory. New York: Wiley.

[5] EFFECTIVENESS - Mahesh Pal, Paul M Mather, An assessment of the effectiveness of decision tree methods for land cover classification, Remote Sensing of Environment, Volume 86, Issue 4, 30 August 2003, Pages 554-565.

[6] QUINLAN - J. Ross Quinlan: C4.5: Programs for Machine Learning Morgan Kaufmann 1993.

[7] IODT - J. R. Quinlan. 1986. Induction of Decision Trees. *Mach. Learn.* 1, 1 (March 1986), 81-106.

[8] C - ANSI X3.159-1989 "Programming Language C.

[9] C++ - C++ standard, 14882:2011.

[10] GO – http://golang.org.

[11] Python – http://www.python.org.

[12] PATCLASS - Pattern Classification, 2nd Edition Richard O. Duda, Peter E. Hart, David G. Stork ISBN: 978-0-471-05669-0 680 pages November 2000, ©2001.

[13] MITCHELL - Machine Learning, Tom Mitchell, McGraw Hill, 1997. http://www.cs.cmu.edu/~tom/mlbook.html.

[14] QEMU - Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '05). USENIX Association, Berkeley, CA, USA, 41-41.

[15] SGI - . http://www.sgi.com/tech/mlc/db/.

[16] UCI - http://archive.ics.uci.edu/ml/.

[17] EMPTY - http://empty.sourceforge.net/.

[18] ML - Ryszard S. Michalski, Jaime G. Carbonell, Tom M. Mitchell (1983), Machine Learning: An Artificial Intelligence Approach, Tioga Publishing Company.

[19] SURVEY - Wu et al. Top 10 algorithms in data mining. Knowledge and Information Systems, 14(2008), 1: 1-37.