UPPSALA
UNIVERSITET

UPTEC X 15 011
Examensarbete 30 hp
Juni 2015

# Automating model building in ligand-based predictive drug discovery using the Spark framework

Staffan Arvidsson

UPPSALA
UNIVERSITET

# Degree Project in Bioinformatics

Masters Programme in Molecular Biotechnology Engineering,
Uppsala University School of Engineering

| **UPTEC X 15 011** | **Date of issue 2015-06** |
|---|---|


Author

**Staffan Arvidsson**


Title (English)

## Automating model building in ligand-based predictive drug discovery using the Spark framework

Title (Swedish)


Abstract

Automation of model building enables new predictive models to be generated in a faster, easier and more straightforward way once new data is available to predict on. Automation can also reduce the demand for tedious bookkeeping that is generally needed in manual workflows (e.g. intermediate files needed to be passed between steps in a workflow). The applicability of the Spark framework related to the creation of pipelines for predictive drug discovery was here evaluated and resulted in the implementation of two pipelines that serves as a proof of concept. Spark is considered to provide good means of creating pipelines for pharmaceutical purposes and its high level approach to distributed computing reduces the effort put on the developer compared to a regular HPC implementation.



Keywords

Machine learning, predictive drug discovery, Spark, Big Data, Hadoop, pharmaceutical bioinformatics, pipelining, cloud computing, OpenStack



Supervisors

**Ola Spjuth**
**Uppsala University**

Scientific reviewer

**Åke Edlund**
**KTH Royal Institute of Technology**


| Project name | Sponsors |
|---|---|
| Language<br>**English** | Security |
| **ISSN 1401-2138** | Classification |
| Supplementary bibliographical information | Pages<br>**46** |


**Biology Education Centre** Biomedical Center Husargatan 3, Uppsala
Box 592, S-751 24 Uppsala Tel +46 (0)18 4710000 Fax +46 (0)18 471 4687

# Automating model building in ligand-based predictive drug discovery using the Spark framework

*Staffan Arvidsson*

**Populärvetenskaplig sammanfattning**

Farmaceutisk bioinformatik är ett vetenskapligt område som kombinerar läkemedelskemi och informationsteknik för att underlätta framställning av nya läkemedel. Tillämpningar är exempelvis screening av potentiella läkemedel och hur de beräknas interagera med tilltänkta målproteiner av farmakologisk relevans samt bestämning av fysikaliska och biologiska egenskaper hos nya läkemedel, utan krav på laboratoriestudier. Fördelen är att nya potentiella läkemedel kan studeras *in silico* istället för *in vitro*, vilket både snabbar upp framtagning av nya läkemedel och gör det billigare då färre laborativa studier behöver genomföras.

Ramverket Spark har studerats i detta arbete för att undersöka dess lämplighet för att skapa arbetsflöden, eller så kallade pipelines, för farmaceutiska ändamål. Under arbetet har två pipelines utvecklats; en där prediktiva modeller skapas för att hitta potentiella interaktioner mellan nya läkemedel och proteiner med viktiga biologiska funktioner som kan leda till biverkningar, och en där prediktiva modeller skapas för att bestämma fysikaliska och biologiska egenskaper hos läkemedel. Slutsatsen av arbetet är att Spark är ett lämpligt ramverk som gör skapandet av pipelines både rättfram och flexibelt. Spark är även lämpligt då det innehåller ett bibliotek för skapandet av prediktiva modeller genom användandet av maskininlärning, samt att det har stöd för skapandet av pipelines inbyggt.

**Examensarbete 30 hp**

**Civilingenjörsprogrammet Molekylär bioteknik**
**inriktning Bioinformatik**

**Uppsala universitet, juni 2015**

# Contents

# Abbreviations

| | |
|---|---|
| API | Application Programming Interface |
| CDK | The Chemistry Development Kit |
| DAG | Directed Acyclic Graph |
| HDFS | Hadoop Distributed File System |
| HPC | High Performance Computing |
| IDE | Integrated Development Environment |
| InChi | IUPAC International Chemical Identifier |
| MDL Molfile | File format for molecules including spatial information |
| MVC | Model View Control |
| PTP | Protein Target Prediction |
| QSAR | Quantitative Structure-Activity Relationship |
| RDD | Resilient Distributed Dataset |
| RMSE | Root Mean Squared Error |
| SMILES | Simplified Molecular Input Line Entry Specification |
| SVM | Support vector machine |
| vCPU | virtual CPU |

# 1. Introduction

The goal of this thesis has been to evaluate the applicability of a framework called Spark [1] for the creation of pipelines in predictive drug discovery. Spark is a framework built for Big Data applications [2], with the purpose of facilitating easy development of computer programs that can run on computer clusters and cloud services. The Spark framework enables the user to program in a high level fashion, letting the framework deal with distributing computations and scheduling, such things that ordinary High Performance Computing (HPC) forces the programmer to take care of. Spark is thus a promising framework for creating pipelines that scale to big computational problems and can be used on different computer architectures, without putting too much effort into writing code that runs efficiently in a distributed and parallel environment.

The main pipeline developed in this thesis uses machine learning to create predictive models for drug-protein interactions. These models can then be used for predicting protein interactions for new potential drugs and reduce the need for doing all tests *in vitro* and instead do initial screening *in silico*. The generated models are meant to be used in Bioclipse [3, 4] as a plugin. A second pipeline was also implemented, to show that the implementation strategy is general.

# 2. Background

Pharmaceutical bioinformatics or pharmacoinformatics [5] is a field within bioinformatics where the topic is related to drug discovery and how drugs interacts with other substances in the body. These applications can be greatly facilitated by using computers to perform *in silico* analysis instead of relying on making all tests *in vitro*. Drug discovery encompasses the diverse task of *docking* potential drugs to the intended target protein, model the change of activity in the target protein after interaction with the drug, finding other proteins that the drug might interact with and finding physicochemical properties of the drug (e.g. solubility and toxicity).

My thesis focuses on the last two of these areas, finding potential interactions with other proteins and finding physicochemical properties of new drugs. These problems are extremely important during drug development, if a drug would be prone to interact with other proteins the chance for side effects are large. Being able to predict if or how much a drug would interact with other proteins can help to discard bad candidates at an early stage in the development process. Discarding bad candidates early reduces time and cost as fewer candidates need to be tested later on in development. Predicting physicochemical and biological properties are also important as these factors will determine where the drugs will end up in the human body (e.g. how well they can penetrate cell membranes) and whether the drug might be toxic. These tasks are part of the field known as Quantitative Structure-Activity Relationship (QSAR) [6], which aims to find the link between the structure of drugs and their activity (i.e. in terms of toxicity, binding affinity etc.).

This thesis aims to develop a pipeline, and a general technique, for how to easily perform machine learning on big datasets to derive predictive models. These models are generated by learning from already tested chemical substances and it is desirable to have a setup that can be run fast enough so that new models can be generated as soon as new data is available to use for model prediction.

## 2.1 Descriptor generation

When using machine learning algorithms the input has to be formatted in a mathematical way that the algorithms can use. Typically, chemical substances are stored in formats that can describe the structure of the molecule, in for instance SMILES (Simplified Molecular Input Line Entry Specification), InChI (IUPAC International Chemical Identifier) and MDL molfiles [7]. These formats differ in how much detail they contain, where SMILES only

contain the 1D representation and molfiles attempt to describe the 2D and 3D structure of the molecule. However, none of these formats are suitable to be used directly in machine learning algorithms and they need to be converted into a vector-based format by using descriptors.

Descriptors can be almost anything, for instance you could use number of atoms, total molecular charge, solubility/hydrophobicity, number of a specific chemical element etc. The key is to find good descriptors that can discern between different substances and that they are related to the property that is modelled. Machine learning is highly dependent on what type of descriptors you use, and the results may vary depending on what you use and to what type of data you use it on.

The descriptors used in this thesis have been molecular signatures of different heights. This concept is very simple and illustrated in fig. 1 but has proven to generate stable results in most cases [8]. Signatures are generated in the following way; signatures of height zero means going through the molecule and counting all atoms, see fig. 1b, whereas height one means that signatures are each atom and its closest neighbouring atoms, see fig. 1c. Signatures of a higher order are generated in the same fashion, but looking at atoms further away from the current atom. Combining signatures from several heights, e.g. height one to three, then forms the final descriptors.

`CN=C=O`

**(a)** A molecule in SMILES format

| Signature | Occurrences |
|---|---|
| `C` | 2 |
| `N` | 1 |
| `O` | 1 |

**(b)** Signatures using height 0

| Signature | Occurrences |
|---|---|
| `CN` | 1 |
| `CN=C` | 1 |
| `N=C=O` | 1 |
| `C=O` | 1 |

**(c)** Signatures using height 1

**Figure 1: Example of how signature generation is performed**. Signatures are generated from the molecule in fig. **(a)** by iteratively going through each atom in the molecule and counting the number of occurrences of each signature. A signature is the current atom and its neighbouring atoms at a specific *height*. Using height 0 means only looking at the current atom, height 1 means looking at the neighbouring atoms at maximum 1 chemical bound away.

Once the signatures are generated, they have to be transformed into vector form by associating a unique number(ID) to each unique signature. Each molecules signature can thus be represented by vectors, which can be used in the subsequent machine learning. The signature to ID mapping must be kept so that new molecules will have the same mapping when generating signatures for these molecules.

## 2.2 Machine learning

Machine learning is a big field and this report will not go into any greater depth of the subject. This section will only give a brief overview of the most necessary concepts. The important aspect is that by using the molecular descriptors of molecules with known physical or biological properties, it is possible to build *models* that can *predict* the same properties for new molecules. There are several papers proving that machine learning algorithms are applicable for the field of pharmaceutical bioinformatics and drug discovery, see for instance [9], [10] and [11].

Machine learning can be divided into several systematic fields, each trying to solve one type of problem. In *Classification* problems, each record (in this case molecule) can be part of one or several *classes*. The classes are discrete, for instance a molecule can be binding or non-binding, toxic or non-toxic. In *Regression* problems, each record has a continuous value of the quantity that is modelled. This value could be any type of physical or biological property, for instance the binding affinity of a drug to the target protein or the degree of solubility of a molecule in a solution.

The mathematical formulation in both classification and regression is that each record can be described by a vector of *descriptors*, $\vec{X}$, and the *outcome*, $Y$. Each sample can thus be described by the pair $(\vec{X}, Y)$. The outcome, $Y$, is the class or classes that the sample is part of in a classification problem and the value of the modelled property in a regression problem.

Solving classification and regression problems can be done by several algorithms, and picking the best algorithm for your problem can be hard. Decision trees, Neural Networks, Linear and Logistic regression and Support vector machines (SVMs) are examples of popular machine learning algorithms for classification and regression problems. This thesis has tested two types of linear regression algorithms, those which were already implemented in the Spark machine learning library, but other algorithms could also be used once implemented. The focus has been to evaluate the framework and not looking at the end result.

Evaluating and validating the models can be done by for instance *cross-validation* [12]. In $k$-fold cross-validation, the full dataset (of records with known outcome) is split up into $k$ non-overlapping partitions and $k$ models will be created, each leaving out one of the folds. Each of the $k$ models will be evaluated by using the left out fold, which will give an estimate on the predictive power of the created model. There are more variations of cross-validation, but the key is to use known records to get an estimate of how accurate the model is.

## 2.3 Pipeline

The main goal of this thesis is to create a Protein Target Prediction (PTP) pipeline that can create predictive models on drug-protein interactions for a large number of proteins. This application uses existing records of drug-protein interactions as discussed in previous sections. The pipeline should be flexible in how to make changes in order to apply new machine learning algorithms, change descriptors and other parameters. One other important factor was that preferably only one framework should be used, instead of mixing for instance a large set of smaller programs with a high level tool for handling the dataflow between the smaller programs. Instead, this pipeline should be easy to "plug-and-play" with different components.

The overall structure of the pipeline is outlined in further detail in section 5.3, but should contain *data loading*, *filtration*, *descriptor generation*, *model generation & evaluation* and a final *report & publish* step. The complexity of the pipeline increases as multiple datasets are treated at the same time, but each dataset is independent of the others. As each dataset is independent of the others, it is also possible to execute them concurrently and in parallel if there is enough hardware resources. The reason for treating several datasets simultaneously is that several thousands of datasets are of interest and it is far less time-consuming to only run everything once instead of having to run them one by one.

Apart from the main pipeline that predicts protein-drug interactions, another relevant application for running as a pipeline came up during the progress of the thesis work. This new application can be described as a parameter selection pipeline. The pipeline would run several times, changing the number of records used when generating the models, and evaluate the accuracy of the models (measured as e.g. RMSE). Both runtime and model accuracy can thus be plotted against dataset size, making it possible to find a suitable size for optimising accuracy but not wasting computer resources by using too much data. The pipeline could then be run multiple times, using different parameters in model generation and vary other parameters as well.

# 3. Spark Framework

## 3.1 Spark Basics

Spark [1] is an open source project under the Apache Software Foundation. The Spark framework is written in the programming language Scala and has APIs (Application Programming Interface) for Java, Scala and Python. This thesis has used the version 1.3.1 of Spark, using the Scala API to make full use of the functional aspects that Spark give the user as well as skip some of the boilerplate code needed for the Java API. The Scala interface also made it possible to use a library written in Java that is used for handling parsing molecules, see Section 4.2.

This section will be a short introduction to the concepts of Spark, for a more complete walkthrough of this framework the reader is referred to the Spark homepage [13] and the Spark book written by Karau and Zaharia [14].

### 3.1.1 Spark application overview

Spark can be run in both an interactive mode (in a computer shell) or run as a stand-alone application. In interactive mode, the programmer can issue commands one by one, getting responses after each command. This can be handy for testing purposes and for simpler applications, but building pipelines are done in a stand-alone mode. Spark programs written as stand-alone applications consists of a *driver program* which runs the `main` function and issues the distributed operations on the computer cluster. The application also contains a number of *executors* that are running on the other machines on the computer cluster, each storing part of the distributed data and available to perform operations on that data once a distributed operation is issued from the driver. The physical distribution of driver program and executors depend on how the programmer runs the application on the cluster, where both driver and executors have definable number of cores and main memory. The Spark framework will then handle distribution of the driver and executor processes on the cluster.

### 3.1.2 Memory model - RDDs

The main abstraction used in Spark is the *Resilient Distributed Dataset (RDD)* [15]. This data structure provides an interface towards data that is distributed over many machines physically, letting the programmer only having to deal with one object and not be concerned with where data is lo-

cated. RDDs can contain any type of Scala, Java or Python objects and even sets of objects. An important feature of RDDs is that they are *immutable*, meaning that an RDD cannot be changed once it has been created. Forcing RDDs to be immutable simplify things for the framework, not having to deal with updates, synchronisation and issues tied to this. RDDs support two types of operations; *transformations* and *actions*.

Transformations are operations that act on one or more existing RDDs and produces a new RDD. These operations are *coarse-grained*, affecting the entire data structure instead of individual records. Transformations are *lazily evaluated* and will not be executed right away; instead the framework will build *lineages* of transformations and only execute the lineages once the program encounters an action. The basic transformation operations includes the higher order functions `map` and `filter`, that either maps a function to each element in the RDD or filters the RDD based on a function. There are also transformations that combine RDDs, for instance `union` and `join`.

Actions are operations that compute non-RDD results, such as storing an RDD to file, counting the number of records in an RDD or computing the sum off all values in an RDD. Once actions are encountered, Spark will check the lineage and dependencies to optimize the computations. The optimisation is done by grouping transformations together, and thus only pass over the data once, Spark will also try to omit transformations that the result do not depend on.

RDDs also support some other important features such as *persisting*, which means that an RDD can be stored in main memory (or on disc if there is not enough space in main memory) once it has been computed. The default principle is that once an action is encountered and the lineage has been used to compute the result, the complete lineage will be re-computed next time an action is called even though some parts might include the same RDDs. Persisting is very favorable in applications that have iterative tasks for instance, or simply where the same RDDs are used multiple times. Spark also includes fault-tolerance, enabling a long run to recover even if machines crashes during a run. This is done both by using checkpointing and by using the lineages, only data that is lost and is needed in succeeding steps in the program will be recomputed.

## 3.2 Spark input flexibility

Spark is very flexible in what kind of file format that it reads, accepting all Hadoop-supported file formats (e.g. files in the normal file system and HDFS [16]). Switching between different formats is as easy as adding the

prefix "file://" to files in the normal file system and omitting it when using HDFS-files. This is another reason to why testing code can be done on a local laptop without the need to install huge HDFS-files and the same code can later be used on a cluster, once using really big data.

## 3.3 Spark compared to High Performance Computing

Traditional High Performance Computing (HPC) has long been the solution for big computational problems and is one of the potential competitors for Spark. HPC programs usually considers computation and data placements jointly, moving data from storage during computation or as a separate task [17, 18]. In a distributed computing setting and in applications that uses large amounts of data, this strategy runs into problems such as congested networks because of all data needed to be sent between computer nodes, leading to deteriorating performance. This has lead to the new paradigm of *data-aware scheduling* in data intensive applications [17]. Spark and other data-aware scheduling frameworks take another approach to data, and instead schedules computations to were data is located. This approach is often very successful, especially when the amount of data is large.

Another difference between HPC and Spark is that HPC often requires running on specialised HPC-clusters. These HCP clusters are usually referred to as supercomputers which uses complex interconnections to facilitate fast communication between cluster nodes. Spark on the other hand runs on commodity hardware [1], which is cheaper and can be rented by cloud services from e.g. Amazon [19].

Spark is however not replacing HPC in compute-intensive applications were bandwidth is not the limiting factor. HPC is usually written in languages such as C, C++ or Fortran to achieve high performance, and outperforms Spark in compute-intensive applications. For applications that both crunches lots of data and are compute-intense, the mix of HPC and data-aware scheduling frameworks would be to prefer, studied for instance by Luckow et al. [20].

## 3.4 Spark Machine Learning

Spark comes with a library for machine learning, called MLlib [21] that contains implementations of some commonly used machine learning algorithms. This library builds a basis for machine learning applications and is continually

improved and extended with new functionality (new algorithms and optimisation methods). One of the reasons for choosing Spark for this project was that there already existed the MLlib, making it easy to assess the applicability of Spark when constructing pharmaceutical pipelines. As Spark is an open source software, it is also possible for others to implement new functionality and publish their results, such as Lin et al. [22] when their group implemented logistic regression and linear SVMs in Spark. It is also possible that the research group that I work for now implements their own algorithms that are more suitable for their applications.

It is worth mentioning that there has been no direct assessment of how well the machine learning works in Spark as this work only focuses on how to build pipelines using Spark. Optimising the pipeline with the correct algorithms and optimisation methods will be done outside the scope of this thesis.

### 3.4.1 Spark ML Pipelines

Spark Pipelines API [23] is a promising feature which enables a user to build and set up their own machine learning pipelines in an easy way. This feature came in the 1.2 version of Spark and it is currently (in version 1.3.1) only an alpha component, meaning that it might encounter some big changes before the API becomes stable. This feature is tightly linked to the Spark MLlib and enables the user to think of pipelines at a higher abstraction level. Currently, the ML pipeline API forces the user to use DataFrames, which is the datatype that Spark SQL uses. DataFrames are similar to RDDs in being lazily evaluated, distributed and fault-tolerant, but differs in what type of data they can store. Where RDDs can store any type of user-defined classes, DataFrames can like SQL databases only store structured data with pre-defined types (e.g. numerical values, strings and more complex types).

The data used in pharmaceutical pipelines are not as structured as typical databases so the lockdown to DataFrames did not seem promising. Another disadvantage is that each step in an ML pipeline needs to pass a DataFrame, which some of the steps in the PTP pipeline should not do (e.g. when creating HTML-files). It would probably be a working approach to use ML pipelines as a part of the complete pipeline, but the time restraint of the project did not allow for this approach to be further investigated. However, the ML pipelines do support DAG (Directed Acyclic Graph) pipelines which is a desired feature and also model selection via cross-validation [23]. Furthermore, it is usually better to use techniques and methods that others use, as these methods get better tested. If more time was available, ML pipelines might be worth evaluating in greater detail.

## 3.5 Spark Drawbacks

Spark has many great features that make it a good framework for implementing the pipelines in this thesis work. However, there are a few features that can be described as negative. First of all is the restriction of only allowing for non-nested Spark-operations, meaning that multiple transformations or actions cannot be used within each other. In some cases it is possible to refactor the code to go around this shortcoming, but in other cases it is not that simple.

Here follows one case where refactoring is possible. Consider that you have one RDD containing your data of interest, another RDD containing a lookup-table and that you wish to join the information in these. One could be tempted to `map` over the data-RDD and for each record do a lookup in the lookup-RDD. This produces nested operations and is thus not supported. Instead, refactoring has to be done by either collecting the lookup-RDD into a non-RDD datastructure (which can only be done if the lookup-RDD is small enough), or do some transformations to both of the RDDs so that a `join` can be performed between them.

Another problem, which cannot be refactored, is the following. Consider that you have a fully functional Spark workflow that uses Spark operations onto some dataset. Then consider that you wish to reuse this workflow several times, for instance with different input parameters or different datasets. Then it would be preferable to do a `map` over the parameters or datasets and apply the workflow on each setup. This could thus, in theory, allow Spark to load-balance several dataset-runs on the available data resources. However, this is not allowed as both the 'upper level' `map` and the 'lower level' workflow uses Spark operations, leading to nested operations. This issue is further discussed and, to some extent, solved in Section 5.3.

Another disadvantage is that Spark is optimised for creating lazy transformations and only a few actions leading to actual work done on the data. However, if the pipeline needs to collect statistics during a run or decide how the pipeline should act in a branching situation, then an action is required for each time some information is needed. This means that a lot of actions are spawned and slows down the run time significantly compared to the work done for machine learning and other 'useful' operations (see Section 6.2.1). However, the Spark developers have a library with asynchronous RDD actions [24] which allows the user to execute non-blocking actions, without having to wait for the result before continuing execution of the program.

# 4. Materials and Methods

## 4.1 Spark & Hadoop

Several versions of both Spark [13] and Hadoop [25] have been used during the work in this thesis, using newer versions as they are launched as stable versions by the Spark community. The versions used for creating the final results seen in the Results section were done with Spark version 1.3.1 built for Scala 2.11 and using Hadoop version 2.5.2.

## 4.2 Chemistry Development Kit

The Chemistry Development Kit (CDK) [26] is an open source toolkit implemented in Java that has functionality for handling diverse applications within chemical manipulations on molecules. This toolkit can read and write chemical data in different file formats and perform manipulations on the loaded molecules, among other functionality. Both Scala and Java runs in the Java Virtual Machine and Scala provides seamless integration of Java code, making CDK easy to integrate in the implementation. CDK version 1.5.10 was used for handling the parsing of molecules in SMILES format and creating descriptors for succeeding ML steps.

## 4.3 Openstack cluster

Uppsala University's resource for high performance computing, UPPMAX [27], was used for testing and doing full scale runs. UPPMAX has a new cloud computer cluster that uses Openstack [28] for virtualisation and handling of computer resources. The computer resources were set up with an image of Ubuntu Precise Pangolin, using m1.xlarge as 'Flavor', giving each machine 8 virtual CPU-cores (vCPUs) and 16 GB of virtual memory.

## 4.4 ChEMBL

Data used for testing was obtained from the open database ChEMBL [29, 30] which is a curated database for bioactive drug-like substances. The actual data and results were not important for the project, as it is mainly focused on building the pipeline and evaluating Spark. Data was collected by using

parameters used in previous work done by the research group, collecting data that should be similar to what is later to be crunched in the pipeline.

## 4.5 FreeMarker

The open source template engine FreeMarker [31] was used for creating HTML-pages in the ending stages of the pipeline (used for reporting statistics about the run). This software attempts to use the MVC pattern (Model View Control), to separate the data model from the visual design. FreeMarker is a Java library that is fairy lightweight and facilitates fast generation of HTML-pages and served as a proof of concept for how to create the reports within the Spark framework.

## 4.6 Programming environment

Programming was done in the OSx operating system, using Eclipse as IDE. This setup was used to mimic the behaviour that will be used by the research group later on. Eclipse has built in support for using Apache Maven [32] for project management and build automation, in this way making it easy to deploy a new pipeline version to the cluster with only a few commands. Maven also takes care of dependencies to other libraries and downloads all extra code you need.

# 5. Implementation

This chapter of the report will outline how the pipelines were implemented using Scala and Spark. To illustrate the flow of data in the two pipelines, flowcharts have been used. Readers not familiar with these types of illustrations are referred to Appendix A for a cheat sheet of the symbols used in these flowcharts.

## 5.1 Pipeline API

The pipelines created in this thesis builds on the idea of pipelines given in the exercises of AMPCamp 5 [33] given at the AMPLab [34] faculty at Berkeley University. A pipeline will in this report be defined as a workflow composed of nodes. A node can be any functional unit and can be connected to another node only if their corresponding input and output types match each other, see fig. 2. Pipelines themselves can be reused by connecting them to other nodes or pipelines, making it possible to assemble big pipelines out of smaller sub-pipelines.

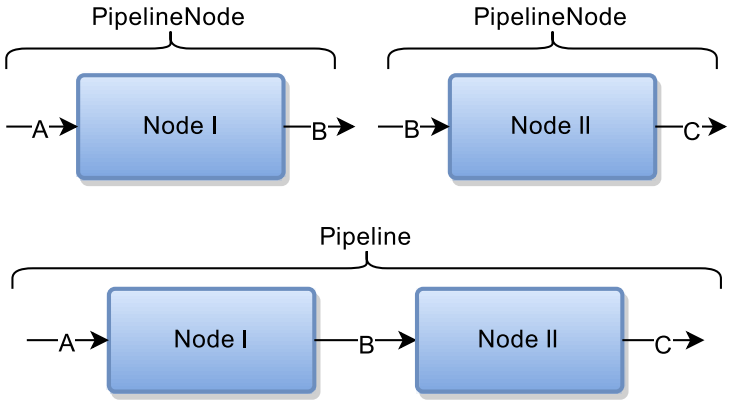


**Figure 2: The basic structure of a pipeline and its components.** A pipeline is composed of *PipelineNodes*, where nodes can be chained or linked together if the output type of the first node matches the input type of the second node. The pipeline composed by node I and II in the figure can now be used as a node itself, having input type A and output type C.

The implementation of the Pipeline API was based on the type-safe functional programming that Scala supports [33]. There are three basic things that is needed for defining pipelines using this method. First the PipelineAPI

needs to be added to your project, seen in fig. 3. This 'API' defines the basic type that is needed, `PipelineNode`, which is the only type that is needed when composing linear (non-branching) pipelines and is taken from the AMPCamp exercises [33]. `PipelineSplit` and `PipelineBranch` are types that I have defined to be able to incorporate branching pipelines as well.

```
1 object PipelineAPI {
2   type PipelineNode[Input, Output] = (Input => Output)
3   type PipelineSplit[T,S] = (Either[T,S] => Any)
4   type PipelineBranch[Input] = (Input => Any)
5 }
```

**Figure 3: Code for defining the Pipeline API.** The most important function is the `PipelineNode`, which can be used by itself to create linear pipelines. The `PipelineNode` is taking two *type parameters*, specifying the input and output types of a function. `(Input => Output)` defines a function that takes one parameter of type `Input` and will return a result of type `Output`. `PipelineSplit` and `PipelineBranch` are types that help when creating branching pipelines, where `PipelineSplit` can act as a decision for how to pass data (depending on the data). `PipelineBranch` can add branching by sending the same data to multiple processes, which all can execute indenpendent of each other (depending on how it is implemented).

Once the pipeline API is defined, the second thing that is needed is to implement the `PipelineNode` type in all of the nodes that you wish to take part of the pipeline. This can be done in Scala by either creating an Object or a Class and extending `PipelineNode`, see fig. 4. The only thing that is needed except extending `PipelineNode` on line 1-2 is to override the `apply`-function (line 4). This `apply`-function is the corresponding function in the PipelineAPI, taking a parameter of type `Input` and returning a result of type `Output`. fig. 4 illustrates a minimal working example and the code within the `apply`-function is just as any other function, it can call other functions, create objects etc.

The third and final step of creating a pipeline is outlined in fig. 5. A pipeline can either be created by itself or can be composed of several nodes, each implementing the `PipelineNode` type. Subsequent nodes are chained together using the Scala function `andThen`, which simply forwards the output of the first function as input to the following function. The object, `pipeline`, on line 5 in fig. 5 is a *function* having the same input type as of `DataLoader` and the same output type as of `Report`.

```
class DataLoader (@transient val sc: SparkContext)
   extends PipelineNode[String, RDD[String]] with Serializable {
   // Override the apply function
   override def apply(filePath: String): RDD[String]= {
      sc.textFile(filePath)
   }
}
```

**Figure 4: Implementing a `PipelineNode`.** To implement a `PipelineNode` you need to *extend* the `PipelineNode` type and specify the type parameters. This is a minimal example that takes a path to a file and reads the file into an RDD and returns it. The SparkContext sc is needed for reading files and is markt as `@transient` because it must not be serialised during execution.

```
1 // Using a single PipelineNode
2 val dataloader = new DataLoader(sc)
3 val data = dataloader("<some file path>") // Uses the apply
      function
4 // Combining several nodes
5 val pipeline = new DataLoader(sc) andThen
6    new ExtractData andThen
7    new Model andThen
8    new Report
9 pipeline("<some file path>") // Executes all steps in pipeline
```

**Figure 5: How to use PipelineNodes and combine them into pipelines.** `PipelineNodes` can both be used one by one, by creating an object of the class and then calling the apply function as on line 3. Several `PipelineNodes` can be combined by using the scala function `andThen`, which chains the result of each apply function as input to the next (line 5-8). The combined nodes can then be executed in the same way as a single node.

These three steps are enough for setting up complete *linear* pipelines and the type system will make sure that the chaining of subsequent nodes are correct (nodes that do not match will not compile). If the pipeline needs *branching*, the `PipelineSplit` in the API will have to be used. This API only needs to be implemented once and it is enough to create a new instance for each split needed. Each splitting instance can be used as a node itself and be added with the `andThen` function. An implementation and further details can be found in Appendix B.

## 5.2 Parameter Selection pipeline

The Parameter Selection pipeline was the second pipeline implemented during the work of this thesis, but it is somewhat simpler which makes it an easier introduction. The overall structure of the pipeline can be seen in the flowchart in fig. 6. This pipeline will create models and evaluate them for a set of different training data sizes, thus evaluating the influence of amount of training data on both the predicted error of the generated models and how runtime will be influenced (thus looking at scaling of the algorithm).

Looking at the flowchart in fig. 6, the first step is to split the complete dataset into a *training dataset* and a *testing dataset*. This step makes sure that testing will always be performed on the same records to easier make a distinction between models. Once the correct parameters have been found using this pipeline, cross-validation (Section 2.2) can be used for getting a better estimate on the predictive quality of the final model.

For each training dataset, the pipeline will begin by generating the signatures of each molecule in the training dataset and give a mapping of signatures to unique ID. The signature to unique ID mapping can then be used in the generation of signatures for the testing data set (as these datasets must share this mapping). Once both training and testing data is in a vector-form that the machine learning algorithm uses, the model and evaluation can be performed.

## 5.3 Protein Target Prediction pipeline

### 5.3.1 The structure

The *Protein Target Prediction* (PTP) pipeline, introduced in Section 2.3, has one added level of complexity compared to the parameter selection pipeline. This pipeline should take a file with multiple datasets, each dataset corresponding to drug-protein inhibitory data from a unique protein. Each dataset is thus independent from every other dataset and could be executed independently of each other. The complete pipeline can be seen in fig. 7, where each dataset $D$ runs a sub-pipeline indicated by the curly brackets.

The PTP pipeline also includes steps for *report* and *publish*, where report should generate an HTML-page with statistics about each dataset and the corresponding model (if there is one) and the publish step should pack the model so that it can be imported to Bioclipse [3, 4]. The reporting step is described in Section 5.3.4 and the publish step will not be covered in this report.

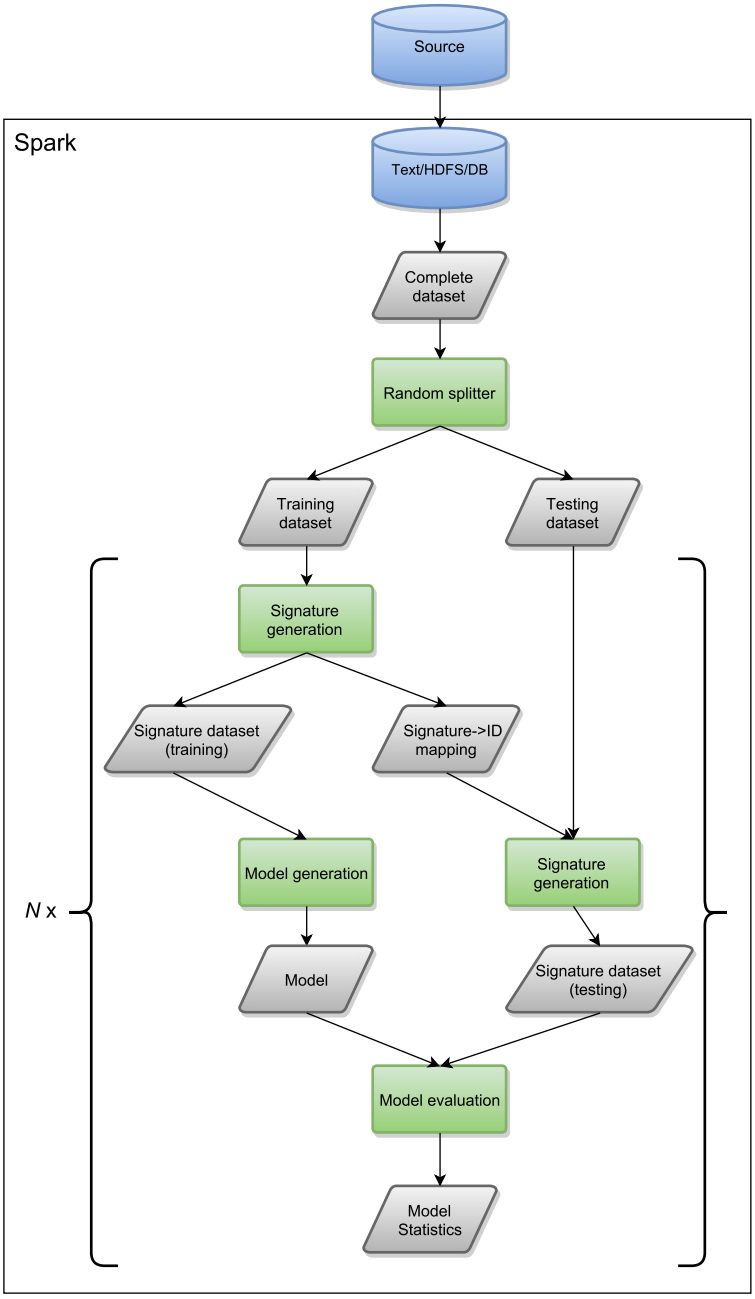


**Figure 6: A flowchart of the Parameter Selection pipeline.** The flow in the Parameter Selection pipeline follows this figure. Once the data is loaded into Spark the complete dataset will be divided into a *training dataset* and a *test dataset*, these partitions will be kept during the whole pipeline to always use the same test data for evaluation. The pipeline will then perform the same *sub-pipeline* (indicated by the curly brackets) several times, each with increasing size of the training dataset. The sub-pipeline performs signature generation (construction of descriptors), model prediction and finally model evaluation.

Another complexity compared to the Parameter Selection pipeline is that the pipeline has a decision event where the pipeline should direct the flow depending on what the data looks like at that particular point in the pipeline, see the red rectangle in fig. 7.

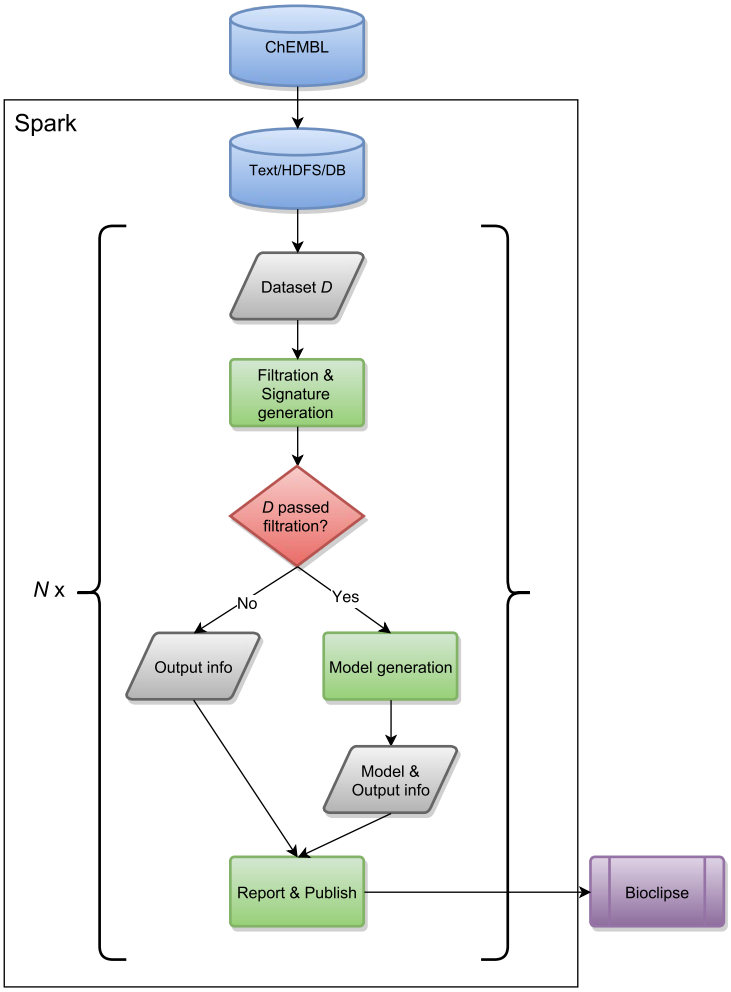


**Figure 7: A flowchart of the PTP pipeline.** This is an overview of the complete PTP pipeline and what stages it includes, see fig. 8 for a closer look at the stages that each Dataset $D$ run through (sub-pipeline within the curly brackets). The basic steps are that each Dataset $D$ is first filtrated based on user defined preferences (molecule size, inhibitory value etc.) and signatures are generated. If $D$ is still large enough to produce a good enough model it will be passed on to model generation, model evaluation and final report/publish, otherwise the pipeline will just generate a report on the filtration step.

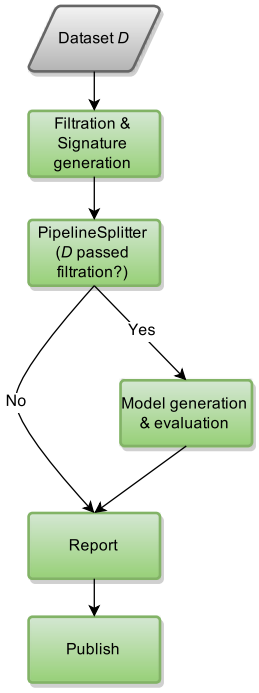


**Figure 8: A flowchart of the sub-pipeline in the PTP pipeline.** This flowchart shows the sub-pipeline within the complete PTP pipeline (the curly brackets in fig. 7). Here the decision point in the previous flowchart has been changed into a process that will direct the flow in the pipeline. Datasets that have too few records to be likely to generate models of high quality do not pass the *model generation & evaluation* step and will go directly to the *Report & Publish* steps.

### 5.3.2 Implementation

The implementation of the PTP pipeline is rather straightforward and builds on the `PipelineAPI` discussed in the beginning of this chapter. Once each process in fig. 7 and 8 are implementing the `PipelineNode` type, they can be composed into the complete pipeline, see fig. 9. Take specially note of the high level of abstraction that can be achieved in this way, as the resemblance between the flowchart and implementation is very similar. The object, `pipeline`, that is created on line 12 corresponds the sub-pipeline in fig. 8 and it can be used for each of the datasets constructed on line 3.

```
// Load file and extract all datasets (one each for each protein)
val loader = new DataLoader(sc) andThen new ExtractTargetDatasets
val datasets = loader(filePath)
// Define the branches
val left = new ReportFailure(outputDirectory) andThen// Left branch
   new Publish()
val right = new SignatureToLabeledPoint(sc) andThen // Right branch
   new MakeMLModel(sc) andThen
   new ReportSuccess(output_directory) andThen
   new Publish()
// Complete pipeline
val pipeline = new FilterGenSignatures(sc) andThen
   new PipelineSplitter(left, right)
```

**Figure 9: Code for defining the PTP pipeline.** The pipeline can be defined in a high level approach, note the similarity to the flowchart in fig. 8. The constructor of `PipelineSplitter` takes two parameters (left and right) which is the two branches that can be executed, depending on the output from FilterGenSignatures. FilterGenSignatures gives either an *OutputObject* that contains information about the filtration or it gives an (RDD, *OutputObject*) tuple that will be sent for further execution in the pipeline.

### 5.3.3 Improving performance - concurrent programming

The key thing in the PTP pipeline is that several datasets are present, each being independent of all other datasets. This means that potentially the sub-pipeline which performs signature generation, modelling and report/publish can be performed in parallel as long as the machines has spare computer resources to do so. This lead to the creation of a higher order function (that is, a function that takes another function as input) which can take any user-defined pipeline and execute the pipeline with an array of RDDs as input. Each dataset can then run in a separate thread, which is allowed by Spark, and can be scheduled on the cluster to fully utilize all computer resources that are available. The improvement in runtime is shown in Section 6.2.1.

The higher order function can be called as `asyncPipeline(pipeline, datasets)` where `pipeline` and `datasets` are the objects defined in fig. 9. `asyncPipeline` is simply a higher order function with the signature `def asyncPipeline[T,S: ClassTag](pipeline: (T=>S), input: Array[T]): Array[S]`, meaning that the type system will force `pipeline` to have the same input parameter as the array `input` has and that the function will

return an array of the type which `pipeline` has as return type. Under the hood this function uses Scala Futures [35] and corresponding Await to spawn a new thread for each dataset.

### 5.3.4 Reporting statistics as HTML

The output of the PTP pipeline contains both the generated model and statistics for that model. Statistics should be in HTML-format so that it can be used in the Bioclipse framework, thus adding the requirement of being able to write HTML-files within the pipeline on a computer cluster. The Freemarker [31] template engine was used and found to make creation of HTML-pages fairly straightforward. To write files on the cluster, the application must be executed in *yarn-client* mode instead of the normal *yarn-cluster* mode. This changes how the cluster executes the application, either by considering it to be executed externally (client mode) or internally on the cluster (cluster mode).

Examples of how the output HTML-files could look like can be seen in fig. 10 and 11. These only serves as proof of concept and the final HTML-files should look a bit nicer and could contain other information.

# Activin receptor type-1 : Failed dataset

This dataset was unfortunately skipped due to not fulfilling the filtration requirements. See further information below.

## Data Info

Initial Dataset Size: 31
Dataset Size after filtration: 30

## Filtration options

Filtration has been performed with the following options:

| Option | Value |
|---|---|
| minSignatureHeight | 1 |
| maxSignatureHeight | 3 |
| minNumRecords | 50 |
| minHeavyAtoms | 5 |
| maxHeavyAtoms | 50 |
| ic50_threshold | 10,000 |

**Figure 10: An example HTML-page for a failed dataset.** If a dataset is considered to small for generating models on, the pipeline will skip model generation and only write out statistics about the filtration step. This is an example of how such a page could look like and is how it is currently managed in this implementation. The final HTML-report is likely to look different as this just gives a proof of concept for the ability to write HTML within Spark.

# 3-phosphoinositide dependent protein kinase-1

## Model information

The model has the Mean Square Error (MSE) of: 2,705,698.213. The model used was Mock_linearRegression, using the following options:

| Option | Value |
|---|---|
| numIterations | 500 |
| model_name | LinearRegressionWithSGD |
| stepSize | 0.001 |

## Data Info

Initial Dataset Size: 343
Dataset Size after filtration: 269

## Filtration options

Filtration has been performed with the following options:

| Option | Value |
|---|---|
| minSignatureHeight | 1 |
| maxSignatureHeight | 3 |
| minNumRecords | 50 |
| minHeavyAtoms | 5 |
| maxHeavyAtoms | 50 |
| ic50_threshold | 10,000 |

**Figure 11: An example HTML-page for a successful dataset.** This is how the output in HTML can look like for a dataset that goes through the complete pipeline. The page contains model information (parameters, algorithm and RMSE), data information (number of records used in model) and filtration parameters used. The final HTML-report is likely to look different as it just give a proof of concept for the ability to write HTML within Spark.

# 6. Results

## 6.1 Parameter Selection pipeline

The Parameter Selection pipeline outputs the execution time and the root mean squared error (RMSE) of the model when applied to an external test set as a function of training set size, see fig. 12. This pipeline could also be used with other machine learning algorithms or new parameter values and evaluate these against each other. The Parameter Selection pipeline is in this case used to predict on *Log D* values for different drug-like substances, which is a measure of lipophilicity (how well the drug can penetrate the lipid bilayers in cell walls), but the pipeline can be used for evaluating models on a variety of biological or physical properties. The total dataset size can in these applications be in the order of a few hundreds of thousands and it is also possible to pick a dataset size, which takes both RMSE and runtime into consideration.

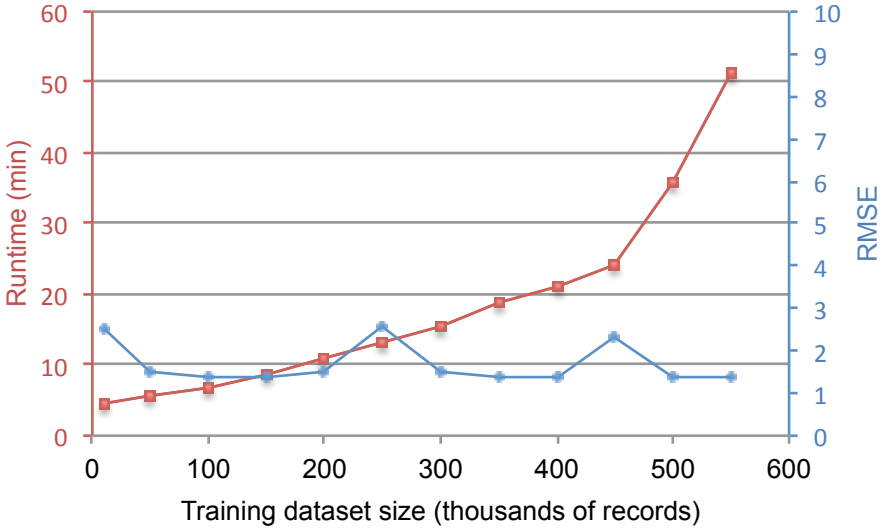


**Figure 12: Output from the Parameter Selection pipeline.** This is an example of how the output from the Parameter Selection pipeline could look like. The RMSE (Root Mean Squared Error) is behaving strangely because of the parameters in the model optimisation have not been tuned yet, which can be done later with for instance a grid search. This run was performed with 9 worker nodes, each with 7 vCPUs and 14 GB virtual memory.

## 6.2 Protein Target Prediction pipeline

The PTP pipeline uses considerably smaller datasets compared to the Parameter Selection pipeline. Two different datasets have been tried, either the complete dataset composed of 1940 proteins with in total 484 thousand records of drug-protein interactions or a subset using only 263 proteins and 80 thousand records. Currently only the small dataset has been able to run successfully, resulting in the speedup graph in fig. 13 for the asynchronous version of the pipeline. The speedup graph shows a linear speedup going from one to five nodes, but then the speedup decreases. This is probably due to the dataset being too small to benefit of the added nodes.

There has been issues when running the bigger dataset on the computer cluster, which are not yet solved. This might be due to a severe crash of the cluster a few weeks earlier, which lead to some changes in the cluster setup.

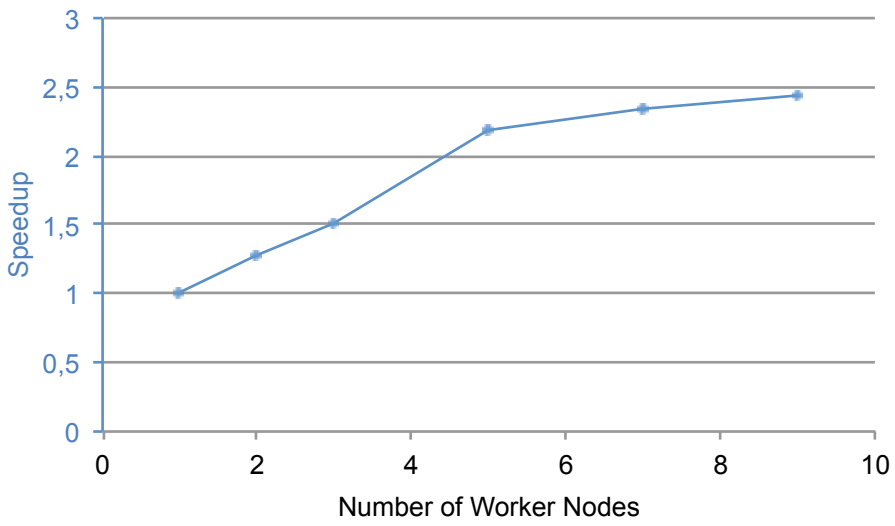


**Figure 13: Speedup for the PTP pipeline using a small dataset and `asyncPipeline`.** The small dataset contains 263 protein datasets and in total more than 80 thousand records. The speedup is good initially when adding new worker nodes but levels off quickly, probably due to the dataset being too small to gaining more by using more machines. This speedup is generated by using the final PTP pipeline, which uses the `asyncPipeline` discussed in Section 6.2.1. Each worker node has 7 vCPUs and 14 GB virtual memory.

### 6.2.1 Concurrent execution to improve performance

As mentioned earlier, the PTP pipeline runs several protein datasets at the same time, while each dataset is independent from the others. Evaluating the gain in runtime while using the `asyncPipeline` function which can run datasets concurrently compared to the naive approach of running each dataset one by one gave a respectable speedup of more than 5 times when using a setup with 9 computer nodes, each having 7 vCPUs and 14 GB virtual memory, see fig. 14. This speedup is probably due to that each individual dataset is fairly small, and it was possible to *repartition* each dataset to only be stored on one computer node. Thus it is possible for the driver program to run several operations concurrently as they will operate on different nodes, keeping in mind that the driver schedules operations based on data locality.

The difference in using the normal, blocking, RDD actions and their non-blocking counterparts in the asyncRDD library [24] gave a 12% increase in runtime for blocking over non-blocking, fig. 14. This was found when swapping the blocking `count` to the non-blocking `countAsync`, when collecting statistics about the initial dataset size. Adding one extra action thus lead to a 12% increase in runtime, making it important to use non-blocking calls whenever possible, and try to restrict the number of actions in your code.

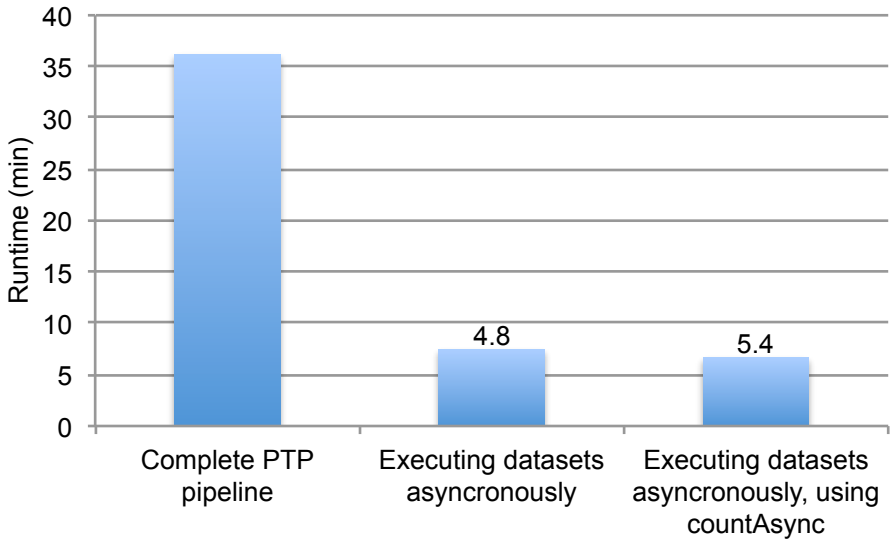


**Figure 14: Comparison between serial and asynchronous code.** These runs were performed using the small dataset in the PTP pipeline and 9 worker nodes, each having 7 vCPU cores and 14 GB virtual memory. The leftmost bar shows the runtime of the final pipeline when executing all datasets one by one in a synchronous fashion. The middle bar shows the runtime when executing all datasets asynchronously with the `asyncPipeline` function. The rightmost bar shows the final pipeline, which uses the non-blocking `countAsync` function for collecting statistics. The label above the two rightmost bars are their corresponding speedup compared to the leftmost bar. Speedup is defined as the quotient $Runtime_{old}/Runtime_{new}$.

# 7. Discussion

## 7.1 Modularity

The pipelining approach used in this thesis made it easy to plug and play with different components without making further changes. If e.g. the input is formatted in a new way or perhaps in the form of a database instead of text-file, it is enough to write a new parser or make changes to the current one, and nothing else needs to be changed in the pipeline. This is important as new descriptors, machine learning algorithms and other factors should be easy to switch between, making it easy for the end user. The `PipelineSplitter` class also provides an easy view of how a pipeline should branch of, depending on different events and make it easy to programmatically define the complete pipeline in a concise way. It is also beneficial to be able to reuse pipelines in bigger pipelines so that sub-pipelines can be defined separately, making it easier to change smaller parts of the pipeline.

## 7.2 Alternative frameworks

Spark is indeed a promising framework for this kind of applications, especially as they put time into developing the machine learning pipeline library [23]. However, there are many up and coming projects with similar goals of Spark, such as the Flink [36] and the Crunch [37] frameworks. These too are open source projects, part of the Apache Software Foundation.

Flink is a framework that has very much in common with Spark as both uses in memory computation and sits on top of Hadoop. The difference between the two is that Flink was built for accommodating cyclic workflows efficiently and that the internal structure differs. Flink was shown to be faster in the iterative process of finding k-means, in a study at Sidney University [38] were Spark, Hadoop and Flink was compared. Hadoop was outperformed greatly by the other two frameworks because of its lack of in memory storing. The downside of Flink is that it does not, at this point, support fault-tolerance as in Spark where a fault easily can be fixed by computing the *lineage* of that data and recompute only what is necessary.

Apart from looking at frameworks that can incorporate everything themselves, there is another option to how to build pipelines. There exists several workflow tools, e.g. Luigi [39] and Nextflow [40], with as many different approaches and flavours of how to create pipelines using separate tools. By using these types of high level workflow tools, one could write each step in

the current pipelines as independent programs and simply pipeline them together using any of these workflow tools. In very big pipelines and workflows, this might be the preferred way but this has not been investigated further within this thesis. The research group is currently using the Luigi framework but prefers using only one framework as the Luigi approach had many drawbacks, such as the need for using several bash-programs to handle file transfers, need to write intermediate steps to file etc.

## 7.3 Spark in non-Big Data applications

Spark is design to be a Big Data [2] framework, meaning that it is focused on handling large amounts of data, usually in the range of hundreds of gigabytes and up to petabytes [41]. Big Data applications has huge gains in having data aware scheduling and distributed data between nodes (e.g. HDFS), which can minimise the amount of data sent between nodes. The typical Big Data applications are to crunch really big log-files and perform business intelligence queries on those, tasks that have little in common with the pipelines implemented in this thesis. The pipelines implemented in this thesis have used files of up to 100 megabytes of data, several orders of magnitude smaller than what is generally considered as Big Data. Instead of the rather uncomplicated log-files, the pharmaceutical problems have molecule data which requires the CDK library or similar software to just be able to read the data and generate descriptors. Overall, these properties makes these pipelines a lot more compute intensive than the general Big Data applications, which typically would push users towards HPC-environments to solve their problems.

# 8. Conclusions

Spark has been found to provide good means for the generation of pipelines for pharmaceutical purposes. The pipelines generated in this work have shown how implementation can be done in a straightforward way, programmatically very similar to the corresponding flowcharts. These pipelines can be expanded with further components and all components can be used in a plug-and-play fashion, letting the type system make sure that non-matching components are not linked to each other. A pipeline can also be considered as a pipeline-stage by itself and be reused in bigger pipelines, making it easy to compose bigger pipelines and reuse components.

Spark provides a good foundation to build on top of, as it has libraries for both machine learning and machine learning pipelines. These libraries are likely to be extended in the future as Spark has a big community of developers and open source supporters. The Spark ML pipeline is a promising feature that might be worth looking further into, it does not support the creation of the complete PTP pipeline developed in this thesis, but it might be useful to have as a sub-pipeline for handling the machine learning parts. The benefits of including ML pipelines would be the built in support for cross-validation, parameter grid searches and added testing that the open source community provides.

Finally, Spark might not be the optimal solution when only looking at the runtime, but it has many great advantages in other areas. The Spark framework simplifies the code writing as the programmer do not need to think about the scheduling and sending of data between processes, these issues are handled completely by the framework. Furthermore, Spark can be used as the 'complete' tool for creating the entire workflow, instead of mixing several different tools to automate things. As the goal for the project was to evaluate the use of Spark in creating pharmaceutical pipelines, the result has been that Spark do provide the means necessary to create pipelines and it does so with a good scaling in a distributed environment. Execution time also seams to beat the previous implementation that the group had for the PTP pipeline, which was a HPC-implementation using Luigi as workflow manager.

# Acknowledgements

First I would like to thank my supervisor Ola Spjuth for the opportunity to do this highly educational master's thesis at the FarmBio institution. The project has not only provided insights into Spark and pharmaceutical bioinformatics, but also taught me important concepts used in development of software and in research, concepts that will be important in the rest of my career.

I also wish to thank my family and my girlfriend Ellen for supporting me during all these years, trying to combine full time studies with a cycling career. This time has not been easy on any of us, but you have made it possible for me to accomplish this.

# References


[1] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Spark: Cluster Computing with Working Sets". In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'10. Boston, MA: USENIX Association, 2010, pp. 10–10.

[2] Rajendra Kumar Shukla, Pooja Pandey, and Vinod Kumar. "Big Data Frameworks: At a Glance". In: *International Journal of Innovations & Advancement in Computer Science* 4.1 (2015).

[3] Ola Spjuth, Tobias Helmus, Egon Willighagen, Stefan Kuhn, Martin Eklund, Johannes Wagener, Peter Murray-Rust, Christoph Steinbeck, and Jarl Wikberg. "Bioclipse: an open source workbench for chemo- and bioinformatics". In: *BMC Bioinformatics* 8.1 (2007), p. 59. ISSN: 1471-2105.

[4] Ola Spjuth, Jonathan Alvarsson, Arvid Berg, Martin Eklund, Stefan Kuhn, Carl Masak, Gilleain Torrance, Johannes Wagener, Egon Willighagen, Christoph Steinbeck, and Jarl Wikberg. "Bioclipse 2: A scriptable integration platform for the life sciences". In: *BMC Bioinformatics* 10.1 (2009), p. 397. ISSN: 1471-2105.

[5] Narendra Nyola, G Jeyablan, M Kumawat, Rajesh Sharma, Gurpreet Singh, and N Kalra. "Pharmacoinformatics: A Tool for Drug Discovery". In: *Am. J. PharmTech Res.* 2.3 (2012). ISSN: 2249-3387.

[6] Corwin Hansch. "Quantitative approach to biochemical structure-activity relationships". In: *Accounts of Chemical Research* 2.8 (1969), pp. 232–239.

[7] Jarl Wikberg, Martin Eklund, Egon Willighagen, Ola Spjuth, Maris Lapins, Ola Engkvist, and Jonathan Alvarsson. *Introduction to pharmaceutical bioinformatics*. Oakleaf Academic, 2010.

[8] Jonathan Alvarsson, Martin Eklund, Ola Engkvist, Ola Spjuth, Lars Carlsson, Jarl ES Wikberg, and Tobias Noeske. "Ligand-based target prediction with signature fingerprints". In: *Journal of chemical information and modeling* 54.10 (2014), pp. 2647–2653.

[9] Ovidiu Ivanciuc. "Drug design with machine learning". In: *Encyclopedia of Complexity and Systems Science*. Springer, 2009, pp. 2159–2196.

[10] Vladimir V Zernov, Konstantin V Balakin, Andrey A Ivaschenko, Nikolay P Savchuk, and Igor V Pletnev. "Drug discovery using support vector machines. The case studies of drug-likeness, agrochemical-likeness, and enzyme inhibition predictions". In: *Journal of chemical information and computer sciences* 43.6 (2003), pp. 2048–2056.

[11] Robert Burbidge, Matthew Trotter, B Buxton, and S Holden. "Drug design by machine learning: support vector machines for pharmaceutical data analysis". In: *Computers & chemistry* 26.1 (2001), pp. 5–14.

[12] Payam Refaeilzadeh, Lei Tang, and Huan Liu. "Cross-validation". In: *Encyclopedia of database systems*. Springer, 2009, pp. 532–538.

[13] *Apache Spark*. URL: `https://spark.apache.org/`.

[14] Holden Karau, Andy Konwinski, Patrick Wendell, and Zaharia Matei. *Learning Spark*. First. O'Reilly Media, Feb. 2015.

[15] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing". In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012, pp. 2–2.

[16] A Kala Karun and K Chitharanjan. "A review on hadoop—HDFS infrastructure extensions". In: *Information & Communication Technologies (ICT), 2013 IEEE Conference on*. IEEE. 2013, pp. 132–137.

[17] Tevfik Kosar. "A new paradigm in data intensive computing: Stork and the data-aware schedulers". In: *Genome* 40 (2006), p. 50.

[18] Tevfik Kosar and Mehmet Balman. "A new paradigm: Data-aware scheduling in grid computing". In: *Future Generation Computer Systems* 25.4 (2009), pp. 406–413.

[19] *Amazon Elastic Compute Cloud (Amazon EC2)*. URL: `http://aws.amazon.com/ec2/`.

[20] Andre Luckow, Mark Santcroos, Ashley Zebrowski, and Shantenu Jha. "Pilot-data: an abstraction for distributed data". In: *Journal of Parallel and Distributed Computing* (2014).

[21] *Spark Machine Learning Library (MLlib) Guide*. URL: `https://spark.apache.org/docs/1.3.1/mllib-guide.html`.

[22] Chieh-Yen Lin, Cheng-Hao Tsai, Ching-Pei Lee, and Chih-Jen Lin. "Large-scale logistic regression and linear support vector machines using Spark". In: *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE. 2014, pp. 519–528.

[23] *Spark ML Programming Guide*. URL: `https://spark.apache.org/docs/1.3.1/ml-guide.html`.

[24] *Spark Asynchronous RDD Actions API*. URL: `https://spark.apache.org/docs/1.3.1/api/scala/index.html#org.apache.spark.rdd.AsyncRDDActions`.

[25] *Apache Hadoop*. URL: `http://hadoop.apache.org/`.

[26] *The Chemistry Development Kit*. Dec. 2014. URL: `http://sourceforge.net/projects/cdk/`.

[27] *Uppsala Multidisciplinary Center for Advanced Computational Science*. URL: `http://www.uppmax.uu.se/`.

[28] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. "OpenStack: toward an open-source solution for cloud computing". In: *International Journal of Computer Applications* 55.3 (2012), pp. 38–42.

[29] Anna Gaulton, Louisa J Bellis, A Patricia Bento, Jon Chambers, Mark Davies, Anne Hersey, Yvonne Light, Shaun McGlinchey, David Michalovich, Bissan Al-Lazikani, et al. "ChEMBL: a large-scale bioactivity database for drug discovery". In: *Nucleic acids research* 40.D1 (2012), pp. D1100–D1107.

[30] A Patr´cia Bento, Anna Gaulton, Anne Hersey, Louisa J Bellis, Jon Chambers, Mark Davies, Felix A Krüger, Yvonne Light, Lora Mak, Shaun McGlinchey, et al. "The ChEMBL bioactivity database: an update". In: *Nucleic acids research* 42.D1 (2014), pp. D1083–D1090.

[31] *FreeMarker template engine*. URL: `http://freemarker.org/`.

[32] *Apache Maven*. URL: `https://maven.apache.org/`.

[33] *Ampcamp 5 - big data bootcamp*. Nov. 2014. URL: `http://ampcamp.berkeley.edu/5/`.

[34] *Amplab UC Berkeley*. URL: `https://amplab.cs.berkeley.edu/`.

[35] *Scala Documentation; Futures and Promises*. URL: `http://docs.scala-lang.org/overviews/core/futures.html`.

[36] *Apache Flink*. URL: `http://flink.apache.org/`.

[37] *Apache Crunch*. URL: `https://crunch.apache.org/`.

[38] Jack Galilee. *A STUDY ON IMPLEMENTING ITERATIVE ALGORITHMS USING BIGDATA FRAMEWORKS*. URL: http://sydney.edu.au/engineering/it/research/conversazione-2014/Galilee-Jack.pdf.

[39] *Luigi pipelining tool for batch jobs*. URL: https://github.com/spotify/luigi.

[40] *Nextflow - Data-diven computational pipelines*. URL: http://www.nextflow.io/.

[41] Hsinchun Chen, Roger HL Chiang, and Veda C Storey. "Business Intelligence and Analytics: From Big Data to Big Impact." In: *MIS quarterly* 36.4 (2012), pp. 1165–1188.

# Appendices

# A. Flowchart symbols

Here is a brief overview of the symbols used in the flowchart diagrams in the thesis, see fig. 15, which is a subset of the standard flowchart symbols. Colours are added for making it easier to read the charts and they are not general for all flowcharts.

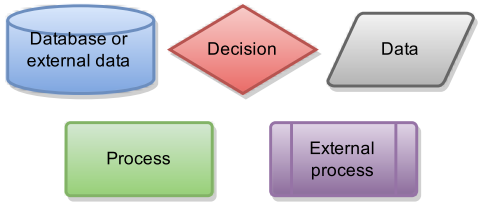


**Figure 15: Symbol lookup for the flowcharts in this thesis.** The blue symbol is typically associated with databases, but used here for symbolising data stored on some file system before execution of the pipeline. The red rectangle, labeled 'Decision', is a point were the flow is guided between different directions depending on the input. The gray rectangle symbolises input and output data from processes. The green rectangle symbolises processes that do manipulations on the data in the system. The purple rectangle is the end process, external to the pipeline, that gets the final results.

# B. Implementation of PipelineSplitter

Fig. 16 presents an implementation of the `PipelineSplit` type defined in the `PipelineAPI` in fig. 3. This type only needs to be implemented once in your project, the type parameters of the class will take care of all potential input types you might want to throw at it. Using the `PiplineSplitter` can be done in the same fashion as in fig. 9, where the two branches are defined and then given as parameters to the construction of a new `PipelineSplitter` instance. Other implementations of the `PipelineSplit` type may give other possibilities depending on personal preference.

```
1  class PipelineSplitter[T,S](left: (T=>Any), right: (S=>Any))
2    extends PipelineSplit[T,S] {
3
4     def apply(in: Either[T,S]): Any = {
5        in match {
6           case Right(input) => return right(input);
7           case Left (input) => return left (input);
8        }
9     }
10 }
```

**Figure 16: Implementation of the `PipelineSplitter`.** This is an implementation of the `PipelineSplit` in the `PipelineAPI` defined in fig. 3. This class only needs to be implemented once, and the type parameters `T` and `S` will be inferred once instantiated in an object; `new PipelineSplitter(left,right)` should be sufficient for creating a splitting event. Note that this is only an example implementation and it can be implemented differently.