Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

MASTERTHESIS

Optimising Scientific Software for Heterogeneous Cluster Computers: Evaluation of Machine Learning Methods for Source Code Classification

vorgelegt von

Ruben Felgenhauer

MIN-Fakultät

Fachbereich Informatik

Studiengang: Informatik

Matrikelnummer: 6659393

Erstgutachter: Prof. Dr. Thomas Ludwig

Zweitgutachter: Prof. Dr. Peter Hauschildt

Betreuer: Jannek Squar, Prof. Dr. Peter Hauschildt

Hamburg, den 3. August 2021

Abstract

Since high performance compute centres are shifting more to use accelerators like GPUs, vector processors, or many-core CPUs, HPC programmers are often confronted with a very heterogeneous hardware environment. Different computation units have different requirements in order to use them most efficiently. Typically, scientific software is optimised for specific target architectures based upon decisions that are made before it is known which hardware composition will be available at the time of running. This can lead to cluster computers being used under capacity which wastes computational resources and energy.

With the evolution and resulting gain in popularity of automatic parallelisation tools like OpenMP and sophisticated static code analysis methods, source code can increasingly be written in a more readable fashion with acceptable performance cuts. Therefore, given the choice between performance and maintainability, it can increasingly be made in favour of the latter.

However, at the time of writing, this only goes so far that the programmer decides *which* sections to parallelise and optimise *for which target architecture*. Ultimately, to efficiently tackle cluster heterogeneity, the goal should be to automatically find the optimal mapping of sub-programs to computation units and performing the required parallelisation. Central to this task is source code classification.

It has been shown by Barchi et al. that machine learning classification can be used to determine a suitable computation unit for OpenCL source code samples. In this thesis, we evaluate machine learning methods for general-purpose classification of program parts to extend this principle to support all ahead-of-time compiled programming languages. First, we combine the ASTNN by Zhang et al., a neural network architecture that can be used for the classification of C source code samples, with RetDec, a retargetable decompiler by Křoustek (Avast Software) based on LLVM which generates C code from machine instructions.

This is compared with a straight-forward approach where general-purpose classifiers from `scikit-learn` are trained on byte-value data of the object code files' `.text` section. We show that the modified ASTNN outperforms these methods in all of our performed benchmarks, but that it comes with several limitations including high memory consumption and training time, and unresponsiveness of the decompiler on some samples.

Acknowledgement

Als erstes möchte ich mich bei meinen Gutachtern und Betreuern Prof. Dr. Thomas Ludwig, Prof. Dr. Peter Hauschildt, und Jannek Squar bedanken. Danke, Peter, dass Du diese Arbeit betreut hast, obwohl, anders als ursprünglich geplant, der Physik-Anteil nun leider auf Null gesunken ist! Danke, Jannek, dass Du mir in zahlreichen Telefonkonferenzen geholfen hast, auf der richtigen Spur zu bleiben und für Deine wirklich extrem ausführlichen Anmerkungen!

Weiter möchte ich mich bei den Mitarbeitern des Studienbüros Informatik bedanken, die mich bei allen Aspekten meines Studiums immer freundlich und zügig beraten und unterstützt haben, und hierbei ganz besonders bei Dagmar Schacht, die mich im Bachelor-Studium am Ende des ersten Semesters davon abgehalten hat, das Studium abzubrechen, und ohne die diese Arbeit deshalb wohl nicht entstanden wäre.

Außerdem möchte ich mich natürlich bei meiner Freundin Lina Meyer bedanken. Danke, Lina, dass Du mir geholfen hast, während der Heimarbeit, die durch die anhaltende SARS-CoV-2-Pandemie erzwungen wurde, die Nerven zu behalten, und dass Du mich Tag für Tag dazu motiviert hast, mich an den Schreibtisch zu setzen und kreativ zu sein!

Schließlich möchte ich mich bei meinen Eltern für ihre Unterstützung bei dieser Arbeit bedanken und dass sie mir geholfen haben, soweit zu kommen, wie ich jetzt bin, und bei meinen Freunden Fabian Schmidt, Jasper Gerwers, Karim Ritter von Merkl, Laurence Grossen, Leonhard Reichenbach, Michaela Schneeberger, Simon Welker, Theresa Cavasin, Tim Kuhrt, und Timon Schümann. Danke, dass ihr während meines Studiums immer für mich da wart!

Contents

Appendices

1

Chapter 1.

Introduction

There is an intricate struggle inside the High Performance Computing (HPC) community that is more present than in other fields of applied computer science. At some point, every HPC programmer will have to make a choice between *readability* and *performance*. Basically, this boils down to a balance between the *runtime* of an application and the *working time* a programmer has to put into it. However, while the runtime of a program is easily measurable, the complete workload that is induced by a deliberate design decision is often unpredictable. The reason for this is quite simple:

> *"It's harder to read code than to write it."*
>
> – Joel Spolsky [1]

A programmer may easily be able to put in an hour of work into his program, replacing well-readable high-level constructs with machine-oriented ones, reducing the runtime by 5 minutes per run, which makes it seem like his work has amortised after 12 runs (a small number for most applications), but it would be a fallacy to unconditionally call this a success, because the subsequent work is not incorporated in the final balance. If later, a different developer joined the team to implement new features, fix bugs, or inherit the code base for maintenance, this gain of time would quickly diminish if they were confronted with an unreadable code base. This has led many members of the HPC community to give a lot more weight to readability than to small runtime advantages.

> *“Premature optimisation is the root of all evil.”*
>
> – Donald E. Knuth [2]

Tools like OpenMP (see Section 3.2.1) have helped reducing the stress on programmers by offering parallelisation that can often be achieved through the addition of very few lines (compiler pragmas) to an existing iterative program. This does not only save time while writing in comparison to parallelisation via POSIX threads, where extensive modification of the source code and the used datastructures is often necessary, but also strongly helps keeping source code more readable.

Since the speed-up per workload for the programmer that is achieved by using OpenMP is comparatively large, one can argue that a stronger focus on tool assistance is generally desirable.

This thesis takes this assumption to the next logical step by researching methods that could finally enable us to even banish preprocessor pragmas from the lives of the average HPC programmer, pushing all necessary work to utilise a heterogeneous set of computational resources as best as possible to the compiler. This could be done by automatically deciding the optimal target architecture for parts of a program.

In the following, we denote a large-scale set of individual computers (nodes) that can be addressed in a way so that they can be used to solve a large computational problem simultaneously through joint effort as a *cluster computer*. We say that a cluster computer is *heterogeneous*, when the nodes employ drastically different types of processing units.

Apart from traditional Server CPUs with up to 64 cores (128 threads), this can mean general-purpose computing on graphics processing units (GPGPUs) with thousands of cores, vector processors / Vector Engines with approximately 10 cores, or many-core processors with up to 72 cores (288 threads).

All of the above systems can be found in high performance systems and require different methods to fully make use of their computational power, thus, different classes of algorithms may be suited differently for optimisation on the individual platforms. A description of these classes and code patterns is given in Section 3.3.

Apart from OpenMP which is usually used to acquire parallelism on the CPU, OpenACC can be used to automatically run code on GPUs in a similar fashion, and building for vector processors is usually done by using a dedicated compiler. A detailed rundown of these and similar tools can be found in Section 3.2.

Three vastly different examples for clusters with a heterogeneous structure are the “Cori” Cluster at the National Energy Research Scientific Computing Center (NERSC) in Berkeley, California, which consists of 2,388 nodes with only (traditional) *Intel Xeon* CPUs and 9,688 nodes with *Intel Xeon Phi* many-core CPUs [3]; the “Aspire 1” cluster at the National Supercomputing Centre (NSCC) in Singapore which has 1,288 nodes with only *Intel Xeon* CPUs and 128 nodes with additional *NVIDIA* GPUs [4]; or the “Mistral” cluster at the German Climate Computing Center (Deutsches Klimarechenzentrum, DKRZ) in Hamburg, which consists of 3,359 nodes with only *Intel Xeon* CPUs, and 21 nodes with additional *NVIDIA* GPUs [5], although the system has also featured a handful of *NEC SX-Aurora* Vector Engines in internal testing systems in the past.

The user base of a typical scientific software is usually a superset of the team of its developers, thus it gets used by people that generally do not know much about its internal structure. Therefore, a widespread behaviour of these users is simply condoning the limitations of the software that they are confronted with. If the application is optimised to only run on CPUs, the user will probably only allocate nodes dedicated to CPU computation, while other nodes of the cluster (e.g. with GPUs) may lie idle, i.e. the computational resources get adjusted to the program’s capabilities which can lead to a waste of potential computational power.

On the other hand, if we manage to adjust the program to the available cluster, one could utilise the cluster in its entirety (if desired). Through tool assistance, this could be done by the user so that the distribution of the individual processes to the heterogeneous hardware is simplified to yet another parameter like the number of processors used. In this scenario, a programmer would write iterative high-level code that is independent of the type of processors and gets specialised only at compile-time.

There is a multitude of research projects that engage in static source code analysis with the goal of automatic parallelisation even without the help of compiler pragmas. For example, Polly, which has by now been integrated into the LLVM framework, attempts to locate relevant loops and to automatically create parallel or GPU optimised code. Based upon that, the tool Molly tries to assist in the creation of MPI code for computers with distributed memory. However, these tools usually require the user to know a priori for which architecture to optimise. At the time of writing, relatively little research has been done in automatically *deciding* for which platform a piece of code is best suited for.

Crucial to this idea is the task of *source code classification*. As has been shown by Barchi et al. [6], source code samples can be grouped into classes which go along with an optimal computation unit. In Chapter 3, the theoretical background of this thesis is explained. This includes the types of hardware that are used in HPC environments (Section 3.1) and modern methods to create programs for them (Section 3.2), developments in code pattern analysis (Section 3.3) and programming languages in HPC (Section 3.4), tools for creation of machine code from high-level source code (Section 3.5) and vice versa (Section 3.6), and descriptions of the basic machine learning methods that were used in this thesis (Sections 3.8 to 3.11). In Chapter 4, we will see concrete examples for source code classifiers that can be used in a scientific HPC context. They are then evaluated through benchmarks (Section 4.3). Finally, in Chapter 5 we conclude this thesis by first giving a qualitative comparison of the methods (Section 5.1), listing steps to improve their performance in general (Section 5.2.1), and finally proposing a hypothetical model based upon the used classifiers that could be used for automatic code mapping (Section 5.2.2).

2

Chapter 2.

Related Work

In 2006, Asanović et al. listed several classes of programs which are characteristic for their computation behaviour and data movement which they titled the “Seven Dwarfs” [7]. They based this list on the work of Colella from 2004, where these patterns were described in detail [8]. Furthermore, Asanović et al. listed six further classes of algorithms that they stated were missing from Colella’s work.

There have been multiple projects to automatically parallelise code with the aid of the programmer. The OpenMP compiler extension is able to produce multi-threaded machine code for loops that have been annotated by the programmer [9, 10]. OpenMP is supported by compilers like `gcc` or `clang` and widely adopted. Likewise, OpenACC allows creation of code for graphics processing units [11]. On a lower level, the OpenCL language and library are generally able to target different processing units, although it usually gets used to create programs for GPUs [12]. The Kokkos project offers an abstract platform that enables programmers to run software portably on a variety of platforms [13]. It uses some of the tools listed in Section 3.2 as a back-end to automatically parallelise code. While it makes it possible that programmers write their applications only once and distribute them on multiple platforms, one might find that code that uses Kokkos is marginally harder to read than sequential code. Furthermore, a disadvantage of Kokkos is that it only supports C++.

Grosser et al. created the tool Polly which analyses loops inside programs to automatically parallelise them [14]. They state that work has also been done for the task of GPU code production. Based on this, Kruse created the tool Molly, which offers semi-automatic parallelisation via message passing (MPI) [15]. Work has also been done by Squar et al. to transform code that has been parallelised with OpenMP to MPI [16]. In 2019, Barchi et al. proposed machine learning methods that were trained on OpenCL kernels which were compiled to LLVM IR for automatic source code mapping to suitable processing units [6]. In their tests, this method was able to predict the optimal computation unit with an accuracy of 85 %.

There have been numerous different approaches to perform static code analysis via machine learning. McDaniel and Heydari have shown in 2003 that a lot of information can already be extracted from binary files using statistical analysis of byte-value histograms [17]. In their work, they used their method to recognize the file type for files with missing headers, but they state that it could also be used for virus scanning or forensic hard drive analysis. In 2015, Clemens took a similar approach to infer the target architecture and endianess from executables containing unspecified machine instructions [18].

There are several models that were based on the abstract syntax trees (AST) of source code: In 2015, Mou et al. proposed a tree-based convolutional neural network (TBCNN) for source code classification which worked on the abstract syntax tree of source code samples [19]. They created a dataset containing labelled source code samples and used this for benchmarks in which their method achieved an accuracy of 94.0 %. The method was outperformed by a model called ASTNN which was proposed by Zhang et al. in 2019 [20]. By splitting up the AST into smaller pieces called statement trees, it reached an accuracy of 98.2 % on the benchmark from Mou et al. In 2018, Dam et al. proposed a deep model based on long short term memory networks (LSTMs) which they used for software defect prediction [21]. A disadvantage of these models is that they are fixated on the classification of samples from a specific programming language during training, since the structure of ASTs varies drastically from language to language.

To generate high-level source code from machine instructions, a decompiler can be used. A common challenge of these tools is to reconstruct the original source code as true to the original as possible. In this thesis, we use the RetDec decompiler from Křoustek [22] which often generates well-readable code and is available under the permissive MIT license.

3

Chapter 3.

Background

3.1. Hardware Evolution in HPC

According to an article that was published inside the blog insideHPC in 2016, the evolution of HPC can be split into three epochs: the progression of supercomputers from singular machines like the Cray-1 in 1976 [23] to modern distributed systems that consist of millions of processor cores located on numerous individual nodes, a development towards increasingly more processor cores per node (therein called the "Multi and Many-core Explosion"), and finally, a more involved development process through hardware-software co-design [24].

The first two claims can be backed by the biannually published TOP500 list, a list of the 500 fastest supercomputers in the world [25]: From 2011 to 2020, the maximum number of overall processor cores per supercomputer has increased from 705,024 to 7,630,848, while the proportion of clusters that are listed to feature accelerators has increased from 7.8 % (39 systems) to 34.8 % (174 systems) [26].

The most popular kind of accelerators are *general-purpose computing on graphics processing units* (GPGPUs). At the time of writing, these systems feature thousands of cores and up to 11.5 teraFLOPs of double precision computing capacity [27][1]. This outstanding performance is achieved through high parallelism and memory-bandwidth. Therefore, GPUs excel when they are performing uniform operations

---

[1] Compare to a traditional CPU like the "AMD Ryzen™ Threadripper™ 3990X" which was benchmarked with 1.571 teraFLOPs in the HPL benchmark [28].

on large amounts of data, e.g. matrix multiplications [29]. However, this also comes with a significant downside: Since GPU computing has originated from GPU shaders used for 3D visualisation, it is to this day strongly linked to libraries like CUDA [30] or OpenCL [12] that make the processing power usable by the programmer in a more convenient manner. It is also possible to reduce the programmer's workload through the framework OpenACC (see Section 3.2.2) which works with compiler pragmas (see Section 3.5.2) similarly to OpenMP (see Section 3.2.1) or OpenMP with accelerator offloading. However, Kirk and Hwu claim that this approach will often not produce optimal results or require further steps. Furthermore, GPUs are not suited for all kinds of tasks and will only deliver their theoretical peak power on highly parallel problems [29].

The second variant of accelerators are *many-core processors* which are almost exclusively built by Intel and sold as *Intel Xeon Phi* which feature up to 72 cores [31] and made up only 0.8 % of the TOP500 systems (4 systems) in 2020 [26]. The individual Xeon Phi models have a large diversity: While initial designs which were codenamed *Larrabee* [32] and later sold as *Knights Ferry* [33] were classified as GPUs and to be used as PCIe extension cards which still required a CPU to be present on the mainboard, more recent models from the *Knights Landing* [34] and *Knights Mill* [31] generation can be socketed directly onto the mainboard and also be used to run the operating system, so these models can not be properly classified as accelerators. In fact, they are instead marketed as *coprocessors* by Intel. These models consist of modified Airmont [35] cores with a comparatively low clock rate of up to 1.7 GHz which were originally used for low-power mobile Atom architectures, but have been equipped with AVX-512 vector units, 16 GB of MCDRAM instead of L3 cache, and fourfold hyperthreading (288 CPU threads in total). Therefore, while Xeon Phi processors have a smaller computation capacity compared to high-end GPUs[2], they can be easier to use in practice, since programs written for x86 CPUs can exploit their power without changes, especially if they make use of AVX-512 instructions.

Finally, a type of accelerator that is present in 0.4 % (2 systems) of the TOP500 list are *Vector Processors* or *Vector Engines*. Both of these systems are *SX-Aurora TSUBASA* processors by NEC, and although they are executed as PCIe cards and need a host CPU that runs the operating system, they are listed as processors rather than accelerators in the TOP500 list which can be considered inconsistent in regards to GPUs. A reason for this could be that one considers the workload on the host CPU as negligible on a system that is accelerated by a vector processor.

---

[2] The Xeon Phi 7290 has 3.456 teraFLOPS of double precision computation power [34].

NEC's vector engines have up to 3.07 teraFLOPs of computing power [36] which is roughly comparable to the performance of Intel Xeon Phis, but with only 10 processor cores instead of 72. However, it is not uncommon to use several vector processors in one machine as performed in a cluster by the German Meteorological Service (Deutscher Wetterdienst, DWD) [37]. The source of this high processing power lies, as the name suggests, in a consequent exploitation of vectorisation, following the Single Instruction Multiple Data (SIMD) paradigm. To ensure this, NEC offers compilers for C, C++ and Fortran which allow for efficient automatic vectorisation and support for OpenMP [38]. It is also possible to explicitly optimise written code to be more suitable for automatic vectorisation.

In summary, there are multiple kinds of accelerators that are used in HPC clusters. Although tools exist that help the programmer to optimise their code for certain architectures, this process is usually still associated with deliberate design decisions that simplify or even facilitate these optimisations, i.e. it is not trivial to write code that will run well on traditional or many-core CPUs, GPUs, and vector processors, and an attempt to do so can lead to code duplication. In the next section, we will give an overview over these tools and frameworks, and in Section 5.2.2, we will present a hypothetical solution to this problem based on the methods that are discussed in this thesis.

3.2. Automatic Parallelisation Tools

3.2.1. OpenMP

OpenMP (Open Multiprocessing) is an application programming interface (API) which simplifies the parallelisation of programs written in C, C++, or Fortran for machines with shared memory [39] through the usage of compiler pragmas (see Section 3.5.2). This means that OpenMP can only be used for parallelisation across machine boundaries, and therefore not for programs that are distributed on cluster computers, where libraries like MPI (see Section 3.2.4) may be used instead[3]. Prior to OpenMP's first release in 1997 (Fortran interface) [9] respectively 1998 (C/C++ interface) [10], programmers were bound to existing, usually more verbose, methods like POSIX threads. Listing A.1 shows a simple C program where an (abridged) computationally expensive function gets called multiple times sequentially in a

---

[3] It is however possible to use a hybrid approach where one uses MPI and OpenMP together to achieve thread-level parallelism with message passing.

loop. A straightforward parallelisation of this program using POSIX threads can be found in Listing A.2. As one can see, this snippet is significantly longer, and, since the number of threads and the number of tasks is not equal, arguably a lot harder to comprehend. In comparison, the parallelisation using OpenMP which can be found in Listing A.3 is almost identical. The compiler pragma in line 13 is informing the compiler to parallelise the following loop. Therefore, OpenMP enables a programmer to parallelise their applications very quickly and to keep them comparatively readable.

In the next section, the API OpenACC will be described which assists the programmer in running programs on GPUs and has a similar syntax to OpenMP. Since 2013, OpenMP also supports accelerator offloading [40] which can be understood as an alternative to OpenACC. However, with both OpenMP and OpenACC, the programmer has to make a decision to select specific loops that are to be parallelised. Section 3.2.3 describes the tool Polly which can help to automatically identify them.

3.2.2. OpenACC

Similar to OpenMP, OpenACC (Open Accelerators) is a library for automatic parallelisation of sequential source code on GPUs which also has a fairly similar syntax: The analogue to OpenMP's `#pragma omp parallel` (see Listing A.3) in OpenACC would be `"#pragma acc parallel"`. Furthermore, OpenACC has several features that are specific to GPUs, e.g. for memory management, because GPUs usually have their own dedicated memory [11]. Since automatic parallelisation with OpenACC will not always lead to optimal results [29], tools like OpenCL [12] or CUDA [30] (which is specific to NVIDIA GPUs) are still relatively common in the HPC community.

3.2.3. Polly

While tools like OpenMP and OpenACC have the power to significantly reduce the work required for parallelisation, a downside of them is that they still require manual intervention at the source code level. While a loss of readability to the extent of a parallelisation approach which purely uses POSIX threads (see Listing A.2) is prevented, this still can be a lot of work, especially if one desires to optimise their code for more than one platform.

The tool Polly which was written in 2012 by Grosser et al. [14] is able to analyse source code by representing memory-accesses and loop bounds as polyhedra. This representation makes it possible to apply geometric optimisation algorithms like the simplex algorithm [41]. Apart from improving data-locality, Polly also aims to automatically identify and parallelise relevant loops using OpenMP. Polly has been integrated into the LLVM framework (see Section 3.5.3) and can be invoked from its front-ends like clang [42].

3.2.4. Molly

While Polly can in some cases be used to automatically parallelise source code on machines with shared memory, this is not possible on cluster computers with distributed memory, i.e. ensembles of many individual computers that are interconnected with each other. The de-facto standard of enabling programs to run on these machines is MPI (Message Passing Interface) [43], a specification which defines various routines for communication between processes which may run on different physical machines. There are several implementations of the MPI standard like OpenMPI [44] (not to be confused with OpenMP) and MVAPICH [45]. Since programmers have to consider things like explicit communication, memory distribution, and several instances of the same program in different states, it has traditionally been a lot harder to write software for distributed clusters than for single machines [46]. In an effort to combine the message passing paradigm with automatic parallelisation, Kruse created the tool Molly [15], which is based on MPI and Polly. At the time of writing, this is done by specifying via compiler pragmas how the memory layout of individual processes should be, e.g. by specifying a transformation rule of an array's indices to split up a large array between several processes. Molly is then able to generate the necessary communication operations. Furthermore, loop bounds are analysed so that the individual processes each only run a smaller part of the original loop. In benchmarks run by Kruse in 2016 [47], Molly achieved about 4.5 % of the performance of a manual MPI implementation, but the latter also had a significantly larger code size with 320 times as much lines of code.

3.3. Code Patterns

In theoretical computer science, the idea that there are algorithms (or classes thereof) that can be used as building parts for a variety of different problem solutions has been present for multiple centuries. An illustrative example for this is the depth-first search which was probably conceptualised first by Trémaux in 1876 [48] and which has since then been applied in numerous situations like finding the connected components or a topological sorting of a graph [49]. In pseudo-code or other abstract representations of algorithms, this led to the concept of sub-algorithms resembling functions which can be called and then used as placeholders of their results.

Regardless of this, the standard way of programming microprocessors in performance-critical environments in the 1950s was still through assembly language, i.e. a textual representation of machine instructions, because the few compilers for higher-level programming languages that were available at that time could usually not compete against hand-coded assembly performance-wise [50]. However, common assembly languages have only rudimentary support for sub-programs through jump statements and do not feature functions in a mathematical sense, i.e. as mappings $f : X \to Y$. Instead, one has to work around this issue by manually copying input parameters to predefined registers, jumping to a certain address, and storing the result in an adequate way.

In 1954, the first draft specification of *The IBM Mathematical Formula Translating System* was published by Backus et al. respectively IBM. After the Fortran manual and compiler were published in the following years, it was shown that the Fortran compiler, being the first optimising compiler, could outperform manually written assembly code by factors up to 20 which helped its industry adoption [51].

The rise in popularity of high-level languages, especially Fortran, lead to the development of various libraries, e.g. the *Scientific Subroutine Package* (SSP) by IBM [52]. This includes the popular BLAS library (see Section 4.3.3) in 1979 [53]. The outsourcing of functionality into third party libraries helped programmers to concentrate on their problems on a more abstract level and makes it possible that performance-critical subroutines have to be optimised only once by the library creator.

In 2004, Colella listed several important aspects in scientific software [8]. In terms of memory, data locality has special importance. If a data element has been recently used, one speaks of *temporal locality* if the same element will likely be used again soon, and of *spatial locality*, if elements neighbouring in memory will likely be used soon. An example for temporal locality is a loop that uses a data element without changing it. An example for spatial locality is a loop in which a mathematical operation is applied to all elements of an array. Locality has large consequences for optimal cache strategies. While it is often possible to optimise for temporality local elements by using paging strategies like *Least Recently Used* (LRU) or more sophisticated ones, where they might be kept in the cache, spatially local elements require prediction of memory accesses for an optimal strategy which is be attempted through read-ahead in modern CPUs and operating systems [54]. Colella lists seven motifs of scientific computing, i.e. reoccurring patterns that can be routinely found in scientific software. These patterns differ in terms of their computation and data access behaviour. There are *structured* and *unstructured grids*, *dense* and *sparse algebra*, *Fast Fourier Transform*, *particle behaviour*, and *Monte Carlo methods*.

An example for structured grid algorithms is the *stencil pattern* in which a spatially local computation is made on a higher-dimensional array, e.g. a matrix. To illustrate this, Listing 3.1 shows an abridged pseudo-code of the main-loop of the Gauß-Seidel method [55]. As one can see therein, for each (i, j) inside the loop, the value that is written to the matrix element $M[i, j]$ is only dependent on the values of its four neighbours inside the matrix. In a typical implementation, $M[i, j]$ and $M[i, j + 1]$ will always be adjacent in memory.

```
for $i = 1$ to $N$
    for $j = 1$ to $N$
        $star = -M[i, j-1] - M[i-1, j] - M[i+1, j] - M[i, j+1]$
                $+ 4 \cdot M[i, j]$
        $residuum = \textsc{GetResiduum}(i, j, star)$
        $M[i, j] = M[i, j] + residuum$
```

Listing 3.1: Abridged pseudo-code of the Gauß-Seidel method.

In 2006, Asanović et al. listed several classes of programs which are characteristic for their computation behaviour and data movement based on the patterns listed by Colella [7]. Except for *spectral methods* which include FFT, and *n-body methods* which include particle methods, the nomenclature is identical. Asanović et al. call these classes the seven dwarfs and also list an additional six dwarfs: *combinatoric logic*, *graph traversal*, *dynamic programming*, *backtracking / branch and bound*, *construct graphical models*, and *finite state machines*.

In 2019, Barchi et al. showed that classification via machine learning can be used to automatically decide an adequate processing unit for a piece of source code. They used a dataset containing OpenCL kernels[4] which were compiled to LLVM IR (see Section 3.5.3) and benchmarked in terms of their execution time on CPUs and GPUs. This data was used to train a neural network which was then able to predict a suitable compilation unit for an unseen compiled kernel [6].

In Section 4.3.1, a prototype application is described which generalises on the stencil code implementation shown in Listing 3.1. This application is used to train machine learning methods for recognition of numerical methods. In Sections 4.3.2 to 4.3.4, machine learning methods are then used in benchmarks using other datasets. In Section 5.2.2, we describe how this could be used in a similar context as the work from Barchi et al.

3.4. Languages in HPC

According to the PopularitY of Programming Language Index (PYPL), the six most popular languages are Python, Java, Javascript, C#, C, and C++ [56]. However, if we restrict this to only the HPC community, the answer is slightly different:

In a systematic mapping study, Amaral et al. showed in 2019, that the most used general-purpose programming languages in HPC were C++, C, Python, Java, and Fortran [57] (listed in the order given therein). These languages can be split into the following categories: C, C++, and Fortran are usually *compiled* languages (although C++ can also be used in a read-eval-print loop fashion through the Cling interpreter [58]), Java is typically *just-in-time-compiled* (JIT) and running inside the Java Virtual Machine (although recent versions of the Android operating system may use Ahead-of-Time compilation or caching of compiled code [59, 60]), and Python is typically an *interpreted* language (although the tools PyPy [61] and Numba [62] also bring JIT to Python). Also taking into account the number of programs written in each language, it is likely that most HPC applications are currently written in compiled languages.

---

[4] In this context, the word “kernel” denotes a small piece of OpenCL source code, e.g. for matrix multiplication, which can be executed on a computational unit, e.g. a GPU, usually by calling it from inside a program written in a general-purpose programming language like C.

The relatively new language Rust that was created in 2010 [63] with the goal of performance, reliability and productivity [64] and had its first stable release in 2015 has at the time of writing not yet widely been accepted by the HPC community according to the above statistics. However, it has been shown by Sudwoj in 2020 that “Rust is a viable language choice for HPC applications, both in terms of its features, which match or surpass those of Fortran and C++, and in terms of its performance” [65]. Therefore, Rust could rise in popularity in the coming years which would further strengthen the influence of compiled languages in HPC.

A problem with many existing static code analysis tools (e.g. Barchi et al., 2019 [6]; Droste, Kuhn, and Ludwig, 2015 [66]) is that they are limited to a specific language. The methods presented in this thesis are independent of the programming language.

In Section 3.5, there will be an overview of the compilation process, i.e. the steps and tools required to translate a piece of source code to an executable file.

3.5. Code Generation

3.5.1. Code Generation Tools

Figure 3.1 shows a typical pipeline to create executable binary code from source code (e.g. C, C++, or Fortran). The description of the tools below can be found at Levine, 1999 [67]. Note that the whole pipeline may colloquially be called a *compiler* since it can usually be induced all at once via a compiler front-end like gcc or clang, but this is technically not correct. A more precise definition of a compiler is given below.

Here, the term *source code* denotes a high-level programming language that is meant to be as comprehensible as possible for humans, containing control-structures and sub-programs, while still being unambiguously interpretable by a computer. Before it can be compiled, the source code has to be run through a *preprocessor* which eliminates preprocessor directives, e.g. `#include` directives in the C programming language that will be replaced verbatim with the file that is included. A *compiler* transforms this source to *assembly code* which is low-level symbolic machine code that contains machine instructions operating on registers, and performs optimisations during the process. Compiler pragmas, like for OpenMP (see Section 3.2.1), also get processed in this step. An overview of compiler architecture can be found in Section 3.5.2. The assembly code is then transformed to its binary representation, the *object code*, by an *assembler*, which also replaces symbolic by real addresses whenever this is possible. Finally, as a software project might consist of multiple compilation units, which will be individually transformed to object code files, the *linker* will merge these into a single executable file and further resolve remaining symbolic addresses.

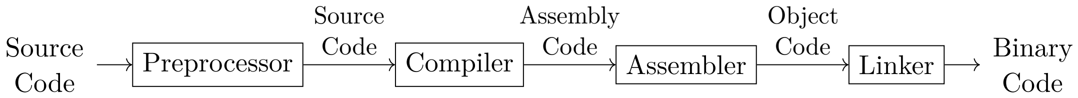


Figure 3.1.: Code Generation Pipeline [67].

On a typical HPC system using a Linux-based operating system, object code files, executable files, and shared libraries are using the Executable and Linking Format (ELF) which standardises the structure of these files through headers, sections and segments. An overview of the basic structure of an ELF file is given in Figure 3.2. Of particular interest for us is the `.text` section which contains the actual machine instructions in binary form. See Section 4.1 for classifiers which work on the `.text` section of object code files.

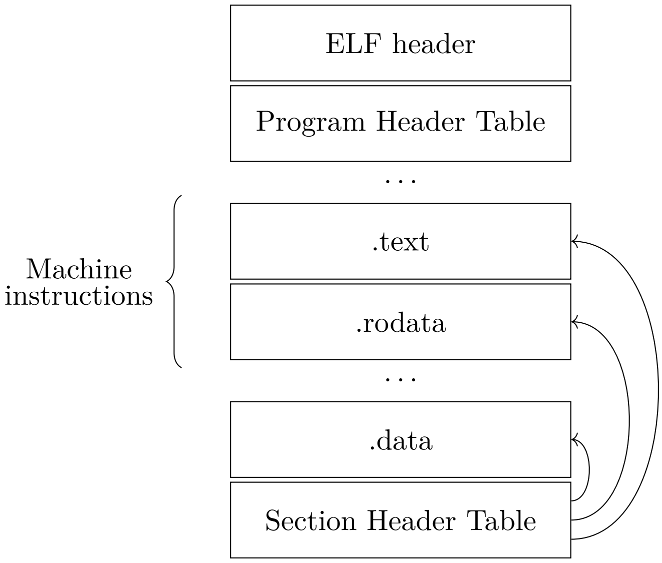


Figure 3.2.: Basic structure of an ELF file (based on Kyi, Koide, and Sakurai, 2019 [68]).

3.5.2. Compiler Internals

The compiler can be considered the heart of the code generation pipeline and consists of multiple phases in itself which is illustrated in Figure 3.3. The individual components can often be split into a front-end, an optimisation step, and a back-end, although the optimisation is sometimes also considered part of the back-end. As an example, an overview of the architecture of the LLVM compiler framework is given in Section 3.5.3. A closer look into the components can be found in Appel and Ginsburg, 1997 [69], and Muchnick, 1998 [70].

The first component of the front-end is the *lexical analyser* which is often also called lexer, tokeniser or scanner, which decomposes the given source code to a sequence of tokens, often through regular expressions. This token sequence is then transformed into an *abstract syntax tree* (AST) by the *parser* which performs syntactical analysis by deciding if the token sequence can be inferred from the grammar of the given programming language. As the name implies, the AST is a tree-representation of the source code's syntactical information, where e.g. a binary operator, i.e. an operator with two operands, is represented by a node with two children. Colloquially, the lexer is often included implicitly when speaking of the parser. The AST is then usually transformed to an *intermediate representation* (IR) by an *IR Generator* which performs the semantic analysis meaning that the type, name, and lifetime of symbols are checked, and that a symbol table is created which stores this information.

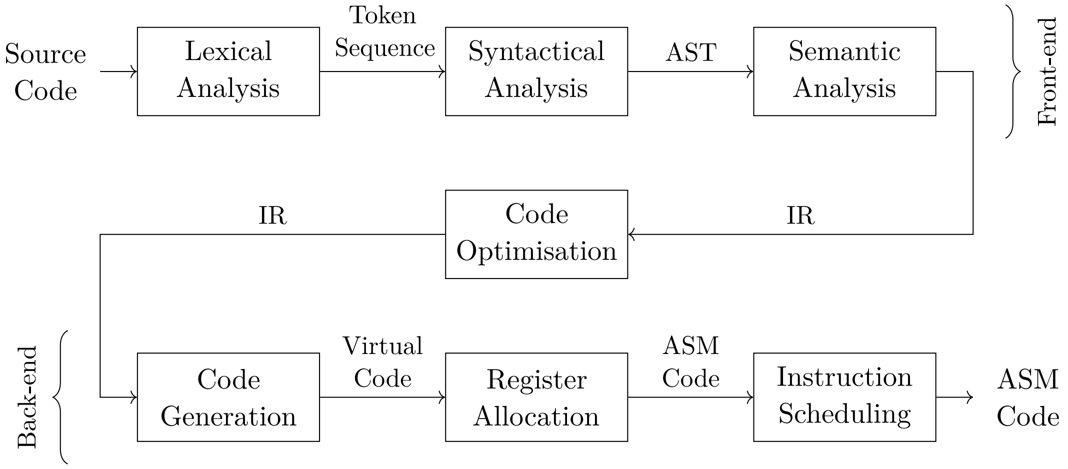


Figure 3.3.: Compiler phases (based on Appel and Ginsburg, 1997 [69], and Muchnick, 1998 [70]).

There are many different examples for IRs and compilers can even feature multiple IRs with a descending abstraction level where different optimisations may be applied on different levels. While high-level IRs are very close to source code, low-level IRs much more resemble assembly code. Inside the LLVM compiler framework (see Section 3.5.3), a single IR called LLVM IR or LLIR is used that acts as input and output of all compiler optimisations.

Inside the back-end, the first step can be *code generation* which can also be called instruction selection. This can often be seen as a translation step from the given IR to virtual code, a representation that resembles assembly code in terms of syntax, but assumes an infinite number of registers in the CPU. This changes after the register allocation, where the code gets limited to the actual physical number of registers, introducing store and load directives whenever necessary. Finally, this assembly code gets rescheduled with the constraint that the overall behaviour must not change to reduce register copy instructions and exploit instruction-level parallelism of modern CPUs.

3.5.3. LLVM

The LLVM[5] project is a compiler framework that can be used to build compilers for many programming languages and architectures due to its modular design.

There is a large number of back-ends for LLVM, including x86, ARM and PowerPC. LLVM compilers are usually cross-compilers by default, meaning that there is no necessity to install special cross compiling binaries if one wanted to compile for a different architecture, e.g. for ARM while working on an x86 machine.

There are also many pre-existing front-ends for LLVM, the most common one being the C/C++ front-end clang.

For Fortran, there are and have been multiple front-ends with the name flang. Historically, flang was the name of an open-sourced version of a Fortran compiler written by PGI and NVIDIA which was marketed as `pgfortran` or `nvfortran`. This compiler fully supports Fortran 2003 and partly Fortran 2008. In 2018, a new project called f18 was started to build a modern compiler that supports Fortran 2018. Since 2020, f18 has been integrated into the LLVM project and renamed to

---

[5] The name LLVM was originally an acronym of *Low-Level Virtual Machine*, but this meaning was removed to avoid confusion, since LLVM is in fact not a virtual machine. The name LLVM is not an acronym any more.

flang[6]. To avoid confusion, the older flang is now more commonly referred to as Classic Flang and is further maintained, since multiple projects depend on it [71, 72].

LLVM's flexibility is possible since every front-end translates the given source code to a low-level representation which is referred to as LLVM IR or LLIR. This intermediate representation is similar to assembly code and can be found in textual or binary form where it is called LLVM Bytecode. All optimisations are performed in a centralised manner by the LLVM Optimiser on this IR. Afterwards, the transformed LLIR is given to an architecture-specific back-end which converts it to an executable. A graphical representation of LLVM's full translation process is given in Figure 3.4.

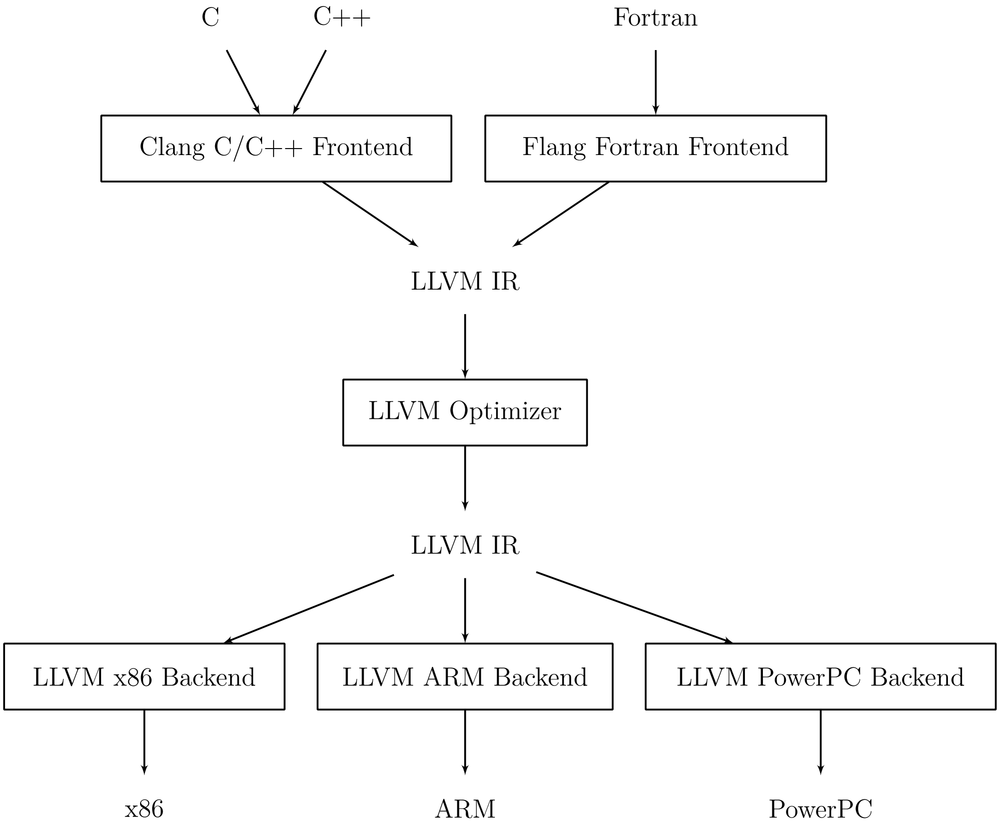


Figure 3.4.: Modular structure of the LLVM compiler (based on Kolek et al., 2013 [73]).

---

[6] It is also worth mentioning that, at the time of writing, the new flang (previously f18) is not available for many Linux distributions including Debian, FreeBSD, Ubuntu, Arch Linux, CentOS, and openSUSE, or via Spack.

3.6. Software Reverse Engineering

In computer science, reverse engineering means the process of reconstructing a higher-level representation of software from a low-level representation like machine instructions, assembly code or bytecode [74]. In the context of compiled languages which are prevalent in HPC (see Section 3.4), popular kinds of tools to achieve this are *decompilers* and *disassemblers*.

A disassembler is a tool which translates binary machine instructions to human-readable assembly and which can therefore be considered the inverse operation to that of the assembler. This process is straightforward and it is easily possible to reconstruct the original assembler input with the possible exception of meta-data and label names. The tool `objdump` [75] which can be used to extract various information about object code files can also be used to disassemble them and identify the start and end addresses of individual functions should they not have been inlined.

A decompiler is an application that generates code in a high-level language from assembly code or machine code, thus they often require a disassembler as a first step. Depending on whether one considers the compiler to be only a small part inside a compiler toolchain, or one calls the entire toolchain compiler, the decompiler can therefore be understood as the inverse of the compiler. Its goal is usually to reconstruct the original source code as faithfully as possible. However, contrary to disassemblers, this is usually not possible easily.

To illustrate this problem, Listing 3.2 shows a small C program which performs the assembly operation `nop` ten times in a `for` loop and returns `EXIT_SUCCESS` which is defined to be zero. `nop` stands for “no operation” and means that a processor core pauses for one cycle. Listing 3.3 shows the abridged x86 assembly output of `gcc` when it is invoked with `-O1` (some optimisations). All meta-data and some labels have been removed. The code reads as follows: Line 1 defines a label for the main method which is not used here. First, the literal value of 10 is written to the register `eax`. A label with the name `.L2` is defined. Then, `nop` is performed. Afterwards, the value of 1 is subtracted from `eax` and written back. If the result is equal to zero, the so-called zero-flag (`ZF`) of the processor is set. `jne` stands for *jump, if not equal* and is a synonym for `jnz` (*jump, if not zero*). It checks if `ZF` is set, and if this is the case, jumps to a specified label, in this case `.L2`. This can be understood as returning to the head of the loop in the C representation. Finally, `eax` is set to zero via `movl` and the value of `eax` is returned. In Listing 3.4, the assembly output of `gcc` using `-O3` (all optimisations) can be found. Here, loop-unrolling has been done. To increase execution speed by preventing jumps and arithmetic operations, `nop` is simply executed ten times[7]. Finally, `eax` is set to zero by computing the `xor` value of itself with itself, which is always zero, and `eax` is returned as in the other listing.

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    for (int i = 0; i < 10; i++) {
        asm volatile("nop");
    }
    return EXIT_SUCCESS;
}
```

Listing 3.2: An example C program which exectutes `nop` in a loop.

```
main:
    movl    $10, %eax
.L2:
    nop
    subl    $1, %eax
    jne     .L2
    movl    $0, %eax
    ret
```

Listing 3.3: Abridged output of the compiler on Listing 3.2 using `-O1`.

```
main:
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    xorl    %eax, %eax
    ret
```

Listing 3.4: Abridged output of the compiler on Listing 3.2 using `-O3`.

---

[7] Normally, a compiler would aim to eliminate all `nop`s from source code, but this is prevented by marking the inline assembly operation as `volatile` in the C source code.

As one can see from these listings, it is very hard for a human to quickly reconstruct the original C sample from each of the assembly listings, since higher-level constructs like loops are entirely absent from assembly languages and have to be translated to jump statements. There are also many details of the implementation that get lost during the compilation process. First, lexical information like identifier names get lost which usually help humans to better understand a source code sample if the programmer gave functions and variables sensible names. Especially preprocessor macros can never be reconstructed, because all of their information gets lost before the compiler is used. Second, the compilation process might introduce changes in behaviour to a program if these changes do not result in a change of net effect, i.e. if they are not visible from outside the program's memory. For example, in Listing 3.3, the loop counter gets decremented, while in Listing 3.2, the counter is incremented, and in Listing 3.4, the counter is absent.

In this thesis, we used RetDec, a retargetable decompiler based on LLVM [22, 76]. In this context, *retargetable* means that RetDec is able to work with compiled files for multiple architectures including x86, ARM, MIPS, and PowerPC as an input. It is possible to reconstruct C source code from complete object code or executable files using RetDec, but one can also choose to specify certain address ranges inside the binary file that shall be used for decompilation.

RetDec performs the decompilation process in multiple phases which are denoted "Preprocessing", "Core", and "Back-End". In the preprocessing phase, it collects various meta-data about the input file [77], e.g. ELF headers. In the "Core" phase, the binary input file is translated to LLVM IR by first using a disassembler to generate assembly code for the specific target architecture and then generating a series of instructions in LLVM IR that is equivalent to the assembly instructions [78]. This can be understood as the inverse operation to the various target-specific back-ends of LLVM (see Figure 3.4) and has the advantage that the result is architecture-independent. RetDec performs some optimisations on the LLVM IR code which are executed as LLVM passes similarly to how optimisations work inside an LLVM compiler. Examples for these passes are system call detection, stack reconstruction, and C++ class hierarchy reconstruction. In the back-end, RetDec translates the LLVM IR to C by starting with a very direct translation which closely resembles assembly and then applying multiple optimisations on it. RetDec features 29 different back-end optimisations like dead code elimination, conversion of `while(true)` loops to conditional loops by extracting `break` conditions or elimination of unused identifiers. Using this bottom-up approach, RetDec is able to generate readable code that closely resembles the original C source code in many cases, even going so far that calls to functions in C standard library headers are identified and reconstructed. Examples for this are listed in Section 4.2.

RetDec's back-end optimisations can be individually and fully disabled, but this is not the case for the LLVM passes performed in the core. In Section 4.2, we describe a source code classifier which uses RetDec alongside the ASTNN (see Section 3.11) which partly suffers from this since some optimisations were unresponsive on some samples and could not be disabled in all cases.

3.7. Language-independent classifiers

If presented with the task of building a language-independent code classifier, one could come up with multiple ideas to use the various levels of program representations that were described in Section 3.5. A common property of all approaches listed below is that they can (or even have to) be used after the compiler optimisations have already been executed. On one hand, this can be an advantage, because optimisations like dead-code-elimination would probably remove noise that could degrade the performance. On the other hand, optimisations can also appear obfuscating which could have an disadvantageous effect.

First, one could use one of the intermediate representations that are used internally by the compiler as described in Section 3.5.2. If one uses an LLVM-based compiler like `clang` or `flang`, this could be the LLVM IR which has a large set of tools built around it. This would come with the disadvantage that users of the classifier were tied to a specific compiler and, if they were not using it in the first place, had to possibly change their building process. While `clang` is very widespread in the scientific community, `flang` is often not a good choice for Fortran programmers (see Footnote 6). Alternatively, one could use the assembly output of the compiler for classification. However, compared to the original source code, both assembly and intermediate representations tend to be significantly longer. This increase in verbosity would lead in a decrease of information density and could lead to a lack of context-awareness of the classifier as has been described by Zhang et al. for similar methods [20]. In this thesis, this approach is, therefore, not considered further.

Next, one could use the ELF binary data that is emitted from either the assembler (object code) or the linker (usually executable code). This can be done with many different general-purpose classifiers on a byte-code level which is further described in Section 4.1. After linking, all individual object code files and statically linked libraries get merged, and link-time-optimisation is possibly performed. While this could be desirable, operating on the object code level also helps to split the data into smaller, more manageable chunks. We also noticed that simply linking the `libm.a` against a program, which is done via `-lm` in most C compilers, significantly increases the final executable's file size, introducing noise to the classification. Therefore, we only used object code in our experiments.

Finally, a more complicated approach is using a decompiler to reconstruct high-level source code from binary files. Most of the disadvantages of working with binary files described above apply here as well, but it is possible that some of them can be compensated by the more expressive nature of the reconstructed source code. It is demonstrated in Section 4.2 that a decompiler like RetDec (see Section 3.6) is often able to reconstruct source code such that the end result is very close to the original source code. Since C source code is a lot less verbose than intermediate representations or assembly, there can be much more information picked up by a single statement.

Based on the choice of program representation, different types of classifiers may be suitable. In this thesis, only machine learning classifiers were used. An overview over the methods that can be applied with the representations described above is given in Sections 3.8 to 3.11.

3.8. Machine Learning Classification

There have been numerous different approaches for static code analysis that involve machine learning which contains multiple sub-fields: In *supervised learning*, one tries to predict a mapping $f : X \to Y$, where X is a set of *samples*, and Y is a set of *labels*. This contains *regression*, where Y is continuous, and *classification*, where Y is discrete. In the latter, the elements of Y are also called classes. Besides other fields, there is also *unsupervised learning*, where patterns are attempted to be found inside datasets that are unlabelled. In this thesis, we only make use of classification, and therefore use the terms *classifier* and *estimator* interchangeably. In practice, one trains a classifier upon a *training set*, a set of samples where this mapping is known, and later predicts the labels for unknown data, e.g. from a *test set* [79]. A common problem of machine learning methods is *overfitting*, i.e. the phenomenon that a classifier achieves good results on the training set, but does not generalise on unseen data very well. For different classifiers, different strategies to prevent overfitting may be attempted [80, 81].

The Python library `scikit-learn` [82] (often abbreviated with `sklearn`) offers a multitude of machine learning methods, e.g. general-purpose classifiers that can be used as drop-in replacements for each other because of their consistent API. A graphical overview of some of `sklearn`'s classifiers can be found in Figure C.1. A short overview over different groups of classifiers that are contained inside `sklearn` and that we used for this thesis is given in Sections 3.8.1 to 3.8.6.

A disadvantage of `sklearn` is that it does not support GPUs, although the project H2O4GPU aims to replicate `sklearn`'s interface and run the models on the GPU [83]. There are many other machine learning libraries for Python, like PyTorch [84], Keras [85], and TensorFlow [86] which are specialised on the creation of neural networks and also offer GPU support via CUDA.

In the context of static code analysis, one can distinguish between methods that represent code as tokens [17, 18], and methods that make use of its abstract syntax tree [19, 20, 21]. Token-based methods may either train directly on the byte-representation of a program or on vector-representations of the tokenised output of the lexer (see Section 3.5.2), e.g. by using Word2Vec (see Section 3.9). In principle, these tasks can be performed by every classifier that supports arbitrary input data. On the other hand, AST-based methods almost exclusively make use of neural networks (see Section 3.8.6).

3.8.1. Nearest Neighbour Classifiers

Nearest neighbour methods can be considered as the most simple machine learning techniques. Their estimations work by performing a majority vote of multiple samples taken into account which get selected based upon the shortest distance from a sample using an arbitrary distance function, i.e. the Euclidean distance. This selection can be made based upon a predefined number k, which is then called *k Nearest Neighbours Classification*, or all samples inside a specified radius r, which is called *Radius Neighbours Classification* [87]. It should be noted that the latter might perform better in environments where the data is not uniformly sampled, but will fail if the minimum distance of all neighbours of a sample is larger than r which is why this method was not used in this thesis. However, the optimal choice for k is also not trivial: A larger k will result in more samples being taken into account which can suppress noise, but can also make it harder to clearly distinguish the individual classes. Since nearest neighbour methods only store the training data and do not construct a model for a function that is attempted to be learnt, they are called non-generalising methods. However, especially the k nearest neighbour classifier is still widely used, since generalisation is not always necessary for acceptable results, and since they showed good results on different problems, e.g. the classification of hand-written digits [88].

3.8.2. Decision Tree Classifiers

Decision Tree Classifiers [89, 90] make predictions based on multiple boolean rules (decisions) that build up a tree. For a sample in a high-dimensional vector space $x = (x_1, \dots, x_n) \in \mathbb{R}^n$, all these decisions have the shape

$$x_i > z \quad \text{or} \quad x_i < z,$$

where $z \in \mathbb{R}$ is an arbitrary threshold for each decision.

This tree can be built up iteratively by starting with the trivial tree that contains only one node and repeatedly selecting a feature where the dataset can be split well into two classes. There are multiple ways to find this feature, e.g. methods that try to reduce the overall variance. The deeper the tree grows, the more fine-grained the decisions become which can lead to an increase in accuracy, but also leads to an increase in memory consumption and prediction time cost as the time to traverse the tree grows logarithmically with the depth. An advantage of decision trees is that the tree can be examined and the decisions that lead to a specific classification can therefore be reviewed, but they are not performing well in situations where data is badly describable by if-then-else rules [90]. In such cases, ensemble methods like Random Forest (see Section 3.8.5) may be better.

3.8.3. Naïve Bayes Classifiers

Naïve Bayes (NB) methods are estimators that are based upon the assumption that all samples from the training set are pairwise conditionally independent, i.e. that for a class variable (label) y, a feature vector $(x_1, \dots, x_m)$, and their corresponding probability distributions

$$P(x_i | y, x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n) = P(x_i | y)$$

holds, so that Bayes' theorem [91]

$$P(y | x_1, \dots, x_n) = \frac{P(y) P(x_1, \dots, x_n | y)}{P(x_1, \dots, x_n)}$$

simplifies to a product of probabilities:

$$P(y|x_1,\dots,x_n) = \frac{P(y)\prod_{i=1}^{n} P(x_i|y)}{P(x_1,\dots,x_n)}.$$

They are called naïve, because this very strong assumption does not hold in general [92]. However, Naïve Bayes Methods show good results in tasks like spam classification nonetheless [93]. Different classifiers make different assumptions about the underlying probability distributions, e.g. Gaussian NB classifiers assume that the probabilities are given by Gaussian probability distributions:

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i-\mu_y)^2}{2\sigma_y^2}\right).$$

3.8.4. Support Vector Machines

Support Vector Machines (SVM) or *Support Vector Classifiers* (SVC) [94] are estimators which divide a set of objects into two classes by a separating hyperplane so that the empty area around the class boundary is maximised, and they are therefore so-called large-margin classifiers. The separating hyperplane can be parametrised by

$$\langle w, x\rangle + b = 0,$$

where $w, x \in \mathbb{R}^n, b \in \mathbb{R}$. This wide, empty border should later ensure that objects which do not correspond exactly to the training objects are also classified as reliably as possible. Given a set of linearly separable data points that can be viewed in a vector space, it is not necessary to consider all training vectors, because vectors that lie further away from the hyperplane do not influence its position and orientation, since they are hidden behind other vectors. The vectors that are closest to the hyperplane are the only ones that are needed for the mathematical description. These are called the *support vectors*. For the case of data that is not linearly separable, SVMs may make use of the so-called *kernel trick*: Instead of explicitly transforming the data to a vector space with a higher dimension in which the data may be actually linearly separable[8], which would be very computationally expensive, one can instead use a kernel function in place of the inner product to construct the hyperplane. Kernel functions are functions $k : X \times X \to \mathbb{R}$ with a

---

[8] It is always guaranteed that such a vector space exists for every set of training data if the number of dimensions is not bounded.

vector space X, if a vector space with an inner product $(F, \langle\cdot,\cdot\rangle)$ and a function $\varphi : X \to F$ exist so that $k(x, x') = \langle\varphi(x), \varphi(x')\rangle$ for all $x, x' \in X$. Examples for kernel functions are linear kernels

$$k(x, x') = \langle x, x'\rangle,$$

polynomial kernels

$$k(x, x') = \langle x, x'\rangle^n, \text{ where } n \in \mathbb{R}^+,$$

and radial basis functions (RBF)

$$k(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right), \text{ where } \sigma \in \mathbb{R}^+.$$

To perform classification on more than two classes, one can combine the estimations of multiple SVMs using a majority vote [95, 79]. Inside `sklearn`, there are multiple ways to use SVMs: Besides the `svm` module which contains `SVC` and `NuSVC` which are very similar and both support various kernels, the default being RBF, and `LinearSVC` which uses a linear kernel, there is also the `SGDClassifier` from the `linear_models` module which uses an SVM as the default.

3.8.5. Ensemble Methods

Ensemble methods [96] combine the estimations of several individual estimators to achieve better results than with a single estimator. In general, one can distinguish between *averaging methods*, which take the average of several classifiers to decrease the variance of the estimations and therefore the overall robustness, and *boosting methods*, which aim to build up a strong classifier from several weak classifiers which only have to perform better than guessing. A popular averaging method is the *Random Forest* which combines the results of several smaller decision trees (see Section 3.8.2) that have been constructed from a random subset of the training data. On the other hand, examples for boosting methods are *AdaBoost* which performs a weighted majority vote of its classifiers, and *Gradient Tree Boosting* which, like a random forest, combines multiple trees, but does not simply average their results and instead combines them similarly to AdaBoost.

3.8.6. Neural Networks

Artificial neural networks[9] (NNs) [97, 98] are machine learning methods that are inspired by the brains of animals, and, as the name suggests, built up by artificial neurons that are connected to each other. These neurons are also called *perceptrons* [99, 100] and each compute an inner product between a weight vector w and an input vector x

$$\hat{y} = \sum_{i=1}^{n} w_i x_i,$$

where $\hat{y}$ is the prediction. A single perceptron is already able to predict a linear function $f : \mathbb{R}^n \to \mathbb{R}$ if the weights are set accordingly. This can be done by updating the weights after the time step t via

$$w_i^{t+1} := w_i^t + r \cdot \ell(\hat{y}_j, y_j) \cdot x_{j,i},$$

where r is the learning rate, and ℓ is a loss function (e.g. $\ell(x, y) = x - y$). Neurons are usually organised in groups called *layers*, where in simple cases the output of one layer is used as the input for the next layer, with the exception of the first layer, which is the input layer, and the last layer, which is the output layer. NNs are therefore also called *Multilayer Perceptrons* (MLP)[10].

Figure 3.5 shows a hierarchy of different types of neural networks. Aside from the most simple examples which are also called *Feedforward Neural Networks*, there are also *Recurrent Neural Networks* (RNN) [101], where data may pass the network multiple times. Subgroups of RNNs are *Bidirectional RNNs* [102], where the layers of the network are connected in both directions, and *Long Short-Term Memory Networks* (LSTMs) [103], where nodes also contain storage to remember certain information over a longer time. The latter approach tackles a specific problem that RNNs have when used on sequential data like natural language which is called the *Vanishing Gradient Problem* [104]: When neural networks update their weights based on the gradient of a loss function which has to be computed through the chain rule, effectively multiplying a large number of values between 0 and 1, this gradient can become increasingly small up to the point where the neural network is not able to be trained any further. In the context of natural language processing, this can mean that RNNs have difficulties picking up the long-range context of words. LSTMs solve this by explicitly remembering this long-range context in their

---

[9] The term "artificial" is usually omitted in computer science.

[10] In `sklearn`, the `MLPClassifier` uses a neural network.

memory cells. A special kind of LSTM is the *Tree-LSTM* [105], where the nodes arrange themselves in a tree structure based on tree-shaped input data like syntax trees. Tree-LSTMs have mainly been used for natural language recognition [106], but have also been used for static code analysis on ASTs [21].

Like other classifiers, NNs may also suffer under the problem of overfitting. In practice, one often splits up a dataset into training, validation, and testing datasets. While the training set is used to set the weights of the network, the validation set is used to also compute a *validation loss*. To prevent overfitting, one should stop training when the validation loss is at its minimum [81].

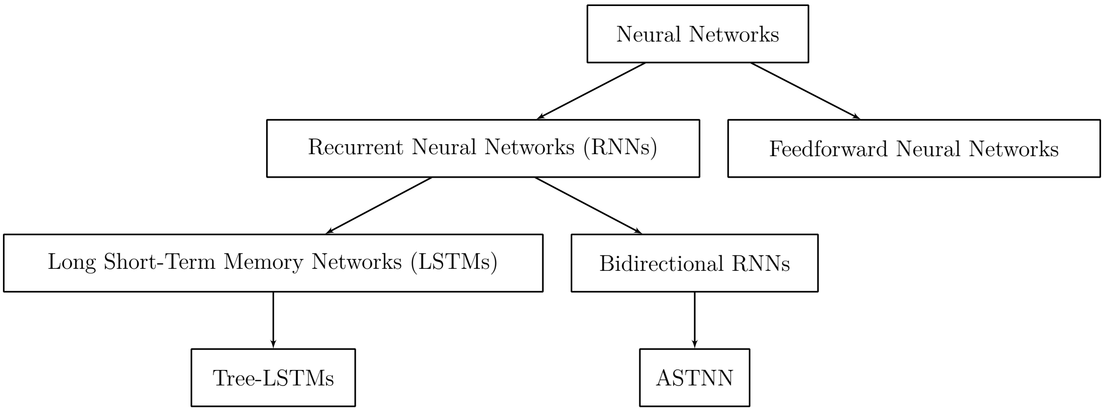

Neural Networks
Recurrent Neural Networks (RNNs)
Feedforward Neural Networks
Long Short-Term Memory Networks (LSTMs)
Bidirectional RNNs
Tree-LSTMs
ASTNN

Figure 3.5.: Hierarchy of different neural network architectures [107, 103][11].

3.9. Word2Vec

Word2vec is a method of representing words inside a natural language text as tuples of numbers using a neural network [108, 109]. This neural network is trained upon large bodies of text with the goal that words that appear in similar contexts will also be similar in terms of the cosine of the angle between them in the resulting vector space (cosine similarity).

---

[11] These arrows do not denote a strict subset relation, but “$a \rightarrow b$” should rather be read as “b originated from a”. Hybrid forms, like a bi-directional LSTM, do exist; and technically, regular LSTMs can be considered a linear specialisation of Tree-LSTMs.

One of the goals of Word2vec is that the vector representations behave in a way so that certain algebraic operations intuitively make sense to a human, e.g.

$$\text{vec}(\text{“Madrid”}) - \text{vec}(\text{“Spain”}) \approx \text{vec}(\text{“Paris”}) - \text{vec}(\text{“France”}),$$

where the function vec denotes the words' vector representation.

Word2vec is frequently used in natural language processing [110], but may also be used for static code analysis using machine learning [111, 112] like the neural network that was used for this thesis (see Section 3.11).

3.10. Online Judge Dataset

The Online Judge (OJ) Dataset was created by Mou et al. in 2015 by downloading source code samples from a pedagogical programming open judge (OJ) system that students could submit their solutions to various programming problems to, so that they can automatically be evaluated by running them. The dataset is labelled with 104 different classes which correspond to the programming problem that is attempted to be solved. For each problem, 500 random samples have been selected. Therefore, there are 52 000 samples in the dataset [19]. Zhang et al. state in their paper that the dataset consists of C programs [20], but this is at least partly untrue, as many snippets (see Listing A.4 for the first sample) are clearly written in C++, but it is not known whether the dataset contains source code snippets which are valid C, but not valid C++[12]. Both Mou et al. and Zhang et al. used the python module `pycparser` [114] to generate the ASTs from the samples which is possible because `pycparser` allows for parsing of many C++ features like `operator<<`. However, it is not possible for many samples from the dataset to be compiled with a C or C++ compiler without changing them. Therefore, it is likely that the samples ran through a preprocessing step to shorten them as much as possible while retaining the ability to parse them.

---

[12] C++ is *not* a strict superset of C [113].

3.11. ASTNN

In 2019, Zhang et al. have proposed the ASTNN, a model for source code classification and clone detection based on abstract syntax trees [20] which outperformed existing methods, achieving an accuracy of 98.2 % on the OJ dataset, while the existing top-performer at this point, the TBCNN from Mou et al., achieved an accuracy of 94 %.

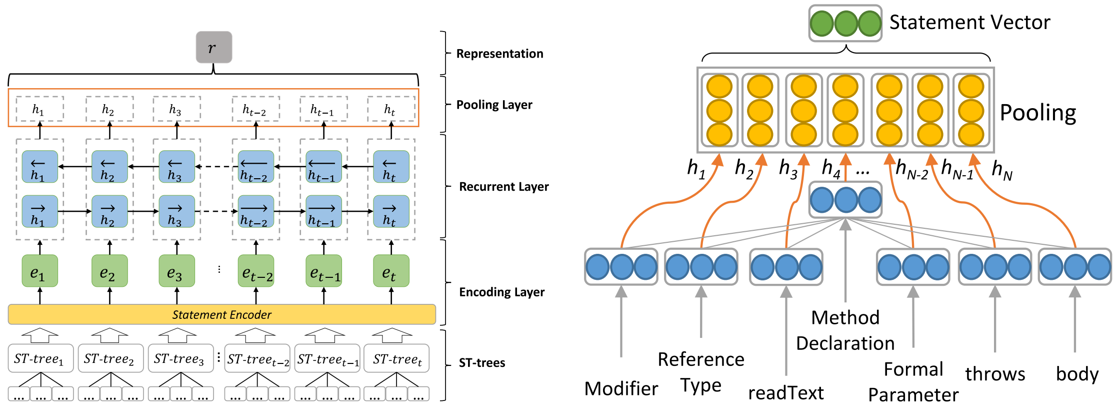
As has been stated in Section 3.8, there are token-based and AST-based machine learning methods for static code analysis. Zhang et al. state that both have significant disadvantages: While token-based methods fail to recognise the greater context of the tokens and will therefore not pick up all of the structural information of source code, a problem with methods that treat ASTs similarly to natural language syntax trees is that the former grow much deeper (they state that node numbers of 7,027 and depths of 76 are common) which can lead to the *Vanishing Gradient Problem* which was described by Hochreiter in 1998 [104]. Although LSTMs [103] were specifically introduced by Hochreiter and Schmidhuber to tackle this problem, even Tree-LSTMs [105] are not guarded against this problem for trees of these depths, since the content of the memory cells gets computed recursively by traversing the tree. This is even worsened by the fact that in many approaches, the ASTs get converted to binary trees which typically increases the depth drastically. Even though this problem could be solved by the usage of control flow or data dependency graphs, Zhang et al. state that they have not followed this approach because this would require the programs to be present in a compiled state and render uncompilable code fragments unusable[13].

Instead, the ASTNN uses an approach called *statement trees* which can be seen as the middle ground between the two aforementioned approaches: Not the whole AST is used for training as is, but it is rather split up into a list of multiple smaller trees that each resemble a single statement, where the root determines the type of the statement. Zhang et al. have also experimented with other granularities like a full AST, individual statement nodes, and the contents of compound statements, i.e. the body between two braces, but these approaches came with the problems described in the paragraph above and have therefore been neglected.

---

[13] Note that this is not really an advantage for this thesis, as we even require compiled programs to be present in the method based on the ASTNN that is described in Section 4.2. Instead, we used the ASTNN for this thesis, because it achieved good results in benchmarks at the time of writing.

The ASTNN has been used by Zhang et al. for source code classification for C/C++ and source code clone detection for C and Java. In this thesis, we will only consider the former task. The overall structure of the ASTNN can be seen in Figure 3.6a. For classification, the implementation is split up into a pre-processing phase (`pipeline.py`) which handles the conversion of a source code sample given as text to a single vector representation (see Figure 3.6b), and a training and testing phase (`train.py`) [115]. The ASTNN is built upon the neural network library PyTorch [84].



(a) The architecture of AST-based Neural Network

(b) The statement encoder, where blue, orange and green circles represent the initial embeddings, hidden states and statement vector, respectively.

Figure 3.6.: Architecture of the ASTNN and the statement encoder therein [20].

First, the source code gets parsed using `pycparser` which emits an AST that is then split up into a list of statement trees. See Figure 4.2a for an AST of a small C program[14]. There are several constructs that acquire special treatment: Method declarations get treated as statements, and apart from that, other statements that can be split into a head and a body, e.g. loops[15], will be treated as two statements. Here, statements do not get nested: Only AST nodes that are *not* statement nodes themselves get included as descendents of a compound statement node. Therefore, the individual statement trees will be disjoint of each other.

---

[14] This AST was dumped from clang and not by `pycparser`, but the general structure will be similar regardless of the parser.

[15] Zhang et al. also list `try` statements here that are technically absent from the C language, but it has already been established that the OJ dataset also contains C++ samples.

In the next step, all nodes of the statement trees are tokenised to string representations depending on the node type, e.g. for the root of the statement trees, this is usually the statement type (e.g. `MethodDeclaration`) converted verbatim to a string, otherwise the values of literals or identifier names may be used which means that the ASTNN can work with both syntactical and lexical information. Afterwards, Word2vec (see Section 3.9) is applied recursively on the tree to retain vector representations $h_1, \ldots, h_N$, $N \in \mathbb{N}$, of the individual nodes, where $h_i = (h_{i1}, \ldots, h_{ik})$, $k \in \mathbb{N}$. Finally, *max-pooling* is used, i.e. a single vector e of length k is computed via

$$e = (\max(h_{i1}), \ldots, \max(h_{ik})).$$

These statement vectors are denoted with $e_1, \ldots, e_t$ in Figure 3.6a. The individual nodes are passed to an RNN composed of multiple layers (depicted as *Recurrent Layer* in Figure 3.6a) which aims to pick up dependencies between the individual statement vectors. The weights of the recurrent layers are then compressed to a single vector representation r of a whole source code fragment via max-pooling. To use this vector for source code classification, a feed-forward network is trained on a one-hot encoding of the individual classes[16]. In a testing environment, this network then outputs the probability for each class that the source code fragment belongs to it. The final prediction can be extracted by selecting the maximum.

By default, the ASTNN will use the GPU, a batch size of 64, and 100 hidden dimensions, and will stop training after 15 epochs. In this configuration, the ASTNN requires a GPU that supports CUDA and 16 GB of RAM, although the ASTNN's configuration can easily be changed. In Table 3.1, an overview over the average time cost per epoch (over 100 epochs) is given for different machine configurations. The GPU was used if applicable. The first benchmark was performed using a desktop PC with consumer hardware, the second benchmark was performed on the host `igor` of the Hamburg Observatory, and the third benchmark was performed on the host `abu1` of the research group scientific computing ("Wissenschaftliches Rechnen", WR) of the Universität Hamburg. Remarkably, the desktop system was about 20 % faster than `igor`, even though the latter has a much more powerful GPU (in terms of

---

[16] One-hot encoding means that for n classes a vector with n entries is used, where only one entry is one and all other entries are zero. The index of the non-zero entry corresponds with the index of the class that is represented. Since models that are trained using one-hot encodings often output values between zero and one for each class, this representation can also be understood as a discrete probability distribution.

raw processing power[17]). It is also worth noting that the benchmark took about 24 to 30 times longer on `abu1`, stretching the average time cost per epoch to over 40 minutes, which is not surprising since we had to train on the CPU there. However, this becomes relevant in Section 4.2 where we are forced to compute on the CPU for memory reasons.

Table 3.1.: Time cost per epoch of the ASTNN on different machine configurations.

CPU	**GPU (VRAM)**	**RAM**	**Time cost**
Intel(R) Core(TM) i7-10700 CPU	NVIDIA GeForce RTX 2060 (6 GB)	32 GB	$(84.0 \pm 5.9)\,\mathrm{s}$
Intel(R) Xeon(R) Silver 4110 CPU	NVIDIA Tesla V100 PCIE (32 GB)	200 GB	$(107.1 \pm 0.8)\,\mathrm{s}$
2 × AMD Opteron(tm) Processor 6344	–	256 GB	$(2561.3 \pm 142.6)\,\mathrm{s}$

---

[17] The NVIDIA Tesla V100 achieved a more than twice as good result as the NVIDIA RTX 2060 in the Geekbench OpenCL benchmark [116, 117, 118].

4 Chapter 4. Methods

In this chapter, we describe and evaluate the source code classifiers that were used in this thesis. In Section 4.1, we describe straight-forward methods that are based on general-purpose classifiers from `sklearn`, and in Section 4.2, we describe a method that is based on the ASTNN and the RetDec decompiler. The methods are evaluated in Section 4.3 through performance benchmarks on multiple different datasets. During the evaluation, multiple problems were found which are discussed in Section 4.4.

4.1. Object Code Classification

We used multiple classifiers from `sklearn` (see Section 3.8) to classify object code files by treating their content as raw byte-data. First, for each object code file, the `.text` section was extracted, because as has been explained in Section 3.5.1, this is the section that contains the actual machine instructions (as opposed to meta-data, variable initialisations, etc., which we consider noise). This is done with the Python package `pyelftools`. Additionally, we can identify the start and end addresses of individual methods inside the object code file using the tool `objdump`. This can be used to classify them individually as in Section 4.3.4, but should be used prudently, because compilers tend to inline functions on higher optimisation levels. Then, we had to ensure that all samples for a benchmark had the same length, because every classifier inside `sklearn` expects this to be the case. For this problem, the following solutions may come into mind: First, one could extend all samples to the largest length, either by padding them with zeros or by repeating the samples in a cyclic manner, or one could crop all samples to the shortest length. For our experiments, we used the first approach, because in this way, no information gets added or lost, although the added zeros might incite some algorithms to partly fit on the (unpadded, i.e. non-zero) length of the samples which is probably not desirable. After the padding, we perform a train-test split with a ratio of 4:1, and use a standard scaler to enforce a mean of 0 and a standard deviation of 1 on the datasets, because this is beneficial for many classifiers [119]. A graphical overview over this process is given in Figure 4.1. The method has been tested on various datasets, and benchmarks thereof can be found in Section 4.3. In the following section, we describe a method based on the ASTNN which outperformed the object code classifiers in all benchmarks.

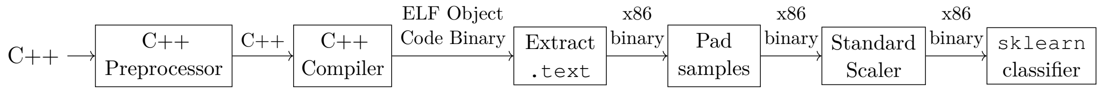
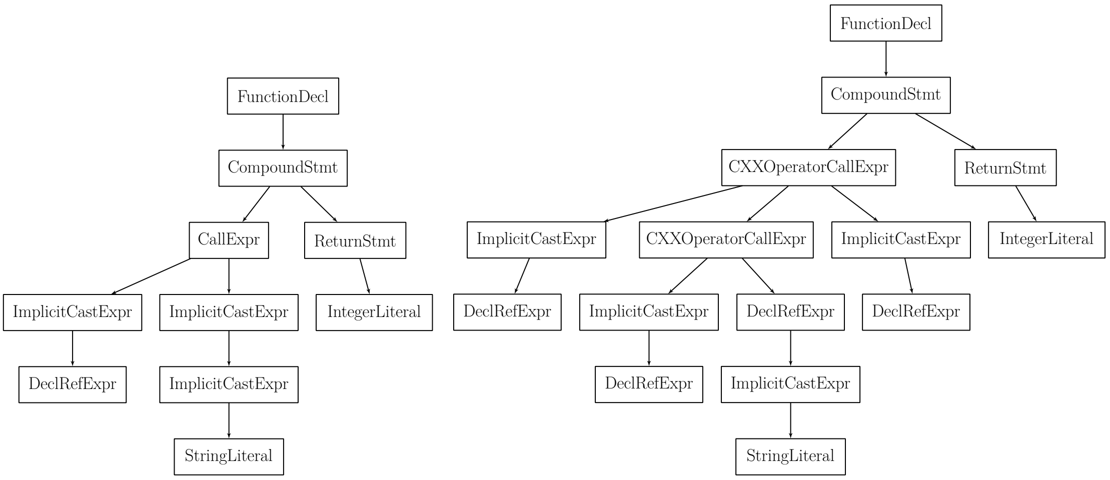


Figure 4.1.: Value flow diagram of the object code classification.

4.2. Classification of Reverse-Engineered Source Code

In Section 3.11, a description of the tool ASTNN was given, which can be used for source code classification. A downside mentioned in this section was that it currently only supports the C programming language for classification tasks. On one hand, the ASTNN's design is extensible and creating support for a different programming language should require only few steps after including a parser for this language. On the other hand, ASTs of source code snippets for the same task in two different languages can be quite different. This is illustrated in Figure 4.2 which shows ASTs for the source code snippets given in Listing A.5 and A.6. Since their effect is identical – both print "Hello World!" to the console – they should be treated as counterparts of each other, but since the printing is done via a function (`printf`) in C and via stream operators (`operator<<`) in C++, it would be very hard to recognise this in the AST. This can also apply to arithmetic code, e.g. the recommended way of the Standard C++ Foundation to implement a subscript operator for a matrix is by overloading the `operator()` [120] which is not possible in C, where one might use multi-dimensional arrays instead. To avoid redundancy in the following, we will call two programs that are semantically identical, i.e. that show the same behaviour, but are syntactically different, *siblings* if they are written in the same programming language, and *cousins* if they are not.



(a) C program (see Listing A.5) (b) C++ program (see Listing A.6)

Figure 4.2.: Abstract syntax trees for the main method of *Hello World* programs.

Our solution to the above problem was keeping the ASTNN mostly unchanged and building upon the basic idea explained in Section 4.1 so that arbitrary object code files can be used as an input for it. We did this by processing them with RetDec, a retargetable decompiler, which is able to create C code from them through Reverse Engineering. In the following, we will denote this as *reconstructed* source code to distinguish it from the *original* source code. Note that, while this may suggest that the languages of these two types are always the same, this is not always the case here.

However, for the case that the languages *are* identical, namely when using C as input, one can see in Listing A.7, where the decompiled *Hello World* program from Listing A.5 is shown, that RetDec can in some cases do a remarkably good job in creating source code that looks very close to the original. This even goes so far that the necessary headers get included so that the `printf` function from the C standard library can be used.

In contrast, the C source code shown in Listing A.8 which is reconstructed from the C++ program shown in Listing A.6 shares very little similarity with it. This is quite expectable since the required language features like `operator<<` are not present in C, so the underlying functions of C++' standard library have to be used instead. In particular, we are actually not interested in a source code reconstruction that is as faithful to the original as possible, but rather in a method where the reconstructions of cousins are as close as possible to each other syntactically. While one might argue that Listing A.8 is a lot harder to read than Listing A.7, they both share the property that the relevant line of the `main` function that prints is a function call, so in terms of their AST, the samples became more similar through the compilation and decompilation process.

Therefore, these steps can be seen as a sort of *source code normalisation* which is also the term we will use from now on. A graphical overview over the full process is given in Figure 4.3, where C++ has been used as an input language. It is trivially transferable to other languages that get compiled to an ELF binary.

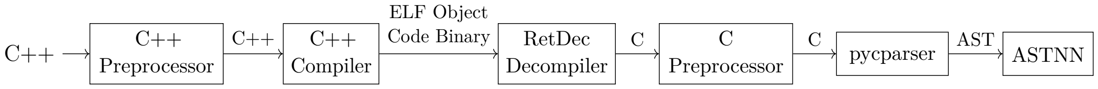


Figure 4.3.: Value flow diagram of the Source Code Normalisation process.

A previously unmentioned aspect of this process is the role of the preprocessor. Since `pycparser`, the python module that is responsible for creating the AST for further usage inside the ASTNN, does not support preprocessor directives, and RetDec will insert `#include` statements to the C standard library in its reconstructed C code, we need to remove them before we can process the source code with it.

In general, it is *not* sufficient to simply delete all lines starting with a hash sign (`#`). This is illustrated in Listing 4.1. Both displayed source code snippets contain valid C programs with a different behaviour although the verbatim content of their `main` methods is identical: On the left, `a` is defined to be `int` so that in the first line of the `main` method, a variable `b` is defined which effectively has the type of `int` pointer. On the right, both `a` and `b` are defined to be the integer literal `1` so that in the first line of the `main` method a multiplication of `a` and `b` is performed which is not used any further. This clarifies that we have to use the actual C preprocessor for the task of resolving preprocessor directives. However, C standard library implementations like the `glibc` contain keywords and qualifiers (e.g. the `__restrict` keyword which can be found at the arguments of `memcpy` inside `string.h`) that `pycparser` also does not support. Therefore, `pycparser`'s source repository contains a so-called “fake libc” which is a collection of headers that have the same names as the standard library headers but that only contain 151 dummy `typedefs` and 207 `#defines` that allow `pycparser` to parse an arbitrary piece of source code without external dependencies [114]. However, we modified the ASTNN slightly to remove all nodes describing `typedefs` from the

AST after the parsing process, since RetDec's output source code never contains them and therefore all of them have to originate from `pycparser`'s fake libc, and all source code samples that include a header from the standard library will contain the same approximately 150 lines at the start. This should improve the neural network's convergence without losing comparability to the original ASTNN on the unmodified dataset.

```c
#include "stdlib.h"

#define a int

int main(void) {
  a * b;
  return EXIT_SUCCESS;
}
```

```c
#include "stdlib.h"

#define a 1
#define b 1

int main(void) {
  a * b;
  return EXIT_SUCCESS;
}
```

Listing 4.1: Illustration that preprocessor directives can drastically change the syntactical interpretation of C statements.

Since a build process for a software usually already exists before analysis is attempted, and the code base of a large HPC application is often split up into modules that are compiled separately, this pipeline can usually be plugged into an existing program. Furthermore, similar to the object code classification described in Section 4.1, we can again utilise `objdump`'s output to identify the start and end addresses of individual methods inside the `.text` section, which can then be passed to RetDec.

4.3. Benchmarks

In this section, multiple benchmarks of the methods described in Sections 4.1 and 4.2 can be found. All timed benchmarks were performed on the machine `abu1`, but some benchmarks took longer than 48 hours (see Table 4.4) and have therefore been run on a different machine. For classifiers from `sklearn`, the arguments `n_jobs=-1` (adapt the number of parallel jobs to the logical number of CPUs) and `random_state=42` (use a fixed seed for the pseudo-random number generator) have been used where applicable, so the results are reproducible across different machines. If necessary, the maximum number of iterations was increased if a classifier terminated with a warning that convergence has not yet been reached.

4.3.1. Prototype Application Dataset

In order to test the approach described in Section 4.1, we created a small application written in C which features ten functions which are listed in Table 4.1 and fulfil different roles: There is memory management (allocation and deallocation of a matrix), matrix initialisation, simulated I/O from the Linux block devices `/dev/urandom` and to `/dev/null`, numeric calculation, and matrix display. Some of these functions are directly inspired by a program called `partdiff` which was originally written by Ludwig and Schmidt [121] and solves partial differential equations like Poisson's equation [122] using the Gauß-Seidel or Jacobi solver [55, 123].

Table 4.1.: Functions defined in and roles of the functions in the prototype application.

Method name	**Functionality**
`allocate`	Memory management
`init_simple`	Matrix preparation
`read_from_dev_urandom`	Input
`preprocess_matrix`	Matrix preparation
`calc_simple`	Numeric calculation
`calc_mirrored`	Numeric calculation
`calc_star`	Numeric calculation
`write_to_dev_null`	Output
`print_matrix`	Matrix display
`deallocate`	Memory management

The application contains compiler directives which make it possible to choose for every function if it is included in the executable or not (with the exception of `allocate` and `deallocate` which always appear together). Furthermore, other compiler directives allow to introduce variations to certain functions: For `read_from_dev_urandom` and `write_to_dev_null`, it is possible to choose the granularity of the I/O (byte-wise, element-wise, row-wise, or whole matrix), if buffered I/O should be used instead of direct matrix accesses, and how the computation should be performed in `calc_star`. Here, a computation similar to the Gauß-Seidel method is performed: In a `for` loop, for each element the sum of its neighbours is computed and used as input for a mathematical function. Here, one can decide via preprocessor directives, if one should incorporate an X shape (north west, north east, south west, south east), a column shape (north and south), a row shape (west and east), the centre, or a combination of these. Depending on the choice, the memory access patterns of the for loop will be different.

A python script was written which uses a Makefile that allows fine-grained control over the several conditional compilation options and builds the prototype application using all possible combinations. Additionally, we used several different compiler options like optimisation levels and different compilers. In total, 870,352 combinations were used for compilation[1]. The purpose of this was that we hoped that the compilers would introduce variations into the resulting machine instructions if different function were present, e.g. by eliminating instructions that calculate values that are not further used, and that different optimisations amplified this effect.

We extracted the `.text` sections from the object code files, extracted the bytes of each function that was present, and removed duplicates. Each sample was labelled with one out of nine labels depending on the function that it originated from, while `allocate` and `deallocate` received the same label since they only appeared together. On these samples, classifiers were trained as described in Section 4.1.

Table 4.2 shows the testing accuracies of three classifiers. Here, the *internal testing accuracy* denotes the accuracy of the classifiers on a testing dataset after a train-test split of 4:1 has been performed, and the *external testing accuracy* denotes the accuracy on an object code file from `partdiff` which has been labelled accordingly, e.g. since `calc_star` is modelled after `partdiff`'s `calculate` function, the two received the same label. As one can see in the table, the classifiers achieved very high internal accuracies over 98 %, but performed poorly when classifying the functions from `partdiff`. We conclude that these high accuracies are a result of overfitting and that the classification does therefore not generalise. The reason for this could be that the variation inside the dataset was not large enough. In Sections 4.3.2 to 4.3.4, we used larger datasets to test the classifiers in scenarios which are closer to practical use.

---

[1] A large part of these combinations is induced by the fact that every function could be either included or not (except for `allocate` / `deallocate`), and the number of possible combinations is given by the cardinality of their power set here.

Table 4.2.: Testing accuracies of different classifiers on the prototype application dataset.

Classifier name	**Testing accuracy**	
	Internal	**External**
`RandomForestClassifier`	100.0 %	11.1 %
`MLPClassifier`	99.5 %	11.1 %
`SGDClassifier`	98.3 %	0.0 %

4.3.2. OJ Dataset

The exact effects of our source code normalisation in a real-world scenario with a mixed language setting are hard to measure. To benchmark it in an isolated context, we compared the ASTNN's performance on the OJ dataset before and after the source code normalisation, and compared the latter with object code classification methods. This is not easily possible since many samples from the OJ dataset are not compilable without further steps. Therefore we developed a source code transformation method that modifies every given sample to make it compilable. A control flow diagram of this process is given in Figure 4.4. As has been mentioned before, unlike stated by Zhang et al., many samples inside the OJ dataset are actually clearly written in C++ and not in C, and it just so happens that these samples could be parsed by `pycparser` anyway. For example, `pycparser` does not reject source code samples that use `operator<<` and many other features specific to C++.

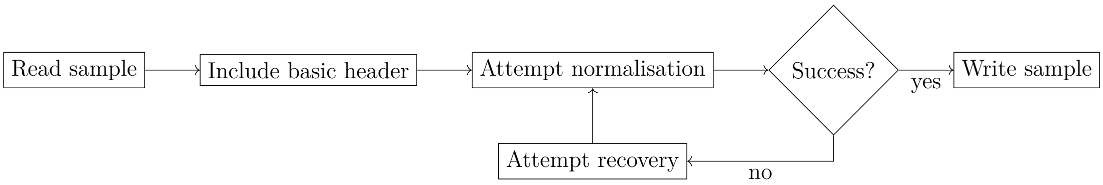


Figure 4.4.: Control flow diagram of the transformation of the OJ dataset.

As a first step, a short header gets included which can be found in Listing A.9. This file only contains some further `includes` and a statement that makes members of the `std` namespace usable without a prepended qualifier. This is actually considered a bad practice [124], but also very prominently used in the OJ dataset (the first sample of the dataset which can be found in Listing A.4 uses `std::cin` and `std::cout` without their namespace qualifiers), therefore it has been included to make samples like these easily compilable.

Afterwards, we attempt to run the normalisation pipeline shown in Figure 4.3. We are not using strict compiler settings like `-Wall -Werror`. Furthermore, we noticed that RetDec seems to hang or take an unusually large amount of time on certain samples: Usually, an extracted method was decompiled in a few seconds, but on some samples, RetDec did not finish after hours. This could be traced back to RetDec's back-end optimisations. The decompilation worked flawlessly when disabling the `CopyPropagationOptimizer` and `WhileTrueToWhileCondOptimizer`.

If the process completed successfully, i.e. the exit code is zero, we can simply use the reconstructed C source code in our modified dataset. Otherwise, we modify the source code further in an attempt to fix the errors that were raised during the normalisation process. This is done by parsing the error messages (which will in most cases come from the compiler) and selecting further steps based on the type of message. The effects of these steps vary widely, e.g. if the compiler stopped because of a missing symbol, we declare this symbol as a global variable, but upon more complicated errors where a fix is not easily retrievable from the error message itself, we also delete the line in which the error occurs. In Figure 4.5, a histogram of the number of attempts to normalise the source code can be found. Note that the y-axis is logarithmic; an excerpt of the concrete portions can be found in Table 4.3.

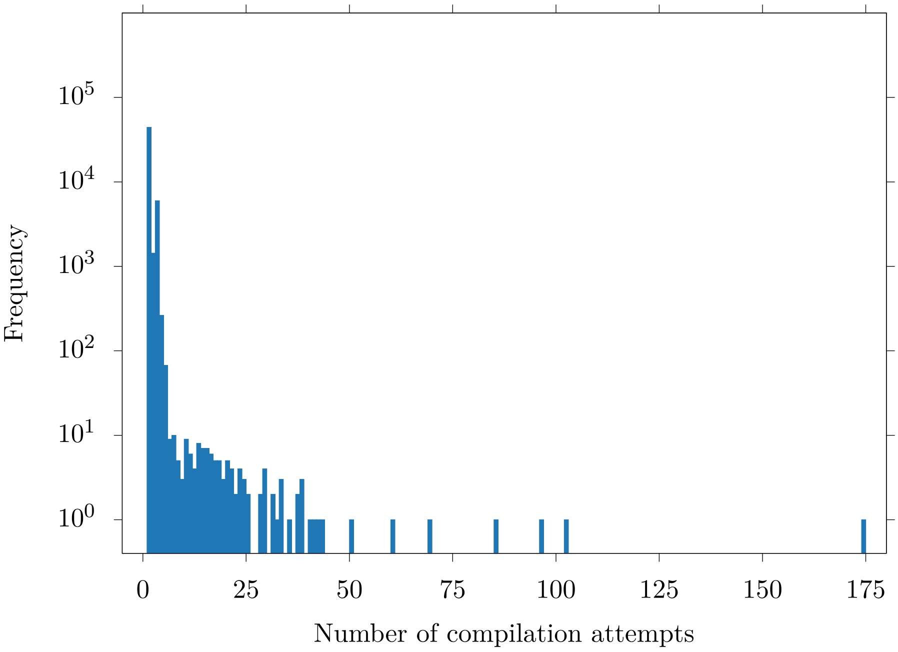


Figure 4.5.: Distribution of the number of attempted normalisations before success on the OJ dataset.

Table 4.3.: Portions of the OJ dataset that compiled after the n-th attempt.

Number of attempts	**Relative frequency**
1	84.9 %
≤ 2	87.6 %
≤ 3	99.1 %
≤ 4	99.6 %
≤ 5	99.7 %
> 5	0.3 %

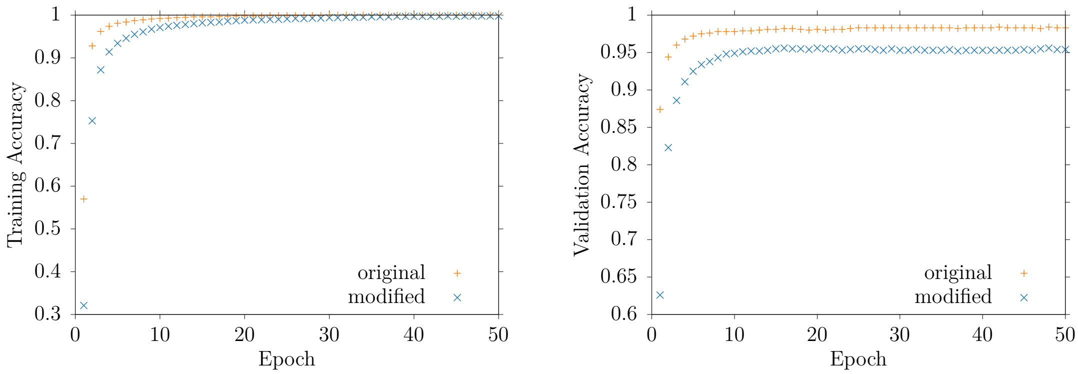
Since about 85 % of the dataset compiled after only including the header, we can assume, that no semantics were lost during the transformation of this portion. We observed that, due to the nature of the recovery process, the deletion of lines also only happens if multiple other steps have been performed unsuccessfully before, so it is probable that a very large portion of the 99.7 % of the dataset that compiled successfully after 5 attempts or less have only gained variable definitions or the like, where no semantic information was lost either. In some cases the deletion of lines has triggered a chain reaction where the complete sample gets deleted line-by-line, in which we replace the program with an empty `main` method as a last resort, but since these cases are extremely rare (there is a single sample that compiled after 174 steps), their statistical significance is negligible.

In summary, one can conclude that the semantic information of the source code samples was retained throughout the normalisation process for at least 84.9 % of the samples, but probably much closer to over 99 %.

Figure 4.6 and Figure 4.7 show the accuracy and loss on the training and validation sets during training over time. As one can see in Figure 4.7b, the validation loss decreases up to a certain point and then increases notably later on. To prevent overfitting, one should stop the training when the validation loss is at its minimum, as has been mentioned in Section 3.8.6. Even though the modified ASTNN clearly converges slower, this is the case after approximately 15 epochs. The final test accuracies are then 98.2 % for the original ASTNN and 95.8 % for the modified ASTNN and dataset which is still larger than the accuracies of compared single-language classification models, like the model by Mou et al. which achieved 94 % testing accuracy [19].

The object code files that were used as RetDec's input for the final dataset were also used for object code classification. A comparison of these methods can be found in

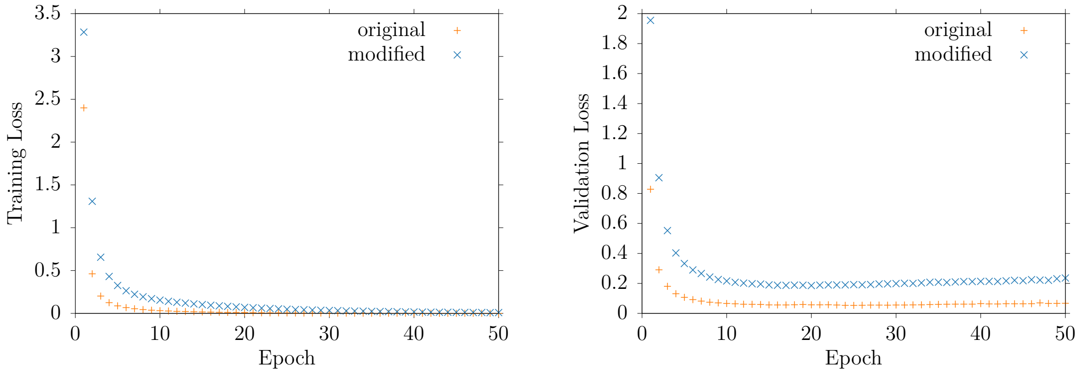
Table 4.4. As one can see therein, `sklearn`'s `GradientBoostingClassifier` was the second-best method overall, but also took longer than 48 hours for training. The third-best method was the `RandomForestClassifier` which also was the quickest with 66.7 s time cost overall, which is especially impressive in comparison with the modified ASTNN which took just under 42 hours. The third- and fourth-best methods were the `DecisionTreeClassifier` and `NuSVC`. We note that all classifiers performed substantially better than guessing (expected correctness 0.96 %[2]).



(a) Training accuracy

(b) Validation accuracy

Figure 4.6.: Training and validation accuracy of the ASTNN on the original and modified OJ dataset over time.



(a) Training loss

(b) Validation loss

Figure 4.7.: Training and validation of the ASTNN on the original and modified OJ dataset over time.

---

[2] This value comes from the fact that the OJ dataset is distributed into 104 classes and the expected correctness for guessing is the reciprocal value of this.

Table 4.4.: Testing accuracies and time costs of different classifiers on the modified OJ dataset.

Classifier name	**Testing accuracy**	**Total time cost**
Modified ASTNN	95.8 %	151 192.2 s
`GradientBoostingClassifier`	54.3 %	> 172 800 s
`RandomForestClassifier`	47.5 %	66.7 s
`DecisionTreeClassifier`	38.6 %	150.2 s
`NuSVC`	37.0 %	> 172 800 s
`MLPClassifier`	31.4 %	13 360.7 s
`SVC`	29.3 %	> 172 800 s
`LinearSVC`	25.3 %	> 172 800 s
`SGDClassifier`	24.2 %	4288.6 s
`KNeighborsClassifier`	23.8 %	146.1 s
`AdaBoostClassifier`	6.2 %	815.6 s
`GaussianNB`	2.2 %	192.0 s

4.3.3. BLAS dataset

In an attempt to benchmark the ASTNN with a Fortran code base, we tested the ASTNN on a dataset that was derived from `refblas`, the reference implementation of the numerics library specification “Basic Linear Algebra Subprograms” (BLAS) [53], which is written in Fortran 77. The library contains 163 methods (subroutines and functions) which are usually contained in one source code file each. The methods are grouped into the following categories which are referred to as *Levels*:

In the following, let $\alpha, \beta \in \mathbb{R}$, $x, y \in \mathbb{R}^n$, $A, B, C \in \mathbb{R}^{n \times m}$, and $T \in \mathbb{R}^{n \times n}$ be triangular. Then, *Level 1* [125] (50 files) contains vector operations of the form

$$y = \alpha x + y,$$

as well as vector norms and scalar products, *Level 2* [126] (66 files) contains vector-matrix operations of the form

$$y = \alpha A x + \beta y$$

and solvers for linear equation systems of the form

$$Tx = y,$$

and *Level 3* [127] (30 files) contains matrix operations of the form

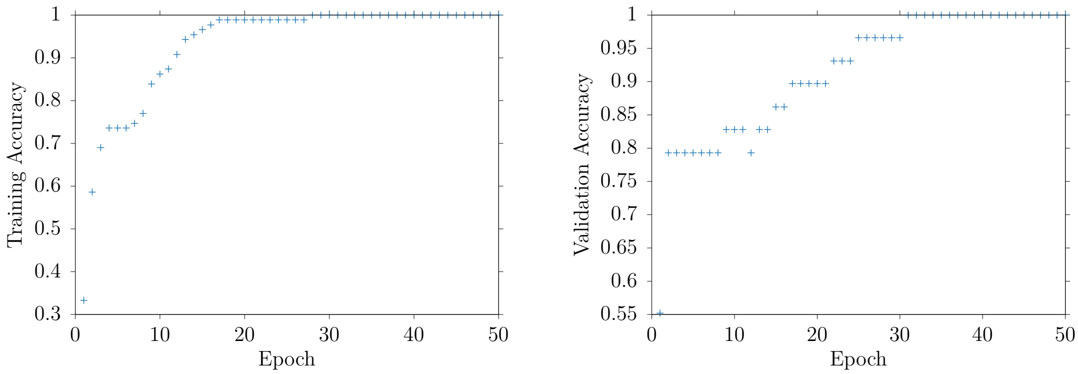
$$C = \alpha AB + \beta C$$

and solvers for linear equation systems of the form

$$TB = \alpha B.$$

Additionally, there are also 17 *Extended precision Level 2 BLAS routines* which we will not consider further.

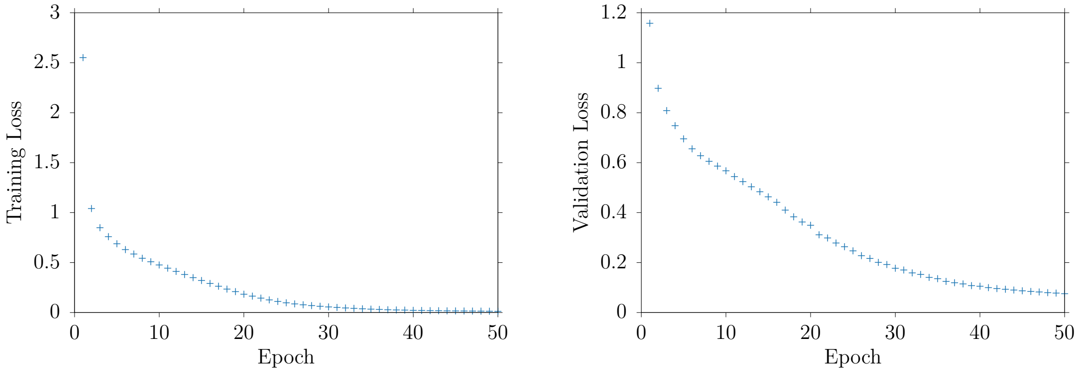
We collected the 146 methods from Level 1 to 3 in a dataset and labelled them with their corresponding level. The accuracy and loss on the training and validation sets can be found in Figure 4.8 and Figure 4.9. As one can see, both the training and validation accuracy approach 100 % around epoch 30. Likewise, the final testing accuracy is also 100 %. This means that the trained ASTNN is able to classify the BLAS dataset perfectly, but it is disputable if it would be able to decide correctly if source code samples that are not from the BLAS dataset are vector, vector-matrix, or matrix operations. However, since the validation does not increase again for the shown epochs, this overfitting could probably not have been prevented through early stopping as for the OJ dataset. Furthermore, the BLAS dataset is extremely small in comparison to the other datasets.



(a) Training accuracy

(b) Validation accuracy

Figure 4.8.: Training and validation accuracy of the ASTNN on the BLAS dataset over time.



(a) Training loss (b) Validation loss

Figure 4.9.: Training and validation of the ASTNN on the BLAS dataset over time.

Table 4.5 shows the testing results of the modified ASTNN and various classifiers from `sklearn` on the BLAS dataset. Since the dataset is very small, all classifiers were able to converge in a matter of seconds. The testing accuracies of `sklearn`'s classifiers are between 63.3 % and 83.3 %, so the overall variety of the accuracies is very small in comparison to the results for the OJ dataset. The top performer was the `MLPClassifier` (Multilayer Perceptron) which uses a neural network, and the second- to fourth-best models were `GaussianNB`, `SGDClassifier`, and `SVC`.

Table 4.5.: Testing accuracies and time costs of different classifiers on the modified BLAS dataset.

Classifier name	**Testing accuracy**	**Total time cost**
Modified ASTNN	100.0 %	8369.3 s
`MLPClassifier`	83.3 %	6.2 s
`GaussianNB`	73.3 %	2.0 s
`SGDClassifier`	73.3 %	2.2 s
`SVC`	73.3 %	3.4 s
`LinearSVC`	70.0 %	4.3 s
`GradientBoostingClassifier`	70.0 %	56.7 s
`RandomForestClassifier`	66.7 %	2.6 s
`AdaBoostClassifier`	66.7 %	7.6 s
`KNeighborsClassifier`	63.3 %	2.1 s
`DecisionTreeClassifier`	63.3 %	2.2 s
`NuSVC`	63.3 %	3.7 s

4.3.4. PHOENIX dataset

We also derived a dataset from the astrophysical simulation PHOENIX by Hauschildt et al. which simulates stellar and planetary atmospheres [128, 129].

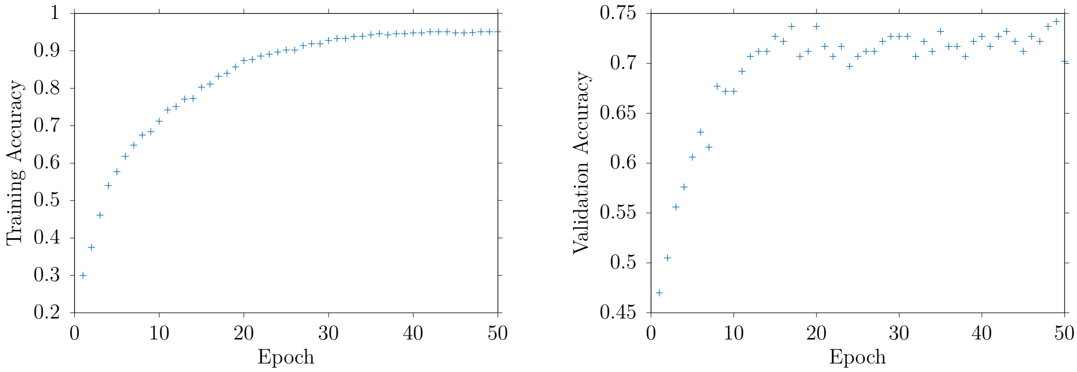
PHOENIX' source code is distributed in approximately 40 modules which are collections of source code files that are engaged with a specific part of the simulation. Examples for modules are "3DRT" (3D Radiative Transfer), "ACES" (Astrophysical Chemical Equilibrium Solver) and "SESAM(E)" (Stoichiometric Equilibrium Solver for Atoms and Molecul(E)s).

Similar to the OJ dataset which contains source code snippets that are aimed to solve specific problems, the PHOENIX modules can also be understood as collections of functions which tackle a specific sub-problem. Therefore, we compiled PHOENIX, created one sample per method that can be identified inside the object code files by `objdump`, and used the corresponding module as the label.

Since the BLAS dataset was very small in comparison to the OJ dataset, we tried to maximise the dataset size for the PHOENIX dataset while simultaneously receiving the same amount of samples per class. Since the number of methods per module varies strongly, this can *not* be done by including *all* modules into the dataset. As can be found in Appendix B, the maximum dataset size can be achieved by including 7 modules and 173 (arbitrary) methods per module which would result in 1211 samples (see Table B.1). Surprisingly, the maximum number of samples in debug mode is smaller with 1204 (see Table B.2). While the largest modules became larger (in debug mode, the modules get compiled without function inlining), the limiting factor is given by the module SESAM which loses one method in debug mode.

Just like with the OJ dataset, we noticed that the RetDec decompiler seems to hang on certain samples again. However, this time the optimisations that were causing problems were the LLVM passes in RetDec's core, and RetDec does only offer arguments to disable back-end optimisations. This concerned all samples from the MPFUN-Lapack module (which is the largest module in the optimised build mode), but also few samples from other modules. Although RetDec was restricted to certain instruction address ranges via a command line argument, we noticed that the decompilation time was also increased by the overall size of the object code file which could mean that RetDec incorporates information from the surrounding machine instructions in order to better understand the context of a segment. However, an obvious correlation between method count, method length, or file size, and the unresponsive behaviour could not be revealed. It is also worth mentioning that RetDec consistently took significantly longer on the object code files that were built in debug mode. Therefore, we only considered the optimised data, excluded the MPFUN-Lapack module completely, and used a timeout of 5 minutes for the decompilation process. This resulted in a dataset with 990 entries (6 classes with 165 samples each).

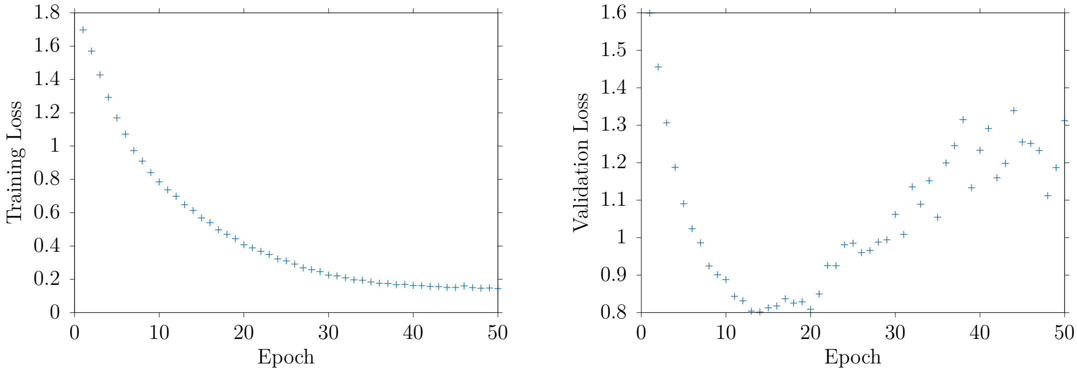
The results of the training process can be found in Figure 4.10 and Figure 4.11. Since the minimum of the validation loss is very clearly after epoch 14, it is evident that we should early-stop training here. The final testing accuracy is then 67.7 %, although the validation accuracy (71.2 %) for this epoch is unusually high in comparison which could have been caused by statistical disparities between the testing and the validation set.



(a) Training accuracy

(b) Validation accuracy

Figure 4.10.: Training and validation accuracy of the ASTNN on the PHOENIX dataset over time.



(a) Training loss

(b) Validation loss

Figure 4.11.: Training and validation of the ASTNN on the PHOENIX dataset over time.

As in the previous sections, Table 4.6 again shows a comparison of different classifiers on this dataset. Since the dataset is larger than the BLAS dataset, all classifiers took significantly longer than before. The accuracies range from 25.8 % to 48.0 %, and the four best-performing classifiers are identical to the results from Table 4.4 (OJ dataset).

Table 4.6.: Testing accuracies and time costs of different classifiers on the modified PHOENIX dataset.

Classifier name	**Testing accuracy**	**Total time cost**
Modified ASTNN	67.7 %	77 210.1 s
`GradientBoostingClassifier`	48.0 %	10 724.7 s
`RandomForestClassifier`	45.5 %	53.9 s
`DecisionTreeClassifier`	39.4 %	53.1 s
`NuSVC`	39.4 %	1079.6 s
`SGDClassifier`	37.9 %	104.6 s
`AdaBoostClassifier`	35.9 %	501.8 s
`LinearSVC`	32.8 %	13 077.8 s
`GaussianNB`	31.3 %	36.1 s
`MLPClassifier`	30.8 %	754.2 s
`SVC`	30.8 %	1074.2 s
`KNeighborsClassifier`	25.8 %	42.7 s

4.4. Problems and Limitations

There are several factors limiting the representativeness of the benchmarks described in Section 4.3 and the real-world applicability of the method described in Sections 4.1 and 4.2.

First of all, it should be mentioned that the ASTNN has an immense memory consumption. As has been mentioned in Section 3.11, its performance (in terms of time cost per epoch) on the unchanged OJ dataset has been best on a desktop PC with consumer hardware and second best on Hamburg Observatory's `igor`. For the modified OJ dataset, these systems were not available any more, because even when using a batch size of 1 and `igor`'s NVIDIA Tesla V100 GPU with 32 GB of VRAM, the neural network ran out of memory during training. Therefore, we were forced to train on the CPU and used the machine `abu1` for training which features 256 GB of

RAM which enabled us to use the default batch size of 64. The average time cost per epoch in this configuration was $(9987.6 \pm 68.0)\,\mathrm{s}$ (about 2 hours and 45 minutes), so together with the results shown in Table 3.1 we can conclude that training one epoch on the modified dataset was about 3.9 as time-consuming compared with the unchanged OJ dataset on the same machine. For the PHOENIX dataset, which contained less but longer samples, the memory problem even worsened so that `abu1`'s RAM was not enough when using the default batch size of 64. We repeatedly halved the batch size and found that the problems vanished with a batch size of 16.

In summary since two thirds of the performed benchmarks with the modified ASTNN showed memory problems, we can conclude that these problems are very likely to return for any real-world example that the method is applied on. Firstly, it is generally unpleasant that the training time is lengthened by a factor between 24 and 30 when using the CPU instead of the GPU (see Section 3.11), and secondly, when training on large datasets containing a variety of scientific codebase excerpts, it is possible that even a machine with 256 GB of RAM will not be enough, even if we reduced the batch size to 1. A possible fix for this could be taking a deep look into ASTNN's vector representations and memory management and trying to optimise these factors to preferably make it possible to use the GPU.

A problem with representativeness arises from the nature itself of the benchmarks above. In Section 5.2.2, we propose a hypothetical model that is able to classify program sections (e.g. compilation units, functions) into their suitability for different processing unit architectures and that is independent of the programming languages of the input. To train such a model, we would need a large dataset of labelled samples, i.e. a collection of source code samples, preferably written in different languages, that is labelled with the processing unit architecture they are best suited for. To our knowledge, such a dataset does not exist at the time of writing. Each benchmark above has been performed with qualitatively different, although remotely similar, labels in mind.

Furthermore, it is noteworthy that while the benchmarks above have been performed with code samples in different programming languages (before normalisation), namely C++, Fortran, and C, the datasets of each benchmark only contained one language each, and the same language for training and testing. Due to the large difference between the individual datasets, we did not succeed in coming up with a metric that could be used for a combination of them, so that we could actually have mixed languages as the input. Therefore, while we showed that our method worked well with multiple languages individually, we technically could not show that it worked well with multiple languages at once.

Finally, as has already been mentioned in Section 4.3, some of RetDec's core and back-end optimisations were seemingly hanging up during the decompilation process. For the back-end, this is not a large problem, since the relevant optimisations could be easily identified and disabled through the command line. However, for the core optimisations which were hanging up only for the PHOENIX dataset, this is not possible so that we were forced to discard the affected samples. In a real-world scenario, this would not be tolerable. It seems likely that these problems were caused by bugs inside RetDec.

In summary, one can conclude that the ASTNN in combination of the proposed source code normalisation method delivers promising results, but one would have to fix multiple flaws before it would be ready in a production environment. In contrast, as can be found in Table 4.4, `sklearn`'s RandomForestClassifier achieved about 50 % of the testing accuracy of the modified ASTNN, but was more than 2200 times as quick overall. It is possible that trying out more methods or fine-tuning the listed methods could close this gap further.

5 Chapter 5. Conclusion

In this thesis, we evaluated multiple machine learning methods for general-purpose source code classification. A qualitative comparison of the methods reviewed can be found in Section 5.1 which also gives an overview over the problems and limitations of the methods. A more detailed description of the latter can be found in Section 4.4. Section 5.2.1 lists the necessary work to solve these problems and other possible steps that could improve the classifiers or their comparability. Finally, Section 5.2.2 describes a scenario in which the evaluated classifiers could be used for automatic mapping of source code samples to adequate computation units, similar to the work of Barchi et al. [6].

5.1. Comparison of the classifiers

During all benchmarks that were performed in Section 4.3, the modified ASTNN was consistently performing the best out of all tested classifiers.

During the benchmarks with the OJ Dataset, the results of which can be found in Table 4.4, the performance of `sklearn`'s classifiers varied very strongly. The top performers are the `GradientBoostingClassifier` which achieved a testing accuracy of 54.3 % in over 48 hours, the `RandomForestClassifier` which achieved 47.5 % (about half has accurate as the modified ASTNN) in only 66.7 s (about 0.04 % of the total time cost of the ASTNN), the `DecisionTreeClassifier` (38.6 %), and `NuSVC` (37.0 %).

For the BLAS dataset (see Table 4.5), the accuracies of the different classifiers were a lot more similar which could have been caused by the small size and arguably smaller complexity of the dataset with only three classes. While the modified ASTNN achieved a testing accuracy of 100 %, the other classifiers achieved results between 63.3 % and 83.3 %. A qualitative analysis of the classifiers is therefore omitted for this benchmark.

Finally, on the PHOENIX dataset (see Table 4.6), the modified ASTNN showed its worst performance out of the three experiments with an accuracy of 67.7 %, and relative to this, the classifiers from `sklearn` performed a lot better in comparison to the other benchmarks, the four top performers being the `GradientBoostingClassifier` (48.0 %), `RandomForestClassifier` (45.5 %), `DecisionTreeClassifier`, and `NuSVC` (both 39.4 %). This order is identical to the results for the OJ dataset.

Overall, the `GradientBoostingClassifier` performed second-best in two of three benchmarks, but can also take significantly longer than the ASTNN. The `RandomForestClassifier` shows a much better ratio of accuracy per training time than both the ASTNN and the `GradientBoostingClassifier`. It is noteworthy, that the three best classifiers after the ASTNN are all tree-based. It is possible that some of the semantic structure of the program could be recognised by these classifiers and that the trees that they build internally even resemble control flow graphs of the programs on the instruction level.

Even though the modified ASTNN was consistently achieving the best testing accuracies, this approach also came with a significant amount of problems as has been described in detail in Section 4.4: Firstly, the ASTNN has an extensive demand on RAM, which forced us to train on the CPU which significantly slowed the process down, and secondly, the perceived bugs in RetDec, the decompiler that was used, made some samples unable to be used for classification, so the method is not generally applicable in its current state.

5.2. Future Work

5.2.1. Improving the Classifiers

There are several steps that could be attempted to improve the performance of the classifiers that have been evaluated in this thesis.

While the method that combined RetDec and the ASTNN showed the best results during the benchmarks in terms of accuracy, it has been shown that it is not ready for production use because of the hardware requirements and the amount of samples that it did not work on at all. First, the unresponsiveness of RetDec should be investigated. Since it occurs in the LLVM passes of RetDec's core, it would probably already be enough to add an option to disable certain passes. However, the various optimisations that RetDec performs (both in the core and the back-end) are actually one of its biggest strengths that help it to produce readable code. If expressive and short code improves the ASTNN's ability to reach a high classification accuracy, a better solution would be to find out why some of the optimisations were causing problems and to fix them. Furthermore, it has been noted that the ASTNN's immense memory consumption restricts it to machines with very high amount of RAM and makes it impossible to use the GPU at the time of writing. It should be investigated why this is the case and if it could be avoided. At the time of writing, the ASTNN relies heavily on the built-in memory management routines from PyTorch which possibly try to store more data on the GPU than necessary. Alternatively, one could also try to implement the ASTNN's design using a different machine learning library or even programming language.

Also note that all classifiers from `sklearn` have not been fine-tuned during this thesis. `sklearn` offers many customisation options for each classifier that each come with an extensive set of best-practices. However, to optimise every available option would have been outside the scope of this thesis. It is probably prudent to try to investigate the available options of the best-performing classifiers, although it is actually not guaranteed that the order of their ranking would stay the same if all classifiers achieved their theoretical maximum performance.

Furthermore, as has been stated in Section 5.1, the three best-performing classifiers overall were tree-based methods. With the `DecisionTreeClassifier`, the tree that is built up internally can be accessed via the `tree_` attribute. Similarly, the `RandomForstClassifier`, being an ensemble method, supplies the list of `DecisionTreeClassifiers` it uses as sub-estimators via the `estimators_` attribute. The `GradientBoostingClassifier` supplies the same attribute, but uses `DecisionTreeRegressors` instead. It is possible that one could learn interesting information about the syntactical structure of low-level program representations upon closer analysis of these trees. It is however clear that this would have been beyond the scope of this thesis, since their nodes only contain boolean decisions of numerical values.

Finally, the classification result of the ASTNN for a source code sample is determined by calculating the probability for each class that the sample belongs to it and then choosing the class with the highest possibility. There are several `sklearn` classifiers with a similar functional principle: For example, the `SGDClassifier` trains n binary classifiers for n classes which each decide a class probability. This implies that these classifiers also have a “second choice”, i.e. the class with the second-highest probability. In a scenario where the number classes is larger than the number of targetted computation units, it is probably a good idea to also take these choices into account.

5.2.2. Automatic Code Mapping

The source code classifiers that were evaluated in this thesis could also be used for automatic code mapping similar to the work of Barchi et al. [6]. In their work, a dataset of OpenCL kernels was created and their execution time was measured on both CPUs and GPUs. The difference between these values was used to train a deep neural network which could then decide the optimal computational unit for a new OpenCL sample.

A fundamental flaw with this approach is that this result can not be used in the context of a whole program without hesitation, because it does not consider how many times the individual kernels are executed at runtime. As an example, one could consider two OpenCL kernels A and B which have a similar execution time on the CPU and of which only one could be offloaded to the GPU because of resource limitations. If A runs twice as fast on the GPU but gets used only once during an application run, and B runs only 10 % faster on the GPU, but gets called a million times, it would be preferable to run A on the CPU and B on the GPU. Instead, an approach that only considers the performance gain per kernel would make the opposite decision. This illustrates that more information about the whole program is necessary for this decision. Static loop analysis could help in this situation, but in general, this decision is undecidable, because loop bounds frequently depend on input values in scientific software, and finding out how many times each sub-program of an application can be called could be used to solve the halting problem [130]. It is however possible to run an application in a defined configuration that is regarded typical for production use and extract the required data from profiling results.

The second disadvantage of the method from Barchi et al. is that it is limited to OpenCL kernels. Although OpenCL can be used to write code for various target architectures, it is less popular than CUDA for GPU applications and not a widespread choice for CPU-only applications[1]. In contrast, the methods discussed in this thesis can be used for all programming languages that are ahead-of-time compiled. However, this also makes it extremely more difficult to find a training dataset that could be used to train them. Such a dataset could be created from a large collection of different scientific software codebases. To facilitate generalisability, it should be ensured that multiple languages are included, and that all code patterns frequent to HPC as described by Asanović et al. and Colella (see Section 3.3) are represented. Similarly to Barchi et al., one could use profiling to find out the optimal architecture for each training sample.

To automatically map compilation units of programs to the optimal target platform, one could rely on automatic parallelisation tools as described in Section 3.2, primarily tools that perform the parallelisation without the help of the programmer, like Polly (see Section 3.2.3). Since OpenMP works for traditional and many-core CPUs and GPUs via accelerator offloading, it could be used as a back-end for these architectures. Additionally, the NEC compiler also supports OpenMP [38] which is useful in case that the source code classifier has predicted that a piece of source code is best suited for vector processors.

Since it is already common to compile scientific software on the cluster computer that it is meant to be run on[2], a compilation process that takes into account the available hardware and the optimal computational units of the individual compilation units, a process could be developed that maps a program to a heterogeneous cluster. During this process, one could also split up compilation units in multiple parts to increase the granularity of this process while ensuring that certain functions are executed on the same hardware. This could go as far as one function per compilation method, although this could again reduce maintainability and would also require link-time optimisation to produce efficient code.

---

[1] At the time of writing, there are 1,817,848 repositories on Github [131] which contain C/C++ code and out of this portion, 4,439 repositories contain the word OpenCL and 12,881 repositories contain the word CUDA.

[2] On many clusters, this is preferably done on dedicated compile or login nodes, the regulations vary between the systems.

Bibliography

[1] Joel Spolsky. *Things You Should Never Do, Part I – Joel on Software*. 2000-04-06. URL: https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/ (visited on 2021-05-23).

[2] Donald E. Knuth. “Structured Programming with Go to Statements”. In: *ACM Comput. Surv.* 6.4 (1974-12), pp. 261–301. ISSN: 0360-0300. DOI: 10.1145/356635.356640. URL: https://doi.org/10.1145/356635.356640.

[3] *Cori – NERSC Documentation.* URL: https://docs.nersc.gov/systems/cori/ (visited on 2021-05-23).

[4] *Our Facility – ASPIRE 1 – NSCC.* URL: https://www.nscc.sg/2020/about-nscc/our-facilityaspire-1/ (visited on 2021-05-23).

[5] *Configuration — User Portal.* URL: https://www.dkrz.de/up/systems/mistral/configuration (visited on 2021-05-23).

[6] Francesco Barchi et al. “Code Mapping in Heterogeneous Platforms Using Deep Learning and LLVM-IR”. In: 2019-06, pp. 1–6. DOI: 10.1145/3316781.3317789.

[7] Krste Asanović et al. *The Landscape of Parallel Computing Research: A View from Berkeley.* Tech. rep. UCB/EECS-2006-183. EECS Department, University of California, Berkeley, 2006-12. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

[8] Phillip Colella. *Defining software requirements for scientific computing.* 2004.

[9] *OpenMP Fortran Application Program Interface.* 1997-10. URL: https://www.openmp.org/wp-content/uploads/fspec10.pdf (visited on 2021-07-31).

[10] *OpenMP C and C++ Application Program Interface.* 1998-10. URL: https://www.openmp.org/wp-content/uploads/cspec10.pdf (visited on 2021-07-31).

[11] *The OpenACC Application Programming Interface Version 2.5.* 2015-10. URL: https://www.openacc.org/sites/default/files/inline-files/OpenACC_2pt5_0.pdf (visited on 2021-06-24).

[12] Aaftab Munshi. “The OpenCL Specification”. In: *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE. 2009, pp. 1–314.

[13] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. “Kokkos: Enabling Manycore Performance Portability Through Polymorphic Memory Access Patterns”. In: *Journal of Parallel and Distributed Computing* 74.12 (2014). Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, pp. 3202–3216. ISSN: 0743-7315. DOI: `https://doi.org/10.1016/j.jpdc.2014.07.003`. URL: `http://www.sciencedirect.com/science/article/pii/S0743731514001257`.

[14] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. “Polly – Performing Polyhedral Optimizations on a Low-Level Intermediate Representation”. In: *Parallel Processing Letters* 22.04 (2012).

[15] Michael Kruse. *Introducing Molly: Distributed Memory Parallelization with LLVM.* 2014. arXiv: `1409.2088 [cs.PL]`.

[16] Jannek Squar et al. “Compiler Assisted Source Transformation of OpenMP Kernels”. In: *2020 19th International Symposium on Parallel and Distributed Computing (ISPDC)*. 2020, pp. 44–51. DOI: `10.1109/ISPDC51135.2020.00016`.

[17] Mason McDaniel and Mohammad Heydari. “Content Based File Type Detection Algorithms.” In: *36th Annual Hawaii International Conference on System Sciences, 2003.* Vol. 9. IEEE. 2003-01, p. 332. DOI: `10.1109/HICSS.2003.1174905`.

[18] John Clemens. “Automatic Classification of Object Code Using Machine Learning”. In: *Digital Investigation* 14 (2015-08), S156–S162. DOI: `10.1016/j.diin.2015.05.007`.

[19] Lili Mou et al. *Convolutional Neural Networks over Tree Structures for Programming Language Processing.* 2015. arXiv: `1409.5718 [cs.LG]`.

[20] Jian Zhang et al. “A Novel Neural Source Code Representation Based on Abstract Syntax Tree”. In: *Proceedings of the 41st International Conference on Software Engineering.* ICSE ’19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 783–794. DOI: `10.1109/ICSE.2019.00086`. URL: `https://doi.org/10.1109/ICSE.2019.00086`.

[21] Hoa Khanh Dam et al. *A Deep Tree-Based Model for Software Defect Prediction.* 2018. arXiv: `1802.00921 [cs.SE]`.

[22] J. Křoustek. “Retargetable Analysis of Machine Code”. PhD thesis. Faculty of Information Technology, Brno University of Technology, CZ, 2015, p. 190.

[23] Richard M. Russell. “The CRAY-1 Computer System”. In: *Commun. ACM* 21.1 (1978-01), pp. 63–72. ISSN: 0001-0782. DOI: `10.1145/359327.359336`. URL: `https://doi.org/10.1145/359327.359336`.

[24] “The Evolution of HPC”. In: insideHPC Guide to Co-Design Architectures – Designing Machines Around Problems. The Co-Design Push to Exascale 2 (2016-08).

[25] Jack Dongarra and Piotr Luszczek. “TOP500”. In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 2055–2057. ISBN: 978-0-387-09766-4. DOI: `10.1007/978-0-387-09766-4_157`. URL: `https://doi.org/10.1007/978-0-387-09766-4_157`.

[26] *November 2020 – TOP500*. 2020-11. URL: `https://www.top500.org/lists/top500/2020/11/` (visited on 2021-06-21).

[27] *AMD Instinct™ MI100 Accelerator – Data Center GPU – AMD*. URL: `https://www.amd.com/en/products/server-accelerators/instinct-mi100` (visited on 2021-06-21).

[28] *AMD Threadripper 3990x 64-core Linpack and NAMD Performance (Linux)*. 2020-02. URL: `https://www.pugetsystems.com/labs/hpc/AMD-Threadripper-3990x-64-core-Linpack-and-NAMD-Performance-Linux-1666/` (visited on 2021-06-21).

[29] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010. ISBN: 0123814723.

[30] Felix Weninger, Johannes Bergmann, and Björn Schuller. “Introducing currennt: The munich open-source cuda recurrent neural network toolkit”. In: *Journal of Machine Learning Research* (2015), pp. 547–551.

[31] *Intel Xeon Phi Processor 7295 16GB 1.5 GHz 72 Core Product Specifications*. URL: `https://ark.intel.com/content/www/us/en/ark/products/128690/intel-xeon-phi-processor-7295-16gb-1-5-ghz-72-core.html` (visited on 2021-08-01).

[32] *Intel Quietly Kills Off Xeon Phi – ExtremeTech*. 2019-05-18. URL: `https://www.extremetech.com/extreme/290963-intel-quietly-kills-off-xeon-phi` (visited on 2021-06-21).

[33] *Intel Unveils New Product Plans for High-Performance Computing*. 2010-05-31. URL: `https://www.intel.com/pressroom/archive/releases/2010/20100531comp.htm` (visited on 2021-06-21).

[34] *Intel Xeon Phi Processor 7290 16GB 1.50 GHz 72 core Product Specifications*. URL: `https://ark.intel.com/content/www/us/en/ark/products/95830/intel-xeon-phi-processor-7290-16gb-1-50-ghz-72-core.html` (visited on 2021-08-01).

[35] *Intel's Silvermont Architecture Revealed: Getting Serious About Mobile.* 2013-05-06. URL: `https://www.anandtech.com/show/6936/intels-silvermont-architecture-revealed-getting-serious-about-mobile` (visited on 2021-07-31).

[36] *NEC SX-Aurora TSUBASA – Vector Engine.* URL: `https://www.nec.com/en/global/solutions/hpc/sx/vector_engine.html` (visited on 2021-06-22).

[37] *SX-Aurora TSUBASA A412-8, Vector Engine Type10AE 8C 1.58GHz, InfiniBand HDR 100 – TOP500.* 2020-11. URL: `https://www.top500.org/system/179872/` (visited on 2021-06-22).

[38] *NEC SX Aurora TSUBASA – VSC documentation.* URL: `https://vlaams-supercomputing-centrum-vscdocumentation.readthedocs-hosted.com/en/latest/antwerp/uantwerp_SX_Aurara_TSUBASA.html` (visited on 2021-06-22).

[39] Peter B Galvin, Greg Gagne, Abraham Silberschatz, et al. *Operating system concepts.* John Wiley & Sons, 2003.

[40] *OpenMP Accelerator Support for GPUs – OpenMP.* URL: `https://www.openmp.org/updates/openmp-accelerator-support-gpus/` (visited on 2021-06-22).

[41] G. Madey et al. "Agent-Based Scientific Simulation". In: *Computing in Science & Engineering* 2.01 (2005-01), pp. 22–29. ISSN: 1558-366X. DOI: `10.1109/MCSE.2005.7`.

[42] *Using Polly with Clang — Polly 13.0-devel documentation.* URL: `https://polly.llvm.org/docs/UsingPollyWithClang.html` (visited on 2021-07-14).

[43] Brandon Barker. "Message Passing Interface (MPI)". In: *Workshop: High Performance Computing on Stampede.* Vol. 262. 2015.

[44] Richard L Graham, Timothy S Woodall, and Jeffrey M Squyres. "Open MPI: A flexible high performance MPI". In: *International Conference on Parallel Processing and Applied Mathematics.* Springer. 2005, pp. 228–239.

[45] Dhabaleswar K Panda et al. "The MVAPICH project: Evolution and Sustainability of an Open Source Production Quality MPI Library for HPC". In: *Workshop on Sustainable Software for Science: Practice and Experiences, held in conjunction with International Conference on Supercomputing (WSSPE).* 2013.

[46] Gabriele Jost et al. "Comparing the OpenMP, MPI, and Hybrid Programming Paradigms on an SMP Cluster". In: *Proceedings of EWOMP.* Vol. 3. 2003, p. 2003.

[47] Michael Kruse. *Molly: Parallelizing for Distributed Memory using LLVM.* Talk. 2016-03.

[48] Charles Pierre Trémaux. “École polytechnique of Paris (X: 1876)”. In: *French Engineer of the Telegraph in Public Conference, December*. Vol. 2. 2010, pp. 0980–603.

[49] Thomas H Cormen et al. *Introduction to Algorithms*. MIT press, 2009.

[50] Mary Hall, David Padua, and Keshav Pingali. “Compiler Research: The Next 50 Years”. In: *Communications of the ACM* 52.2 (2009), pp. 60–67.

[51] Norman S Clerman and Walter Spector. *Modern Fortran: Style and Usage*. Cambridge University Press, 2011.

[52] Ronald F Boisvert et al. “Mathematical Software: Past, Present, and Future”. In: *Computational Science, Mathematics and Software*. Purdue Univ. Press, 2002, pp. 3–27.

[53] *BLAS (Basic Linear Algebra Subprograms)*. URL: `http://www.netlib.org/blas/` (visited on 2021-06-06).

[54] Andrew S Tanenbaum and Herbert Bos. *Modern Operating Systems*. Pearson, 2015.

[55] Robert Schaback and Holger Wendland. *Numerische Mathematik*. Springer-Verlag, 2006.

[56] *PYPL PopularitY of Programming Language index*. URL: `https://pypl.github.io/PYPL.html` (visited on 2021-05-23).

[57] Vasco Amaral et al. “Programming Languages for Data-Intensive HPC Applications: a Systematic Mapping Study”. In: *Parallel Computing* (2019-11). DOI: `10.1016/j.parco.2019.102584`.

[58] V. Vassilev et al. “Cling – The New Interactive Interpreter for ROOT 6”. In: vol. 396. 5. IOP Publishing, 2012-12, p. 052071. DOI: `10.1088/1742-6596/396/5/052071`. URL: `https://iopscience.iop.org/article/10.1088/1742-6596/396/5/052071/pdf`.

[59] *Implementing ART Just-In-Time (JIT) Compiler*. URL: `https://source.android.com/devices/tech/dalvik/jit-compiler` (visited on 2021-05-23).

[60] *Android Runtime (ART) and Dalvik – Android Open Source Project*. URL: `https://source.android.com/devices/tech/dalvik` (visited on 2021-05-23).

[61] Carl Friedrich Bolz et al. “Tracing the meta-level: PyPy’s tracing JIT compiler”. In: *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. 2009, pp. 18–25.

[62] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. “Numba: A LLVM-Based Python JIT Compiler”. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 2015, pp. 1–6.

[63] *Future Tense*. 2011-04-29. URL: `https://www.slideshare.net/BrendanEich/future-tense-7782010` (visited on 2021-05-23).

[64] *Rust Programming Language*. URL: `https://www.rust-lang.org/` (visited on 2021-05-23).

[65] Michal Sudwoj. "Rust Programming Language in the High-Performance Computing Environment". en. MA thesis. Zurich: ETH Zurich, 2020-09. DOI: `10.3929/ethz-b-000474922`.

[66] Alexander Droste, Michael Kuhn, and Thomas Ludwig. "MPI-Checker – Static Analysis for MPI". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM '15. Austin, Texas, USA: ACM, 2015-11. ISBN: 978-1-4503-4005-2. DOI: `http://doi.acm.org/10.1145/2833157.2833159`. URL: `http://llvm-hpc2-workshop.github.io/`.

[67] John R. Levine. *Linkers and Loaders*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999. ISBN: 1558604960.

[68] Wai Kyi, Hiroshi Koide, and Kouichi Sakurai. "Analyzing the Effect of Moving Target Defense for a Web System". In: *International Journal of Networking and Computing* 9 (2019-07), pp. 188–200. DOI: `10.15803/ijnc.9.2_188`.

[69] Andrew W. Appel and Maia Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, 1997. DOI: `10.1017/CBO9781139174930`.

[70] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998. ISBN: 1558603204.

[71] *flang-compiler/flang: Flang is a Fortran language front-end designed for integration with LLVM*. URL: `https://github.com/flang-compiler/flang` (visited on 2021-05-25).

[72] *Home – flang-compiler/flang Wiki*. URL: `https://github.com/flang-compiler/flang/wiki` (visited on 2021-05-25).

[73] Jozef Kolek et al. "Adding microMIPS backend to the LLVM compiler infrastructure". In: *2013 21st Telecommunications Forum Telfor (TELFOR)* (2013), pp. 1015–1018.

[74] Andrew Johnson-Laird. "Software Reverse Engineering in the Real World". In: *U. Dayton l. rev.* 19 (1993), p. 843.

[75] Milan Stevanovic. "Linux toolbox". In: *Advanced C and C++ Compiling*. Springer, 2014, pp. 243–276.

[76] J. Křoustek. "On Decompilation of VLIW Executable Files". In: *Scientific Journal Problems in Programming* 16.1 (2015), pp. 29–37.

[77] J. Křoustek and P. Matula. *RetDec: An Open-Source Machine-Code Decompiler*. [talk]. Presented at Pass the SALT 2018, Lille, FR. 2018-07.

[78] *Capstone2LlvmIr – avast/retdec Wiki.* URL: `https://github.com/avast/retdec/wiki/Capstone2LlvmIr` (visited on 2021-07-14).

[79] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics).* Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387310738.

[80] Douglas M Hawkins. "The Problem of Overfitting". In: *Journal of chemical information and computer sciences* 44.1 (2004), pp. 1–12.

[81] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* `http://www.deeplearningbook.org`. MIT Press, 2016.

[82] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[83] N Gill, E LeDell, and Y Tang. *H2O4GPU: Machine Learning with GPUs in R and Python.*

[84] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32.* Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[85] François Chollet et al. *Keras.* `https://keras.io`. 2015.

[86] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/`.

[87] Evelyn Fix. *Discriminatory Analysis: Nonparametric Discrimination, Consistency Properties.* Vol. 1. USAF school of Aviation Medicine, 1985.

[88] U. Ravi Babu, Y. Venkateswarlu, and Aneel Kumar Chintha. "Handwritten Digit Recognition Using K-Nearest Neighbour Classifier". In: *2014 World Congress on Computing and Communication Technologies.* 2014, pp. 60–65. DOI: `10.1109/WCCCT.2014.7`.

[89] J. R. Quinlan. "Induction of Decision Trees". In: *Machine Learning* 1 (1986), pp. 81–106.

[90] L. Breiman et al. *Classification and Regression Trees.* Taylor & Francis, 1984. ISBN: 9780412048418.

[91] James Joyce. "Bayes' Theorem". In: *The Stanford Encyclopedia of Philosophy.* Ed. by Edward N. Zalta. Spring 2019. Metaphysics Research Lab, Stanford University, 2019.

[92] Harry Zhang. "The Optimality of Naive Bayes". In: *AA* 1.2 (2004), p. 3.

[93] Johan Hovold. “Naive Bayes Spam Filtering Using Word-Position-Based Attributes.” In: *CEAS*. 2005, p. 41.

[94] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. “A Training Algorithm for Optimal Margin Classifiers”. In: *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory*. ACM Press, 1992, pp. 144–152.

[95] Alex J. Smola and Bernhard Schölkopf. *A Tutorial on Support Vector Regression*. 2004.

[96] Lior Rokach. “Ensemble-Based Classifiers”. In: *Artificial Intelligence Review* 33.1 (2010), pp. 1–39.

[97] Stephen Cole Kleene. *Representation of Events in Nerve Nets and Finite Automata*. Princeton University Press, 2016.

[98] Warren S McCulloch and Walter Pitts. “A Logical Calculus of the Ideas Immanent in Nervous Activity”. In: *The Bulletin of Mathematical Biophysics* 5.4 (1943), pp. 115–133.

[99] Frank Rosenblatt. “The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain.” In: *Psychological Review* 65.6 (1958), p. 386.

[100] Paul Werbos. “Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences”. PhD thesis. Harvard University, 1974.

[101] Danilo Mandic and Jonathon Chambers. *Recurrent Neural Networks for Prediction: Learning Algorithms, Architectures and Stability*. Wiley, 2001.

[102] Mike Schuster and Kuldip K Paliwal. “Bidirectional Recurrent Neural Networks”. In: *IEEE Transactions on Signal Processing* 45.11 (1997), pp. 2673–2681.

[103] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

[104] Sepp Hochreiter. “The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions”. In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6.02 (1998), pp. 107–116.

[105] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. *Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks*. 2015. arXiv: `1503.00075 [cs.CL]`.

[106] Thai Thien. *Pytorch Implementation of Sentiment Classification in Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks*. 2017. URL: `https://github.com/ttpro1995/TreeLSTMSentiment`.

[107] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning Representations by Back-Propagating Errors”. In: *Nature* 323.6088 (1986-10), pp. 533–536. DOI: `10.1038/323533a0`.

[108] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: `1301.3781 [cs.CL]`.

[109] Tomas Mikolov et al. *Distributed Representations of Words and Phrases and their Compositionality*. 2013. arXiv: `1310.4546 [cs.CL]`.

[110] Kenneth Ward Church. “Word2Vec”. In: *Natural Language Engineering* 23.1 (2017), pp. 155–162.

[111] Igor Popov. “Malware Detection Using Machine Learning Based on word2vec Embeddings of Machine Code Instructions”. In: *2017 Siberian Symposium on Data Science and Engineering (SSDSE)*. IEEE. 2017, pp. 1–4.

[112] Tieming Chen et al. “Droidvecdeep: Android malware detection based on Word2Vec and deep belief network”. In: *KSII Transactions on Internet and Information Systems (TIIS)* 13.4 (2019), pp. 2180–2197.

[113] *Learning C++ if you already know C, C++ FAQ*. URL: `https://isocpp.org/wiki/faq/c#is-c-a-subset` (visited on 2021-05-29).

[114] *eliben/pycparser: Complete C99 parser in pure Python*. URL: `https://github.com/eliben/pycparser` (visited on 2021-05-27).

[115] *zhangj111/astnn*. URL: `https://github.com/zhangj111/astnn` (visited on 2021-07-10).

[116] *NVIDIA Tesla V100 PCIe 32 GB review: GPU specs, performance benchmarks*. URL: `https://askgeek.io/en/gpus/NVIDIA/Tesla-V100-PCIe-32-GB` (visited on 2021-07-21).

[117] *NVIDIA GeForce RTX 2060 review: GPU specs, performance benchmarks*. URL: `https://askgeek.io/en/gpus/NVIDIA/GeForce-RTX-2060` (visited on 2021-07-21).

[118] *Geekbench 5 – Cross-Platform Benchmark*. URL: `https://www.geekbench.com/` (visited on 2021-07-21).

[119] *sklearn.preprocessing.StandardScaler — scikit-learn 0.24.2 documentation*. URL: `https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html` (visited on 2021-07-08).

[120] *Operator Overloading, C++ FAQ*. URL: `https://isocpp.org/wiki/faq/operator-overloading#matrix-subscript-op` (visited on 2021-05-26).

[121] Thomas Ludwig and Andreas C. Schmidt. *partdiff – Partial differential equation solver for Gauß-Seidel and Jacobi method*.

[122] Siméon-Denis Poisson. *Mémoire sur la théorie du magnétisme en movement*. L’Académie, 1826.

[123] Carl Friedrich Gauss. “Werke (in german), 9”. In: *Göttingen: Köninglichen Gesellschaft der Wissenschaften* 763 (1903), p. 764.

[124] *Coding Standards, C++ FAQ.* URL: `https://isocpp.org/wiki/faq/coding-standards#using-namespace-std` (visited on 2021-05-27).

[125] C. L. Lawson et al. “Basic Linear Algebra Subprograms for Fortran Usage”. In: *ACM Trans. Math. Softw.* 5.3 (1979-09), pp. 308–323. ISSN: 0098-3500. DOI: `10.1145/355841.355847`. URL: `https://doi.org/10.1145/355841.355847`.

[126] Jack J. Dongarra et al. “An Extended Set of FORTRAN Basic Linear Algebra Subprograms”. In: *ACM Trans. Math. Softw.* 14.1 (1988-03), pp. 1–17. ISSN: 0098-3500. DOI: `10.1145/42288.42291`. URL: `https://doi.org/10.1145/42288.42291`.

[127] J. J. Dongarra et al. “A Set of Level 3 Basic Linear Algebra Subprograms”. In: *ACM Trans. Math. Softw.* 16.1 (1990-03), pp. 1–17. ISSN: 0098-3500. DOI: `10.1145/77626.79170`. URL: `https://doi.org/10.1145/77626.79170`.

[128] P. H. Hauschildt, E. Baron, and F. Allard. “Parallel Implementation of the PHOENIX Generalized Stellar Atmosphere Program”. In: *ApJ* 483 (1997), p. 390.

[129] P. H. Hauschildt and E. Baron. “Numerical Solution of the Expanding Stellar Atmosphere Problem”. In: *Journal of Computational and Applied Mathematics* 109 (1999), pp. 41–63.

[130] Martin Davis. *Computability and Unsolvability.* Courier Corporation, 2013.

[131] *GitHub.* URL: `https://github.com/` (visited on 2021-07-31).

[132] *Supervised learning – scikit-learn 0.24.2 documentation.* URL: `https://scikit-learn.org/stable/supervised_learning.html` (visited on 2021-06-08).

Appendix A.

Source Code Samples

```c
#include <stdlib.h>

#define MAX 1024

void expensive_function(int i) {
    // ...
}

int main(int argc, char const *argv[]) {
    for (int i = 0; i < MAX; i++) {
        expensive_function(i);
    }
    return EXIT_SUCCESS;
}
```

Listing A.1: Example for a simple program where a computationally expensive function gets executed in a loop.

```c
#include <pthread.h>
#include <stdlib.h>

#define MAX 1024
#define NUM_THREADS 16

void expensive_function(int i) {
    // ...
}

struct args_t {
    int begin;
    int end;
};

void *thread_fun(void *args) {
    int begin = ((struct args_t *)args)->begin;
    int end = ((struct args_t *)args)->end;
    for (int i = begin; i < end; i++) {
        expensive_function(i);
    }
    return NULL;
}

int main(int argc, char const *argv[]) {
    struct args_t args[NUM_THREADS];
    pthread_t threads[NUM_THREADS];
    for (int i = 0; i < NUM_THREADS; i++) {
        args[i].begin = i * MAX / NUM_THREADS;
        args[i].end = (i + 1) * MAX / NUM_THREADS;
        pthread_create(&threads[i], NULL, &thread_fun, &args[i]);
    }
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    return EXIT_SUCCESS;
}
```

Listing A.2: The program from Listing A.1 parallelised with POSIX threads.

```
#include <omp.h>
#include <stdlib.h>

#define MAX 1024
#define NUM_THREADS 16

void expensive_function(int i) {
    // ...
}

int main(int argc, char const *argv[]) {
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel for
    for (int i = 0; i < MAX; i++) {
        expensive_function(i);
    }
    return EXIT_SUCCESS;
}
```

Listing A.3: The program from Listing A.1 parallelised with OpenMP.

```
int main()
{
    int a;
    int bai,wushi,ershi,shi,wu,yi;
    cin>>a;
    bai=a/100;
    a=a%100;
    wushi=a/50;
    a=a%50;
    ershi=a/20;
    a=a%20;
    shi=a/10;
    a=a%10;
    wu=a/5;
    a=a%5;
    yi=a;
    cout<<bai<<endl;
    cout<<wushi<<endl;
    cout<<ershi<<endl;
    cout<<shi<<endl;
    cout<<wu<<endl;
    cout<<yi<<endl;
    return 0;
}
```

Listing A.4: First sample from the OJ dataset.

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    printf("Hello World!\n");
    return EXIT_SUCCESS;
}
```

Listing A.5: Example *Hello World* program written in C.

```cpp
#include <iostream>

int main() {
    std::cout << "Hello World!" << std::endl;
    return EXIT_SUCCESS;
}
```

Listing A.6: Example *Hello World* program written in C++.

```c
#include <stdint.h>
#include <stdio.h>

int64_t printf(char * a1);

int main(int argc, char ** argv) {
    printf("Hello World!\n");
    return 0;
}
```

Listing A.7: Program from Listing A.5 after compilation and subsequent decompilation to C.

```
#include <stdint.h>

int64_t __cxa_atexit(int64_t a1, char *a2, int64_t *a3);
int64_t __cxx_global_var_init(void);
int64_t _GLOBAL__sub_I_helloworld_cpp(void);
int64_t _ZNSolsEPFRSoS_E(int64_t *a1, int64_t *(*a2)(int64_t *));
int64_t _ZNSt8ios_base4InitC1Ev(int64_t *a1);
int64_t *_ZStlsISt11char_traitsIcEERSt13basic_↲
    ↪ ostreamIcT_ES5_PKc(int64_t *a1, char *a2);

int64_t g1 = 0;

int main(int argc, char **argv) {
  int64_t v1 = *(int64_t *)15;
  int64_t *v2 = _ZStlsISt11char_traitsIcEERSt13basic_↲
      ↪ ostreamIcT_ES5_PKc((int64_t *)v1, "Hello World!");
  _ZNSolsEPFRSoS_E(v2, (int64_t * (*)(int64_t *)) * (int64_t
      ↪ *)41);
  return 0;
}

int64_t __cxx_global_var_init(void) {
  _ZNSt8ios_base4InitC1Ev((int64_t *)"Hello World!");
  return __cxa_atexit(*(int64_t *)87, "Hello World!", &g1);
}

int64_t _GLOBAL__sub_I_helloworld_cpp(void) {
  return __cxx_global_var_init();
}
```

Listing A.8: Program from Listing A.6 after compilation and subsequent decompilation to C (lines containing very long identifiers have been broken).

```
#include <iostream>
#include <cmath>
#include <cstring>
#include <cassert>

using namespace std;
```

Listing A.9: Source code header that has been used for the source code recovery process described in Section 4.3.2.

B

Appendix B.

PHOENIX dataset module selection

The following two tables contain an overview over the PHOENIX modules and the number of modules that are contained within them. The lists are ordered by the number of methods per module in descending order. The rightmost column contains the product of the index (leftmost column) and the number of methods in the module. For each module, this is the total number of methods that would be contained in the dataset if one included this module and all modules above into it.

Table B.1.: PHOENIX modules and their method counts in optimised build mode.

Index	**Module name**	**Number of methods in module**	**Total number of methods in dataset**
1	MPFUN-Lapack	315	315
2	MPFUN-MPFR	289	578
3	3DRT	253	759
4	Cassandra	184	736
5	KAPCAL	182	910
6	SESAM	173	1038
7	MISC	173	1211
8	Sybil	97	776
9	S3R2T	85	765
10	NMS3	73	730
11	LTELINES	66	726
12	FPPRESS	66	792
13	DRIFT	57	741
14	NLTE	34	476
15	ACES	34	510
16	DISK	30	480
17	GR2T	27	459
18	NM_3DRT	20	360
19	Hiro	17	323
20	SubSlatec	15	300
21	FUZZ	5	105
22	OpenCL	1	22

Table B.2.: PHOENIX modules and their method counts in debug build mode.

Index	**Module name**	**Number of methods in module**	**Total number of methods in dataset**
1	3DRT	402	402
2	MPFUN-Lapack	315	630
3	Cassandra	308	924
4	MPFUN-MPFR	289	1156
5	KAPCAL	183	915
6	MISC	176	1056
7	SESAM	172	1204
8	LTELINES	116	928
9	Sybil	106	954
10	S3R2T	85	850
11	NMS3	81	891
12	FPPRESS	66	792
13	DRIFT	54	702
14	NM_3DRT	42	588
15	ACES	42	630
16	NLTE	36	576
17	GR2T	31	527
18	DISK	30	540
19	Hiro	17	323
20	SubSlatec	15	300
21	FUZZ	5	105
22	OpenCL	1	22

C

Appendix C.

Class hierachy of sklearn classifiers

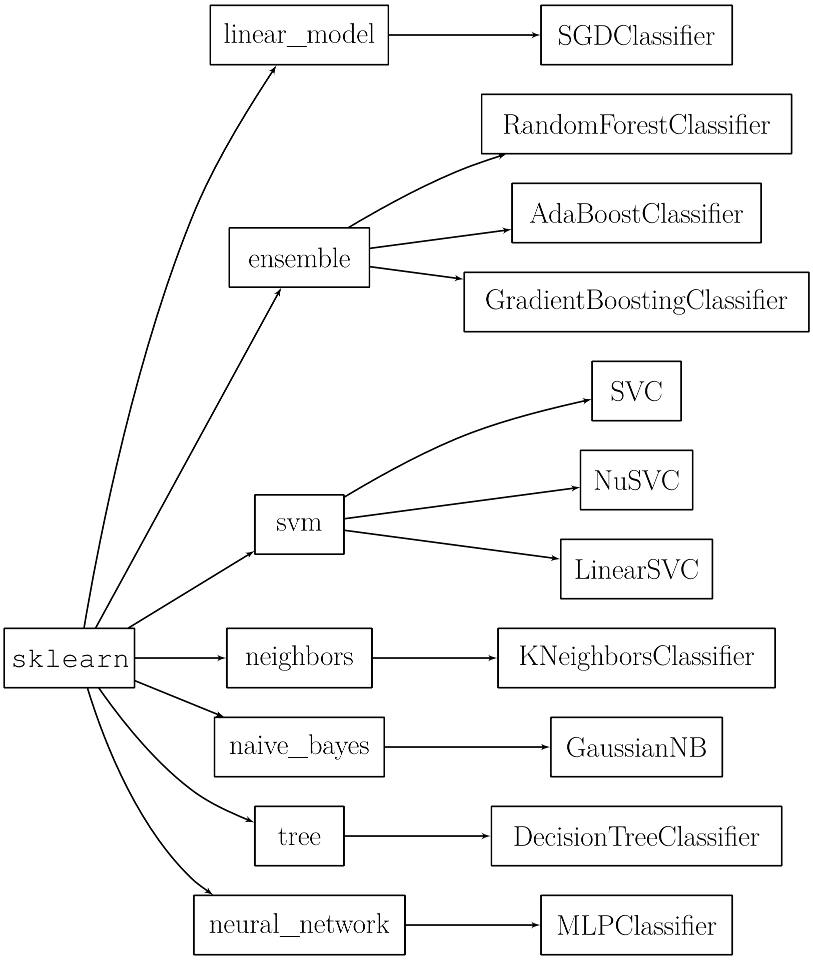


Figure C.1.: Class hierarchy of some Classifiers in `sklearn` [132]. In this hierarchy, the column on the right represents classes, and the two columns on the left are modules, so the first example can be addressed as `sklearn.linear_model.SGDClassifier`.

List of Acronyms

API	Application Programming Interface
AST	Abstract Syntax Tree
ASTNN	Abstract Syntax Tree Neural Network
AVX-512	Advanced Vector Extensions 512-bit
BLAS	Basic Linear Algebra Subprograms
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
ELF	Executable and Linking Format
FLOPS	Floating Point Operations per Second
GPGPU	General-Purpose Computing on Graphics Processing Unit
GPU	Graphics Processing Unit
HPC	High Performance Computing
IR	Intermediate Representation
JIT	Just-in-Time Compilation
LSTM	Long Short-Term Memory Network
MIT	Massachusetts Institute of Technology
ML	Machine Learning
MLP	Multilayer Perceptron
MPI	Message Passing Interface
NB	Naïve Bayes
NN	Neural Network
OpenACC	Open Accelerators
OpenCL	Open Computing Language
OpenMP	Open Multiprocessing
PCIe	Peripheral Component Interconnect Express
POSIX	Portable Operating System Interface
PYPL	PopularitY of Programming Language index
RAM	Random Access Memory
RetDec	Retargetable Decompiler
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
SVM/SVC	Support Vector Machine/Classification
TBCNN	Tree-Based Convolutional Neural Network

List of Figures

List of Listings

List of Tables

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Masterstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Veröffentlichung

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

Hamburg, den 3. August 2021

Ruben Felgenhauer