Phase Order Search with Feature Selection
João Martins & David Henriques

Group Info

João Martins - jmartins@cs.cmu.edu
David Henriques - dhenriqu@cs.cmu.edu

Project Web Page

`http://www.cs.cmu.edu/~jmartins/15745/`

Abstract

In this work we study the phase-order search problem with a machine learning perspective. Using the LLVM testing suite we 1) gather relevant code statistics from a large number a benchmarks, 2) collapse the phase-order state space in a meaningful way, 3) perform an exhaustive search on the reduced state space and 4) use several well-known ML algorithms to learn phase-orderings that better optimise code with given features. Our results are promising, and show that even in the collapsed state space it is possible to learn orderings that out-perform default compiler optimisations.

1 Introduction

Modern compilers optimise code by running several optimisation passes. Each pass changes the code, producing a semantically equivalent program that (hopefully) performs better in respect to some metric, like time. These changes are *not* independent from each other. Optimisations require certain conditions to be fulfilled and one optimisation may generate or remove oportunities for others to be run. For example, constant propagation replaces some operands by constants, making some variables become unused. A dead code elimination pass may then remove these variables, but only if applied after the constant propagation pass. Finding better orderings for running these passes can have a significant impact in the final performance. This problem is known as the Phase Order Search problem.

One significant problem for the Phase Order Search is the fact that there is an exponential number of orderings of possible passes, which makes an exhaustive search of the orderings unfeasible. Another complication is that there seems to be no one ordering that is best for all programs; specific characteristics make some programs more amenable to some orderings that may not work so well for others. Machine learning techniques have sucessfully been applied to classification problems in large search spaces and, as such, seem likely to be able to help in the phase order search problem. We investigate such approaches in this project.

The goal is to identify characteristics (features) of the code that can then be used to predict good orderings for optimising code with those characteristics in respect to execution time. For this purpose, we use 76 benchmarks from LLVM's benchmark suite. We identify a significant number of such features and find out which among them are most likely be more discriminative through a dimensionality reduction technique known as PCA. After aggregating these features, we train several classifiers to learn a mapping from features to optimal orderings. We used several different learning methods for training classifiers and compare their effectiveness.

[1, 2] address the problem of finding an optimal ordering of optimisation passes, but do not discuss learning techniques. [3] gives an overview of feature selection techniques in general settings. [4] tackles the long runtimes of iterative compilation by using limited heuristics and static performance impact analysis. [5] addresses a similar problem for finding which passes to run to reduce compile time.

2 Machine Learning Algorithms

The task we are trying to perform requires the use of *classification* algorithms, i.e., algorithms which, given training data that associates features with category labels, identify to which category a new observation belongs.

Dimensionality Reduction

Often, the number of features we have to deal with is overwhelming. This not only slows down classification algorithms due to the large number of computations, but it also reduces the intuitive understanding of the algorithm, as it is hard for a human to parse too many features. Principal Components Analysis is a well known feature selection technique used to

reduce the dimensionality of Machine learning problems. It identifies the directions in the feature space which show the greatest variability and are therefore the most likely to impact the performance of the program. This is done by computing the eigenvectors associated with the largest eigenvalues of the covariance matrix of the features. Working only in this reduces feature space often produces more informative results than otherwise. For more details, we refer to [6].

Neural Networks

Neural Networks are a popular classification/regression technique. A neural network consists of a number of input nodes, hidden layers and output nodes. Input nodes are connected to the first level of hidden layers who in turn are connected to the second layer and so on until the last layer is connected to the outputs. Each node receives a set of inputs from the preceding layer and then carries that signal through a tranfer function to the next layer. Information is then aggregated at output nodes and, if it compares unfavourably with the desired output, the error is backpropagated through the network. This way, neural networks of sufficient complexity can learn arbitrary functions. We use MATLAB's nn-toolbox for training neural networks. We kept most settings on default: 10 hidden layers and 15% of the data reserved for validation. For more information on Neural Networks we refer to [7].

SVM

The Support Vector Machine (SVM) is a technique designed to compute the hyperplane which best separates two classes of data in the sense that it maximises distance to any point in either class. In principle, the data must be linearly separable, although this can be circumvented through the use of kernel methods which apply an (invertible) transformation on the data to a space where it can be linearly separated. The method can also be generalised to an arbitrary number of classes by comparing each individual class with all others, either aggregated (one-against-all) or isolated (one-against-one). We use $\text{SVM}^{\text{multiclass}}$, an implementation of support vector machines that performs one-against-all classification and uses Gaussian kernels. For more information on SVM we refer to [8].

K-NN

K-NN is perhaps the most widely used amongst classification algorithms, probably due to its simplicity. Each test point is classified according to the K training points with features most similar to the test point (for some metric, usually Euclidean distance) under a majority voting scheme. We implement K-NN and, due to the simplicity of this method, we ran our experiments for K=1 and for K=5. For more information on K-NN, we refer to [9].

Decision Trees

A decision tree is a predictive model that, for each feature, branches into two subtrees on the remaining features. For classification the features of the test point are used to make decisions sequentially along the decision tree, from root to one leaf, which yields the classification for that test point. Thus, feature ordering is crucial for performance and chosen according to the features that give the most information (reduce enthropy the most) at each decision point. This model greatly benefits from PCA, since information is condensed in less features than it would be without using the dimensionality reduction technique, which makes the tree significantly shorter. We use MATLAB's function `classregtree` to create these trees with dynamic pruning. For more information on Decision Trees, we refer to [10].

3 Experimental Setup

3.1 Overview

We used the LLVM testing suite and infrastructure for this project. From within the SingleSource subset we took 76 benchmarks that worked with our process. These SingleSource benchmarks have a single source file, which makes compiling and feature gathering a lot easier. Furthermore, the testing infrastructure, once we understood how to use it, made automation more accessible, which was almost a necessary pre-requisite for our large benchmark suite.

We first developed several tests (in the LLVM testing infrastructure sense) for gathering statistics about programs. We used a combination of `opt` passes and our own developed passes for this. The next batch of tests ran

our benchmark set with different pass orderings of our choosing, then with `-O3` and with `-O0`. Program features and their runtimes were used as the core of our dataset.

This dataset was imported into MATLAB, and we applied the algorithms mentioned above: Decision Tree, Neural Networks, Five-Nearest Neighbours, One-Nearest Neighbours. Outside of MATLAB we used a Support Vector Machine implementation.

To automate the entire process, we wrote over two dozen shell scripts, python scripts and LLVM tests.

3.2 Benchmarks

We managed to work with 76 benchmarks with a single source file, found within the `test-suite/Singlesource/Benchmarks/` folder. There are more tests within these specifications, but they either failed to compile for some orderings, or failed to run altogether. In fact, four of the benchmarks we used failed to compile/run with the `-O3` and `-O0` flags.

The complete list is the following:

`Adobe-C++/loop_unroll`, `Shootout/ackermann`, `Shootout/ary3`, `Shootout/fib2`, `Shootout/hash`, `Shootout/heapsort`, `Shootout/hello`, `Shootout/lists`, `Shootout/matrix`, `Shootout/methcall`, `Shootout/objinst`, `Shootout/random`, `Shootout/sieve`, `Shootout/strcat`, `McGill/misr`, `Stanford/Bubblesort`, `Stanford/FloatMM`, `Stanford/IntMM`, `Stanford/Oscar`, `Stanford/Perm`, `Stanford/Quicksort`, `Stanford/RealMM`, `Stanford/Towers`, `Stanford/Treesort`, `Misc-C++/Large/ray`, `Misc-C++/Large/sphereflake`, `BenchmarkGame/fannkuch`, `BenchmarkGame/n-body`, `BenchmarkGame/nsieve-bits`, `BenchmarkGame/partialsums`, `BenchmarkGame/puzzle`, `BenchmarkGame/recursive`, `BenchmarkGame/spectral-norm`, `Dhrystone/dry`, `Dhrystone/fldry`, `Misc/dt`, `Misc/fbench`, `Misc/ffbench`, `Misc/flops-1`, `Misc/flops-2`, `Misc/flops-3`, `Misc/flops-4`, `Misc/flops-5`, `Misc/flops-6`, `Misc/flops-7`, `Misc/flops-8`, `Misc/fp-convert`, `Misc/himenobmtxpa`, `Misc/lowercase`, `Misc/mandel`, `Misc/oourafft`, `Misc/perlin`, `Misc/pi`, `Misc/ReedSolomon`, `Misc/salsa20`, `Shootout-C++/ary`, `Shootout-C++/ary2`, `Shootout-C++/ary3`, `Shootout-C++/fibo`, `Shootout-C++/hash`, `Shootout-C++/heapsort`, `Shootout-C++/hello`, `Shootout-C++/matrix`, `Shootout-C++/methcall`, `Shootout-C++/objinst`, `Shootout-C++/random`,

`Shootout-C++/reversefile`, `Shootout-C++/sieve`, `Shootout-C++/spellcheck`, `Shootout-C++/strcat`, `Shootout-C++/sumcol`, `Shootout-C++/wc`, `Misc-C++/bigfib`, `Misc-C++/mandel-text`

3.3 Feature Extraction

To extract relevant features from code, we used the outputs from the `-aa-eval`, `-instcount`, `-lda` and `-regions` passes, as well as of two passes we made that count number of loops, average number of instructions per loop, maximum loop depth and number of basic blocks.

- `-aa-eval` obtains a series of statistics about aliasing in the code.
- `-instcount` gives us the number of instructions of all types appearing in the code. We convert these to percentages to make it easier to classify large and small code-bases together.
- `-lda` looks at memory access dependencies between loops.
- `-regions` gives us information about code regions, which are zones of code whose CFG only have one entry and one exit. This is a good measure of how isolated code is.

All these statistics are obtained using LLVM tests and using regular expressions to extract the relevant numbers. They are then compiled into `.csv` files to be imported and used in MATLAB and Excel.

3.4 Orderings

The phase-order space is far too large to explore completely. For example, `opt` itself provides well over one hundred different analysis and optimisation passes. To reduce this, we decided to classify the optimisation passes according to the type of actions they execute on the code. By reading through the LLVM documentation, we identified four main classes.

- Code modification: passes in this class alter existing code in varying ways, but do not move or remove it. The ordering for this class is: `-argpromotion`, `-constmerge`, `-constprop`, `-globalopt`, `-ipsccp`, `-correlated-propagation`, `-ipconstprop`, `-lower-expect`, `-loweratomic`, `-lowerinvoke`, `-lowerswitch`, `-mem2reg`, `-prune-eh`,

`-mergereturn`, `-reassociate`, `-reg2mem`, `-scalarrepl`, `-sccp`, `-scalarrepl-ssa`, `-simplify-libcalls`, `-tailcallelim`

- Code motion: these passes effect modifications by moving instructions from one place to another. The ordering for this class is: `-block-placement`, `-always-inline`, `-inline`, `-jump-threading`, `-partial-inliner`, `-sink`
- Code elimination: elimination passes are self-descriptive! The ordering for this class is: `-adce`, `-dce`, `-deadargelim`, `-die`, `-dse`, `-early-cse`, `-globaldce`, `-gvn`, `-instcombine`, `-instsimplify`, `-memcpyopt`, `-mergefunc`, `-simplifycfg`
- Loop-related optimisations: some passes reason only about loops. Since loops are extremely important and are executed very often by their very nature, we felt these optimisations deserved a class of their own. The ordering for this class is: `-indvars`, `-lcssa`, `-licm`, `-loop-idiom`, `-loop-deletion`, `-loop-extract`, `-loop-extract-single`, `-loop-instsimplify`, `-loop-reduce`, `-loop-rotate`, `-loop-unroll`, `-loop-simplify`, `-loop-unswitch`

With only four classes to consider, it is now computationally feasible to extensively test and run with all twenty-four possible *class* orderings. However, the individual pass ordering within these classes is still fixed.

3.5 Compilation

We configured the test infrastructure to use `clang` as the default compiler, along with the LLVM toolset (e.g. `llvm-ld`, `llvm-as`, ...), and set the default optimisation flag to `-O0`. The test infrastructure, in practise, compiles each benchmark in the following manner:

1. `clang $(CFLAGS) -O0 -S $< -o $@ -emit-llvm`. This compiles to `.ll` files from `.c` or `.cpp` (by using the `clang++` compiler and the additional `$(CPPFLAGS)` option) files.
2. `llvm-as $< -o $@`. This assembles `.ll` files into `.bc` files.

After this, our tests optimise the byte-code in the following way:

3. `opt $(PASSES) $< -o $(TEST_NAME).opt.o`. This optimises the code with the pass ordering given by `$(PASSES)`, which is obtained from a text file. This test is called by a script that iterates all orderings and changes the pass text file.

4. `llvm-ld -disable-opt $(TEST_NAME).opt.o -o=$@`. This instruction links the byte-code into the final executable. The `-disable-opt` flag is used to disable link-time optimisations and make sure our orderings have maximum impact.

The exact same procedure is applied to the `-O3` and `-O0` optimisation passes. We are not compiling directly with `clang -O3`, but are going through an unoptimised compilation, then running `opt -O3`.

Finally, the tests are run and timed, and the results put into `.csv` files to be used in MATLAB and Excel.

4 Experimental Results

In this section, we present the experimental results of the procedures described in the previous sections. For each Machine Learning method, for each of the 76 benchmarks, we trained the classifier in the other 75 examples and then evaluated it in the remaining benchmark, as in leave-one-out cross validation. This makes our results a lot more robust than by simply dividing them once into training and testing datasets.

Dimensionality Reduction

PCA reduced the dimensionality of the feature space to essentially 4 feature vectors since eigenvalues smaller than the largest 4 had much smaller magnitude than the 4 largest eigenvalues. Analysing the eigenvectors we see that the features with the most influence were `% Call Instructions`, `% Load Instructions`, `# Basic Blocks`, `# Regions`, `# Simple Regions`, `# Loops`.

This is a diverse set of features that we think covers the broad spectrum of program characteristics that are known to be runtime bottlenecks such as excessive memory access (load instructions), unoptimised code in loops (number of loops), function call overhead (call instructions), and how compartmentalised the code is (number of regions and basic blocks).

Comparison Against Best Ordering in Domain

The learning algorithms, despite having very high ($> 95\%$) error rates in identifying the *best* possible ordering (out of the total 24), chose orderings that were significantly close to the best in terms of runtime. On average, the ML methods learned orderings that achieved execution times between 84% (SVM) and 90% (Neural Network) of the best possible ordering. We present 7 indicative cases chosen semi-randomly (we wanted to have some cases where we fared well and some where we fared poorly, but chose randomly within those classes) in Figure 1.

It is also worth noting that some benchmarks had extremely small runtimes (< 50ms), making even tiny variations have a much bigger impact on the average.

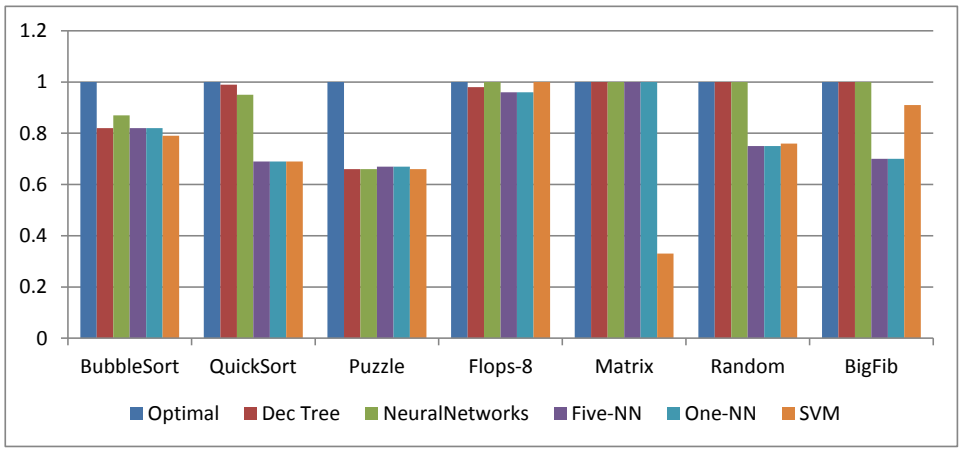

Figure 1: Comparison against best possible ordering. Showing (optimal time / learned ordering time). Higher is better.

Comparison Against Unoptimised Code

When comparing against unoptimised code, the learning algorithms, unsurprisingly, fare significantly better. On average, methods run more than twice as fast as the unoptimised code. It is interesting to notice the high variability of the results (Figure 2), indicating that some programs are very hard to optimise while some others are very amenable to optimisation.

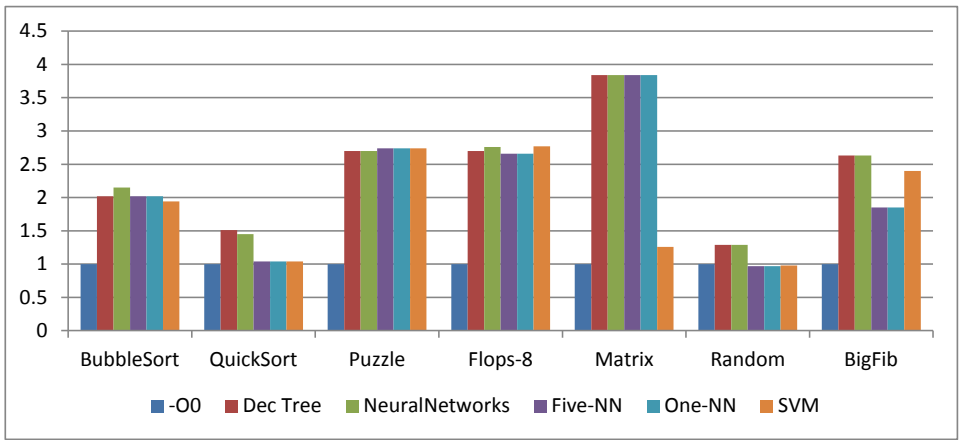

Figure 2: Comparison against non optimised code. Showing (non-optimised time / learned ordering time). Higher is better.

Comparison Against `-O3` optimised Code

When comparing against `-O3` optimised code, it is worth noticing that, while on average the learned orderings are not as good as `-O3` (ranging from one-NN at 85% to Neural Network at 91%), *there are instances where the orderings found by learning algorithms fare better than* `-O3`, which at least suggests that `-O3` ordering can be improved for certain code features. Some cases can be seen in Figure 3.

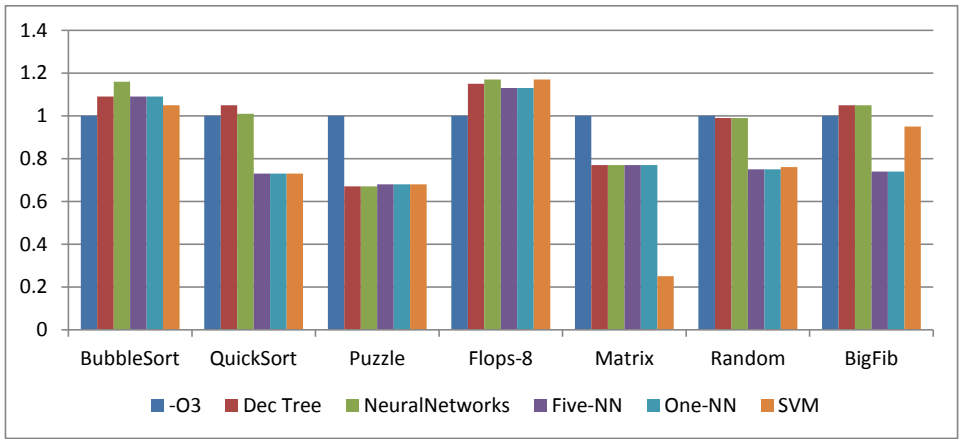

Figure 3: Comparison against `-O3` optimised code. Showing (`-O3`-optimised time / learned ordering time). Higher is better.

Best and Worst Cases

In order to illustrate the high variability of the learning techniques, we analyse the best and worst case benchmarks. In Figure 4 we can see that almost all of the methods do significantly better than even `-O3` (the axis scale is logarithmic). On the other hand, in Figure 5, we see that the machine learning algorithm's performance is ghastly in this case (once again, the scale is logarithmic). These are limit cases, but it's still interesting to notice their existence.

5 Conclusions

In this work we handled the phase-order search problem. We considered a manageable version of the state space by grouping together passes whose effect on the code is similar. In this reduced search space, using features extracted from the code, we were able to train five different classifiers that were, in some cases, able to outperform the `-O3` default optimisations. This

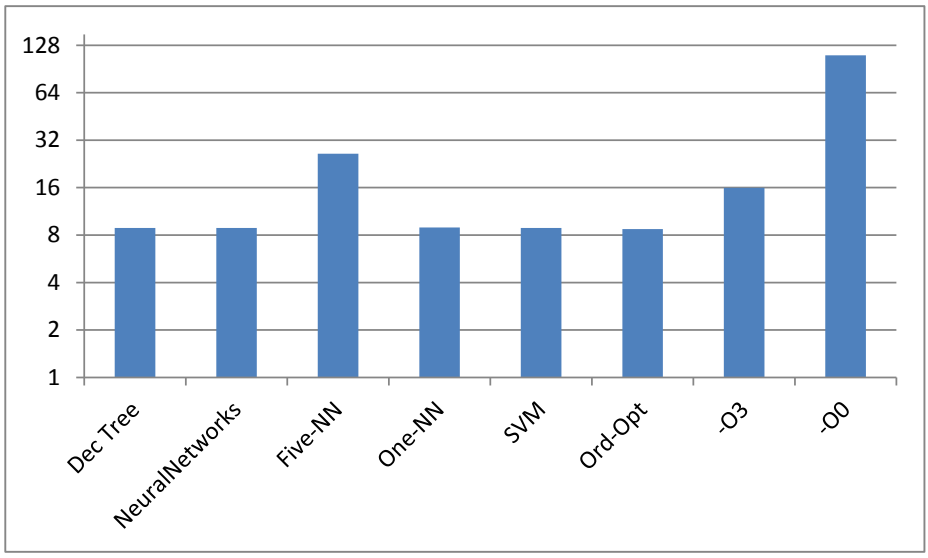

Figure 4: Best case benchmark: loop_unrolling benchmark. Shown is total time taken (s). Lower is better.

is impressive given that no within-class optimisations have been attempted.

Other interesting conclusions from this work were that the best classifiers assigned very uniform labels, but that there were crucial differences on some benchmarks that deserved specific labels, and that the vast majority of test-cases can be classified by looking at an extremely reduced number of features.

References

[1] Agakov, F. and Bonilla, E. and Cavazos, J. and Franke, B. and Fursin, G. and O'Boyle, M. F. P. and Thomson, J. and Toussaint, M. and Williams, C. K. I. Using Machine Learning to Focus Iterative optimisation. In *Proceedings of the International Symposium on Code Generation and optimisation (CGO, pages 295-305)*, 2006.

[2] Prasad A. Kulkarni and David B. Whalley and Gary S. Tyson. Evaluating heuristic optimisation phase order search algorithms. In *Proceedings of the International Symposium on Code Generation and optimisation (CGO, pages 157-169)*, 2007.

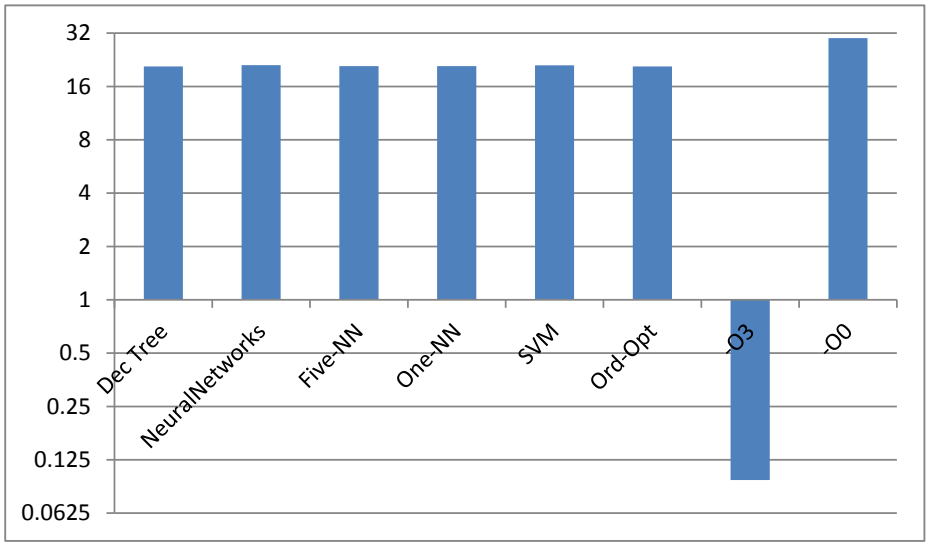

Figure 5: Worst case benchmark: objinst benchmark. Shown is total time taken (s). Lower is better.

[3] Andrew Y. Ng. On Feature selection: Learning with Exponentially many Irrelevant Features as Training Examples. In *Proceedings of the Fifteenth International Conference on Machine Learning (pages 402-414)*. 1998.

[4] Triantafyllis, Spyridon and Vachharajani, Manish and Vachharajani, Neil and August, David I. Compiler optimisation-space exploration. In *Proceedings of the international symposium on Code generation and optimisation: feedback-directed and runtime optimisation (CGO, pages 204-215)*. 2005.

[5] Gennady Pekhimenko. Machine learning algorithms for choosing compiler heuristics, 2008.

[6] Jolliffe, I. T. (1986). Principal Component Analysis. Springer-Verlag. pp. 487. doi:10.1007/b98835. ISBN 978-0-387-95442-4.

[7] Christopher Bishop. Neural Networks for Pattern Recognition. Oxford University Press, USA, 1st edition, January 1996.

[8] Nello Cristianini and John Shawe-Taylor, An introduction to sup- port vector machines and other kernel-based learning methods, Cambridge University Press, 2000.

[9] Belur V. Dasarathy, ed. (1991). Nearest Neighbor (NN) Norms: NN Pattern Classification Techniques. ISBN 0-8186-8930-7.

[10] J. R. Quinlan, Induction of decision trees, Mach. Learn. 1 (1986), 81-106.