Predicting job resource utilization based on historical data

Matthew Swart - 5597250

April 2, 2018

Daily supervisors:
Robert Kreuzer
Dr. Wouter Swierstra

Second examiner:
Dr. Wishnu Prasetya

Department of Computing Science
University of Utrecht

Abstract

We present a generic prediction framework in Haskell. This framework consists of two parts. The first part introduces a fixed structure for supporting various prediction models. A few of those models have been implemented. The latter part introduces a DSL to guide the programmers in making predictions. The interpretation of this DSL can be defined in an arbitrary fashion to allow different semantics. Furthermore, we also elaborate and discuss a few examples of the framework. One of these examples is a case study at Channable. This case study tries to improve the utilization of server resources by first estimating the resource usage of a specific job and then using that estimate to schedule the job on an appropriate server. Finally, this scheduling algorithm is compared with the original algorithm by modeling a theoretical simulation.

Acknowledgments

First of all, I would like to thank Wouter Swierstra for his advice and feedback. Next, I would like to thank Channable for giving me the opportunity to write my thesis on-site. I also want to thank the employees of Channable for the fun and informative time. In particular, I would like to thank Robert Kreuzer for his many ideas and suggestions.

Contents

1 Introduction

Technology enables companies to automate repetitive tasks like sending the daily newsletter to all customers or making backups. In the digital world, many of these tasks can be performed automatically at a specific time by a task scheduler. The task scheduler keeps track of the planning activities and carries them out. This has the advantage that tasks will not be forgotten to be executed. The tasks can be scheduled in an arbitrary manner. When jobs use on a regular basis a lot of computing power, it might be better to spread the work of the jobs across multiple servers. The total amount of servers needed to carry out these jobs depends on the required resources needed for performing the jobs. The resource usage of a job is, therefore, an important factor in planning the job. Resource information of new incoming job can be obtained by making estimates that are based on the resource use of previous jobs. This kind of information can be included in the planning of the jobs. Let's say we need to schedule three tasks over N servers. When the tasks are sent to the task scheduler, we also include the estimated resource usage of the task. When scheduling the tasks, we first look at the current resource usage of each of the N servers. The scheduler then assigns the task to the server that is most suitable for executing the task. Maximizing the throughput of the jobs allows more jobs to be carried out in the same time frame. Making such estimates often requires in-depth knowledge about the different types of models. We, therefore, introduce a generic prediction framework that helps to make such estimates. Next, for a proof-of-concept, this framework will be applied in a case study to help with the scheduling of jobs.

1.1 Case study: Channable

The number of platforms on which digital products can be offered is enormous and grows by the day. Manually offering all of the different products to each of the channels requires a lot of time. Channable has developed a tool for distributing digital products to various online channels such as Marktplaats and eBay. The customers of Channable use this tool to manage their own products on the various channels. The products of the customers consist of all kinds of items that can be offered across the web. For example, the items of a webshop, but also sports activities can be shared across the different channels. For adding products to the tool, they support various input formats such as XML, CSV, and Magento. All of these different input formats can be submitted simultaneously. Then Channable merges the products and delivers them in parallel to all the supported channels. Each of these operations follows a strict order of operations. That is why each of these tasks is performed by a batch job.

1.2 Motivation

Machine Learning explores the data to make predictions about a particular domain. This area focuses on data-driven algorithms for constructing models

and use those models to make predictions. These predictions are often based on historical events. Suppose team X won 18 out of 20 times from team Y. Then, we expect that based on this information, team X will win again at a new meeting between the two teams. The use of these algorithms often requires in-depth knowledge of the different models. Many popular higher-level programming languages often already consist of an implementation of the popular Machine Learning models. These models are often implemented by experts in the field of Machine Learning and the specific language. This ensures that these types of models are often implemented in an efficient manner. Therefore, it is not wise for a normal programmer to manually implement these models. It is better to use the libraries developed by the community. The use of these type of models is often followed by a fixed procedure. First of all, a form of historical data will be loaded into the environment. Next, this data will be analyzed by a self-learning algorithm. The result of this analysis can be used to make estimates of certain events. Unfortunately, the components of this procedure are often subdivided into different libraries. This is also the case in Haskell. Finding and understanding the various components takes a long time. We, therefore, develop a GPF(generic prediction framework) in Haskell that assists the programmer in making predictions. The library provides an interface for writing and using models that could be used to predict continuous labels. It is developed in Haskell due to its type-safety, yet flexible possibilities of the language. The disadvantage of a functional programming language is that it is less easy to define a sequence of operations. In Haskell, you can easily solve this problem by using lets and wheres. However, this can quickly become unclear when we need a lot of continuations, which is likely to happen when we want to make several predictions. In order to improve this process, we added a DSL to the library that guides the programmers to make multiple predictions. This DSL also allows additional functionality like loading a CSV file to the environment.

Channable developed its own job scheduler tool for managing their jobs. This tool is written in Haskell and it's called jobmachine. The jobmachine consists of two parts. The first part of the jobmachine determines the schedule of when jobs have to be executed. The second part delivers the jobs to a small but variable number of workers. Each of the workers has a limit of X slots available for the jobs that can be performed at once. The following four properties need to hold for the job scheduler:

1. Job priority must always be honored: Higher-priority jobs (e.g. interactive jobs from users) must always run before lower priority jobs
2. Jobs may only run if all their dependencies are satisfied
3. Jobs should run as close as possible to the time that they were scheduled (without violating the first two requirements of course)
4. Jobs should never crash (the code must be correct and the workers need to consists of enough resources

The first two properties are satisfied by the current schedule. The third is satisfied as long as there are enough free worker slots available. However, this property is violated when all the slots are filled. This means that new jobs that are satisfied by the above criteria still have to wait until there is a slot free. This violation is currently solved by providing the servers with a lot of CPU, memory and network throughput. Much more than actually needed. This is known as over-provisioning.

Despite the over-provisioning, there are some very memory intensive jobs, that forces the server to use more memory than available in the RAM. The Operating System solves this by using VM(Virtual Memory,) which is a memory management method that is implemented by using both hardware and software. The purpose of a VM is to map virtual addresses of a process into the physical memory. The VM uses as default storage for the processes the RAM of a computer. The RSS(Resident set size) is the amount of memory allocated for the process in the ram memory. However, the virtual memory switches to the secondary storage when a process uses more RSS than available. This part of the memory is called the swap space. In the worst case, it could kill the jobs when the server runs out of swap space. This does not happen in practice due over-provisioning. This brings us back to the first problem: The current Jobmachine makes inefficient use of the resources since it does not know how many resources a job needs. They have to over-provision and even then they cannot guarantee that they will not accidentally schedule a lot of memory-heavy jobs to the same worker.

From this, we conclude that over-provisioning of the servers is not the most optimal solution. It's better to optimize the workload of the servers, such that we can maximize the throughput. In addition to the more efficient execution of the jobs, it can also reduce the cost of the necessary hardware.

We will realize this by taking into account the resource usage of a job while planning the distribution of the jobs to the various servers. Although it's not possible to take into account the exact resource usage, we are going to make estimates about it. The estimates are based on a historical dataset that consists of the domain knowledge of the jobs. Various statistical methods are going to be applied to this dataset to obtain the models. Then these models are used to predict the resource usage of new incoming jobs. After obtaining the results of the prediction, we look at the current hardware usage of the servers. Then we combine these two usages to determine the server that is most suitable for performing the job. The goal is to keep the workload as high as possible, but with a sufficient probability that the servers do not use the swap space.

1.3 Research Question

The goal of this thesis is to develop a framework that assists the programmers in making predictions. We apply this framework to a concrete use case at Channable. For investigating this problem, we defined the following research question: "Can we improve the resource utilization by using estimated resource usage of a job as meta-information for scheduling a job at Channable?". This

question is divided into the following three subquestions:

1. Is it possible to define a framework in Haskell to implement and use various prediction models?
2. Can we predict the resource usage of a job with sufficient accuracy?
3. Can we use the predictions to improve the resource utilization?

The following sections will provide more details on all of these questions.

Is it possible to define a framework in Haskell to use and implement various prediction models? This question investigated what the possibilities are for defining a generic prediction framework. The main purpose of the framework is that it can be used in Haskell as an auxiliary tool for creating and applying various prediction models in a structured fashion. Eventually, this framework is used to estimate the dependent variables of the jobs.

Can we predict the resource usage of a job with sufficient accuracy? The first part of this thesis investigated whether it is possible to make predictions about the resource usage with a sufficient accuracy. Making these predictions requires a historical dataset that consists of the domain knowledge of the jobs. Prior knowledge of Channable is used to determine the domain knowledge of the jobs. Construction the dataset is done in the initial phase of this question. Then we will apply various simple statistical models(e.g. mean, standard deviation and variance) on the dataset. The results of these simple models are used to visualize the data in a more human-readable format. By using prior knowledge and the result of the previous models we tried to identify several patterns in the data. These patterns are used to define a strategy to obtain for each of the relevant resource properties of a job a sufficient prediction accuracy. This strategy includes applying a supervised learning algorithm and feature selections on the selected subsets. All of these approaches are compared with each other. From this comparison, the model that resulted in the best outcome will be chosen to estimate the resource usage of future jobs.

Can we use the predictions to improve the resource utilization? This question investigated whether the proposed scheduling algorithm improves the resource utilization. The goal of the new scheduling algorithm is to optimize the utilization of the available resources while minimizing the risk of going over capacity.

1.4 Thesis Structure

This section provides an overview of the thesis. The next chapter describes the inner workings of the generic prediction framework. We first explain the two main components (Interface and DSL) of the framework. The next subchapter

discusses a few optimizations. Finally, we describe what kind of tests have been added to the framework.

The third chapter describes how we obtained the estimated workload. The first subchapter discusses which resource usages affect the performance of the different jobs. The next subchapter explains various insights of a dataset that consists of the domain knowledge of the jobs. 80% of this dataset is used to determine the most suitable prediction models.

The remaining 20% of the dataset is used during the drafting of a scheduling simulation. Chapter four discusses the result of this simulation. During this simulation, we compared the original scheduling algorithm with the new proposed algorithm. Chapter five discusses the related work of this thesis. Furthermore, chapter six explains the limits and possible additions to this study. Finally, the research questions are answered in the last chapter.

2 Generic Prediction framework

This chapter discusses the inner workings of the generic prediction framework. The predictions of this framework will be based on historical data. Making these predictions requires, therefore, a dataset of that particular domain. This dataset will then be analyzed to obtain the predictions. The framework consists of two parts. The initial part introduces an interface that allows programmers to implement several statistical methods in a structured fashion. The latter part introduces a DSL that assists the programmer in describing prediction programs.

2.1 Interface

The DSL of the framework requires a fixed and generic interface that can be used to implement and use various prediction models. There are several libraries available that assists the user in making predictions based on historical data. Unfortunately, these libraries often consist of different structures and function names for making the predictions. This makes it unsuitable for the DSL. We, therefore, introduce a generic interface to create various models and use these models to predict real values. The goal of this interface is to develop a fixed structure that makes it possible to systematically train and use different models. At the time of writing, one library[12] is known that developed a similar interface. Unfortunately, this library is not well documented and is no longer maintained. This study will, therefore, use a self-developed interface.

2.1.1 Definition

The two main approaches to train data are online and batch algorithms. In the case of batch algorithms, the entire data is used for training the model. Online algorithms can be used to stream new data points to the current model. This has the advantage that the entire dataset does not have to be re-trained. The final

version of the framework only supports batch learning algorithms. However, in the optimization section, we discuss which adjustments must be made to the interface to comply with online algorithms. Furthermore, the interface only focusses on a subset of supervised learning. Supervised learning algorithms are interested in constructing a mapping function that takes x as input variables, and as a result y. This can be formally written as a function f (x) = y. We mainly focus on regression problems. These are problems where the outcome is a real value, such as the total amount of euros. The interface only offers support for a single variable at y.

Eventually, each of supported models needs to have an implementation for the following functions:

$$train :: X\ a \rightarrow Y\ a \rightarrow model\ a$$

$$predict :: X\ a \rightarrow model\ a \rightarrow a$$

The X should be represented as a Matrix and the Y as a sequence of values.

2.1.2 Type class

The purpose of the interface is to maintain the same structure(eg. same function names) for using different models. Haskell is a strongly typed language. This makes it an ideal language to indicate at type level which model will be used. The use of the same functions can be realized by overloading functions. This is supported in Haskell by using type classes. When calling a function of the type class, Haskel determines at type level which instance of type class is used.

Representing multiple input variables is realized by the *Matrix* data type from the Matrix[10] package. The advantage of this package is that it contains several mathematical Matrix operations. These are useful to assists the user in calculating a mapping function. Internally the *Matrix* uses the *Vector* from the Data.Vector[17] module. The advantage of this vector is that it is compatible with other parts of the framework.

Finally, we developed the following interface:

```
1  data TrainingStructure datapoint = TrainingStructure{
2    xData :: Matrix datapoint,
3    yData :: Vector datapoint
4  }
5
6  data PredictStructure model datapoint = PredictStructure{
7    model :: model datapoint,
8    datapoint :: Vector datapoint
9  }
10
11 class Interface model datapoint where
12   mkTrain :: TrainingStructure datapoint → model datapoint
13   mkPredict :: PredictStructure model datapoint → datapoint
```

The datatypes *TrainingStructure* and *PredictStructure* are used to make it more clear to the programmer which information is needed because let's say the

Interface would allow multiple dependent variables. It's likely that this would be represented as a Matrix. In this case, the *TrainingStructure* requires for both of the arguments a *Matrix*. When we omit the *TrainingStructure*, it would be unclear to the user whether it is *xData* or *yData*.

2.1.3 Models

This section covers the implemented statistical models. We focused on rather simple models:

1. Average
2. Minimum
3. Maximum
4. Ordinary least square

The usage of the interface is described using an example.

Average A well-known statistical method is the average. zThe interface requires that each of the models consists of a datatype. The average is represented using the following datatype:

```
data Avg dp = Avg {avg :: dp} deriving (Show, Read, Eq, Generic)
```

This allows us to define the functions of the interface:

```
instance Fractional a => Interface Avg a where
  mkTrain ms =
    let values = yData ms
    in Avg $ sum values / fromIntegral (length values)
  mkPredict = avg . model
```

Minimal and Maximal Computing the minimal and maximal is also supported by the interface:

```
data Min a = Min {getMin :: a} deriving (Generic, Eq, Show)
instance (Ord a) => Interface Min a where
    mkTrain = Min . minimum . yData
    mkPredict pr = getMin $ model pr


data Max a = Max {getMax :: a} deriving (Generic, Show, Eq)
instance (Ord a) => Interface Max a where
    mkTrain = Max . maximum . yData
    mkPredict pr = getMax $ model pr
```

Ordinary Least square The above models do not rely on the supplied input variables. As a result, the *mkPredict* function always results in the same value as the corresponding model. Predictions often depend on multiple input variables. One of the models that can deal with multiple variables is the ordinary least square. This model is based on the following equation:

$$y = X\beta + \epsilon$$

$$\hat{\beta} = (X^T X)^{-1} X^T y$$

The Matrix package consists of various functions to solve the above equation. It is not always possible to compute the inverse of a matrix. In this case, the inverse function returns an error message to the user. Unfortunately, this error message is not always correct. We have therefore chosen to return a generic error message. We did this by catching the error message of the inverse function:

```
computeInverse :: (Fractional a, Eq a, NFData a)
  => Matrix a
  -> Either SomeException (Either String (Matrix a))
computeInverse x = unsafePerformIO $ try evaluate force (inverse (
    multStd2 (transpose x) x))
```

The use of *unsafePerformIO* is safe because the function does not produce any side effects. Next, we added the ordinary least square to the interface:

```
data OLS a = OLS{
  intercept :: a,
  coefficients :: Vector a
} deriving (Show, Read, Generic, Eq)

instance (Show a, Read a, Fractional a, Eq a, NFData a) =>
    Interface OLS a where
  mkTrain ms = linearRegression (xData ms) (colVector $ yData ms)
  mkPredict (PredictStructure ols dp) =
    foldr go (intercept ols) $ zip dp (coefficients ols)
    where go (a,b) c = (a * b) + c


linearRegression ::
    (Read a, Show a, Eq a, Fractional a, NFData a)
    => Matrix a
    -> Matrix a
    -> OLS a
linearRegression x t =
  let rows = nrows x
      x' = force $ fromList rows 1 (replicate rows 1) <|> x
  in case computeInverse x' of
           Right (Right m2) -> toOLS $ getMatrixAsVector $
                    multStd2 (multStd2 m2 $ transpose x') t
           _ -> error "Linear regression went wrong."


toOLS :: V.Vector a -> OLS a
toOLS vs
```

```
  | not $ null vs =  OLS (head vs) (tail vs)
  | otherwise = error "Result of OLS is empty"
```

2.1.4 Example

This section describes the usage of the interface by giving an example. Suppose we are interested in the expected score of a player on a basketball team. To find the expected outcome, we first collect the height, weight, and score of the previous players. This allows us to compute different types of models. First of all, we are interested in the average of the score. This is calculated by the following function:

```
avgScore = [9.2, 11.7, 15.8, 8.6, 23.2, 27.4, 9.3, 16, 4.7]
height' = [6.8, 6.3, 6.4, 6.2, 6.9, 6.4, 6.3, 6.8, 6.9]
weight' = [225, 180 ,190, 180,205,225,185,235,235]

trainAvg :: Avg Double
trainAvg = mkTrain TrainingStructure{
  xData = Matrix.transpose $ Matrix.fromLists [height',weight'],
  yData = Vector.fromList avgScore
}

predictAvg :: Double
predictAvg = mkPredict PredictStructure{
    model = trainAvg,
    datapoint = Vector.fromList [6.5, 200]
}
> predictAvg
> 13.99
```

Then we are curious whether the height and weight of the player provide extra information about the score. We include this information during the executing of the ordinary least square. This can be easily arranged by changing the type of the *trainAvg* function to the *OLS* datatype:

```
getAvgAge :: OlS Double
> predictAvg
> 13.63
```

Changing the type, completely changes the behavior of this function without changing the structure of the program.

2.2 DSL

2.2.1 Primitives

Haskell is a purely functional language. Let's say we need a dataset from the file system. Loading such a file produces side effects. However, when we manually insert the dataset as code, we do not produce any side effects. In this case, it is better to keep the code pure. We, therefore, propose a number of primitives that can be used to describe the procedure. Then this description can be interpreted in various ways. We defined the primitives as an ADT based on the Free monad.

The Free monad makes it easy to distinguish between the logic and the data types. The free monad is defined in the following way:

```
data Free f r = Free (f (Free f r)) | Pure r
```

As a constraint, the Free monad states that the f is a Functor. The advantage of such a constraint is that it is easy to write a sequence of Functors. We are using the Free monad of the package Free[14]. This package contains several useful functions that assist in using the Free monad. The description of the DSL is subdivided into the three main components(dataset, training, predicting).

Dataset The first part of the GPF is focused on loading data into the environment. There are many different types of formats to store data (eg CSV, MySQL and PostgreSQL). For this framework, we decided to focus on the CSV. Nevertheless, we expect that the proposed approach is also applicable for the different data formats. To represent the data, the DSL works with the same type of Vectors as the interface. This Vector is compatibility with the CSV parser Cassava. Cassava is a library for parsing CSV files to Haskell code. It uses the type class FromRecord to parse the CSV. This type class contains a function for parsing a sequence from Bytestrings to an arbitrary type. The type class contains an instance for the Generic data type. This makes it easier to create an instance for new datatypes because GHC consists of a mechanism for automatic deriving generics for arbitrary types. This vector is wrapped around the *Dataset* datatype:

```
data Dataset schema = Dataset {
    _database :: Vector schema
} deriving (Show, Functor)
```

Working with wrappers has the advantage that the primitives know whether the origin is from another primitive or outside this framework. It's also convenient to store a lot of information in a compact format at the type level.

Training a model often does not require all of the attributes of a CSV, rather a subset of the attributes. This is described in statistics as a feature selection. The DSL represents the feature selection using the following wrapper:

```
newtype Feature schema datapoint =
    Feature {getFeature :: [schema → datapoint]}
```

This datatype wraps a list of record names of the corresponding schema datatype. This allows the programmers to define all the attributes that are needed as input for the training function. This provides us with enough information to define the first three primitives:

```
data Instruction next where
    LoadDataset :: Vector.Vector schema → (Dataset schema → next)
    → Instruction next
    LoadDatasetFromFile :: FromRecord schema ⇒
          FilePath → (Dataset schema → next) → Instruction next
    FeatureGPF ::
        [schema → datapoint]
        → (Feature schema datapoint → next)
        → Instruction next
```

We represent the primitives with the Instruction ADT. The first constructor allows the user to directly represent the data in Haskell code. The second constructor requires a *FilePath* as an argument. The idea is that the file located at this *FilePath* will be loaded to the Haskell environment by one of the interpreters. The last construction is used to represent the Feature.

Training Training a model requires various types of information about the model. First, we need a dataset. Next, it must be clear which attributes are used as input variables of this dataset. Subsequently, an attribute must be defined on which the prediction is based. Finally, it must be clear which model should be trained. Using such a model requires access to this kind of information. We introduce the following *Model* wrapper to store this kind of information:

```
data Model schema datapoint model = Model {
    _modelfeature :: Feature schema datapoint,
    _modelResponse :: schema → datapoint,
    _modelDef :: model datapoint,
    _statistics :: Statistics
}

newtype Statistics = Statistics{
    timeInSeconds :: Maybe Double
} deriving Show
```

The *Model* datatype also contains statistics about creating the model. Currently, this datatype only supports the statistic to measure the time of computing the model. Working with *Statistics* has the advantage that in later stages of the framework we can easily extend this with additional statistics. Measuring the time requires the use of IO. It is however not necessary to measure the time. We will, therefore, add an option to the train instruction to configure this kind of meta information. This is determined by the following data type:

```
data MetaInfo = MetaInfo{
    pathObject :: Maybe FilePath,
    timingEnabled :: Bool,
    validateCorrectness :: Bool
}
```

Training a model could take a lot of time. It is better to avoid time-consuming operations. We, therefore, introduce the option to store a model to the filesystem. This allows the usage of the same model multiple times, without retraining. The pathObject defines the location of where the model should

be stored. The last argument of *MetaInfo* is discussed in the error prevention section.

Storing a model requires a function to convert a model to a sequence of characters, and vice versa. This forces us to use a certain data format. We chose to use JSON. This is a standardized format that is easy to read and write for both the computer and programmer.

Fortunately, there exists a package in Haskell that allows conversion between a datatype and the corresponding JSON format. This is arranged by the aeson[22] package. Converting a datatype to JSON is solved by the *ToJSON* type class. The type class *FromJSON* is used to convert a JSON format to a particular datatype. The datatype of a model requires, therefore, an instance for both of the type classes. Fortunately, there is an instance of the Generic representation.

To support the explained functionalities, we added four additional constructors to the Instruction ADT. These will be discussed in the remainder of this section.

```
data Instruction next where
    Train :: (Interface model datapoint,ToJSON (model datapoint),
    NFData (model datapoint))
        => MetaInfo
        → Dataset schema
        → Feature schema datapoint
        → (schema → datapoint)
        → (Model schema datapoint model → next)
        → Instruction next
```

The *MetaInfo* of the *Train* constructor is used to indicate which meta information should be included in the calculation of a model. The second argument represents the historical dataset that will be used to train the model. The *Feature* determines which attributes of the dataset will be used as the *xData*. The fourth argument is used to indicate which attribute should be used as the *yData*. The last argument is a function that serves as the continuation of the free monad. We also added an alternative training constructor to the instructions. This constructor does virtually the same, but also offers support for filtering the dataset. This has been implemented by the datatype below:

```
data Instruction next where
    TrainWithFilter :: (Interface model datapoint, ToJSON (model
    datapoint), NFData (model datapoint))
        => MetaInfo
        → Dataset schema
        → Feature schema datapoint
        → (schema → datapoint)
        → (schema → Bool)
        → (Model schema datapoint model → next)
        → Instruction next
```

However, it is not always necessary to train a model. For example, the user could also manually implement a model using the corresponding datatype of the model. We, therefore, introduce the following constructor:

```
data Instruction next where
    LoadModel :: ToJSON (model datapoint)
        => model datapoint
        → Feature schema datapoint
        → (schema → datapoint)
        → (Model schema datapoint model → next)
        → Instruction next
```

The next instruction describes that a model should be loaded from the filesystem:

```
data Instruction next where
    LoadModelFromFile :: (Interface model datapoint, FromJSON (
    model datapoint),ToJSON (model datapoint))
        => FilePath
        → Feature schema datapoint
        → (schema → datapoint)
        → (Model schema datapoint model → next)
        → Instruction next
```

Predicting The third part of the framework focusses on making predictions. We introduce the following *Predict* constructor:

```
data Instruction next where
    Predict :: (NFData datapoint, Interface model datapoint, Show
    datapoint)
        => MetaInfo
        → Model schema datapoint model
        → schema → (datapoint → next)
        → Instruction next
```

2.2.2 Design

In order to use the *Instructions* as a Monad, we have to lift each of the constructors to a free monad. To improve the readability of the code, we defined the following type synonym to represent the *Instruction* as free monad:

```
type GPF a = Free Instruction a
```

The Instructions are lifted by using the *liftF* function from the Free package. As an example, we write the *Predict* constructor in the following style:

```
liftF $ Predict mempty model input id
```

It is annoying for a user to give this extra information every time. Smart constructors solve this problem by adding extra constraints to a value. Each of the smart constructors is represented as a function. An example of a smart constructor for the *Predict* instruction is defined in the following fashion:

```
predict :: (NFData datapoint, Interface model datapoint, Show
    datapoint)
    => Model schema datapoint model → schema → GPF datapoint
predict mdl row  = liftF $ Predict mempty mdl row id
```

Many of the other smart constructors have a similar structure. These are therefore not covered in this thesis. Smart constructors also offer the option of simply adding more functionality. This allows us to make multiple predictions by mapping over the predict constructor:

```
predicts :: (NFData datapoint, Interface model datapoint, Show
    datapoint)
  => Model schema datapoint model
  -> [schema]
  -> GPF [datapoint]
predicts mdl rows = mapM (predict mdl) rows
```

When training the models, it is not always necessary to provide metaInformation. Adding smart constructors allows us to distinguish between these two kinds of training approaches. An example of this distinction is the functions *train* and *trainWithMetaInfo*.

```
train ::
    (Interface model datapoint, ToJSON (model datapoint),
    NFData (model datapoint))
    => Dataset schema
    -> Feature schema datapoint
    -> (schema -> datapoint)
    -> GPF (Model schema datapoint model)
train ds ft project = liftF $ Train mempty ds ft project id


trainWithMetaInfo ::
    (Interface model datapoint, ToJSON (model datapoint),
    NFData (model datapoint))
    => MetaInfo
    -> Dataset schema
    -> Feature schema datapoint
    -> (schema -> datapoint)
    -> GPF (Model schema datapoint model)
trainWithMetaInfo mInfo ds ft project = liftF $ Train mInfo ds ft
    project id
```

It is not elegant to write the *MetaInfo* constructor directly in a function.This makes the code less readable. It is better to define an instance to the Monoid type class, such that we can add various Monoid operations. This allows us to configure the meta information using independent functions. We added the following Monoid instance:

```
instance Monoid MetaInfo where
    mempty = MetaInfo Nothing False False
    mappend (MetaInfo a b c) (MetaInfo a' b' c') =
        MetaInfo (a <|> a') (b || b') (c || c')
```

Subsequently, a number of auxiliary functions have been defined for configuring the meta information. This is done by the following functions:

```
allMeta :: FilePath -> MetaInfo
allMeta filePath = store filePath <> measureTime <> validate

```

```haskell
store :: FilePath → MetaInfo
store filePath = MetaInfo (Just filePath) False False

measureTime :: MetaInfo
measureTime = MetaInfo Nothing True False

validate :: MetaInfo
validate = MetaInfo Nothing False True
```

2.2.3 Evaluators

The explained *instructions* allow the users to describe a program. Interpreting these instructions can be done in an arbitrary manner.

This gives users the flexibility to decide for themselves how the instructions should be evaluated. The current framework contains three different evaluators, but this can easily be extended by the user. For example, it would be interesting to evaluate the instructions in parallel. At the time of writing this thesis, the following evaluators are supported:

```haskell
eval :: GPF a → a
evalIO :: GPF a → IO a
evalMaybe ::GPF a → Maybe a
```

To evaluate the *Instruction*, we defined various auxiliary functions to assists in interpreting certain components of the DSL. These are also applied in each of the pre-implemented evaluators. The first function allows us to load CSV files from the file system to the Haskell environment:

```haskell
import qualified Data.Vector as V
import qualified Data.ByteString.Lazy as BL
import Data.Csv

readCSVFromFile :: (FromRecord schema) => FilePath → IO (V.Vector
    schema)
readCSVFromFile filePath = do
    content <- BL.readFile filePath
    case decode HasHeader content of
        Right x → return x
        Left y → error y
```

This function forces that each of the provided CSV files needs to consists of a header. This reduces the risk of selecting the wrong column because a header describes each of the fields explicit. The second function parses a file to a particular model. This function requires that the file must conform to a JSON format. This has been implemented by *readModelFromFile*:

```
readModelFromFile ::
    (Interface model datapoint, AE.FromJSON (model datapoint))
    => FilePath
    -> Feature schema datapoint
    -> (schema -> datapoint)
    -> IO (Model schema datapoint model)
readModelFromFile filePath ft project =
  do
      let cantReadModelMsg = error "Can't read model from file"
      mdl <- AE.decode <$> BS.readFile filePath
      return $ Model ft project (fromMaybe cantReadModelMsg mdl)
    emptyStatistics
```

The AESON package is used to convert a JSON format to a particular datatype. The evaluation of the meta information can be performed by one of the following two functions (*mdlMetaInfo* and *predMetaInfo*):

```
predMetaInfo ::
    (NFData predict, Show predict)
    => MetaInfo
    -> predict
    -> IO predict

mdlMetaInfo :: (NFData (model datapoint), ToJSON (model datapoint))
    => MetaInfo
    -> Model schema datapoint model
    -> IO (Model schema datapoint model)

measureInSeconds :: NFData a => a -> IO (Double,a)
```

The *mdlMetaInfo* evaluates the meta information for training a model. The *predMetaInfo* calculates the meta information for measuring the prediction. Both of these functions use the function *measureInSeconds* to measure the time that an expression takes to evaluate to normal form. The next function that will be covered assists in training:

```
train ::
    (Interface model datapoint, NFData (model datapoint)) =>
       Dataset schema
    -> Feature schema datapoint
    -> (schema -> datapoint)
    -> Maybe (schema -> Bool)
    -> Model schema datapoint model
train ds ft yProject rule =
    Model ft yProject mdl nostats
    where
        ds' = filterDs rule ds
        mdl = mkTrain TrainingStructure{
            xData = fromLists $ toList
                                $ fmap (toRow $ getFeature ft) ds',
            yData = map yProject ds'
        }
```

This function uses the interface to train the models. The remaining of the evaluation section discusses each of the implemented evaluators.

evalIO The *evalIO* evaluates the *GPF* in the *IO* monad. This evaluator supports all of the explained functionalities. However, when the provided DSL does not require *IO* operations, it is better to use one of the remaining evaluators. The *evalIO* is implemented in the following way:

```
evalIO :: GPF a → IO a
evalIO (Free (LoadDataset ds next)) = evalIO $ next (Dataset ds)
evalIO (Free (LoadDatasetFromFile filePath next)) =
    readCSV filePath >>= evalIO . next . Dataset
evalIO (Free (LoadModel mdl ft project next)) =
    evalIO $ next $ Model ft project mdl nostats
evalIO (Free (LoadModelFromFile filePath ft project next)) = do
    x <- readModelFromFile filePath ft project
    evalIO $ next x
evalIO (Free (Train mInfo ds ft project next)) = do
    x <- mdlMetaInfo mInfo $ train ds ft project Nothing
    evalIO $ next x
evalIO (Free (TrainWithFilter mInfo ds ft project rule next)) = do
    x <- mdlMetaInfo mInfo $ train ds ft project (Just rule)
    evalIO $ next x
evalIO (Free (Predict mInfo mdl row next)) = do
    x <- predMetaInfo mInfo $ predict mdl row
    evalIO $ next x
evalIO (Free (FeatureGPF ft next)) = evalIO $ next (Feature ft)
evalIO (Pure x) = return x
```

All of the remaining evaluators have a similar coding structure.

eval The second evaluator is the *eval* function. The purpose of this function is to remain as pure as possible. This has the disadvantage that we cannot compute IO operations(Like the constructor *LoadModelFromFile*). Nevertheless, the instructions offer the possibility to write them. In this case, the eval throws an error.

evalMaybe The final evaluator is the function *evalMaybe*. This function supports the same functionalities as the eval, but without having to throw an error. Instead, when an instruction occurs that requires IO operations, we return the Nothing constructor. In all other cases, the result will be wrapped around the Just constructor.

2.2.4 Example

The basketball example from the interface section is also implemented using the entire framework. This section discusses the implementation. The statistics are based on a CSV file that is stored at the filesystem. Furthermore, we are interested in the following statistics:

1. Percentage of field goals of 100 attempts
2. Percentage of Free throws of 100 attempts
3. Average score per game

The CSV also consists of the additional attributes height and weight of the players.The schema of the CSV file is represented using the following ADT:

```
data Player = Player {
  height :: Double,
  weight :: Double,
  fieldGoals :: Double,
  freeThrows :: Double,
  avgPerGame :: Double
} deriving (Generic, Show, Eq)

instance FromRecord Player
```

Next, we used the generic representation of *Player* to use it as CSV schema. Furthermore, we used the height and weight as input variables. This is established by defining the *ftPerson* feature:

```
ftPerson :: [Player → Double]
ftPerson = [height, weight]
```

Next, we add various type synonyms to improve the readability of the code.

```
type Mdl mdl = Model Player Double mdl
type Mdls mdl = (Mdl mdl, Mdl mdl, Mdl mdl)

data ScorePrediction = ScorePrediction{
  fieldGoalsP :: Double,
  freeThrowsP :: Double,
  avgPerGameP :: Double
} deriving Show
```

The statistics are based on the ordinary least square and the average. Training these models requires the same procedure. We, therefore, introduce a generic description for training and using a model. This description can be interpreted in various ways:

```
genericResult ::
  (Interface mdl Double, ToJSON (mdl Double), NFData (mdl Double))
  => Player → Dataset Player
  → GPF (ScorePrediction, Mdls mdl)
genericResult p ds = do
  ft <- feature ftPerson

  fieldGoalsM <- train ds ft fieldGoals
  threeThrowsM <- train ds ft freeThrows
  perGameM <- train ds ft avgPerGame

  fieldGoalsP <- predict fieldGoalsM p
  threeThrowsP <- predict threeThrowsM p
  perGameP <- predict perGameM p
  return  (ScorePrediction fieldGoalsP threeThrowsP perGameP,
           (fieldGoalsM, threeThrowsM, perGameM))
```

Next, we define the function *predictStatistics*. This function requires as first argument a player with the height and weight. Then it describes the procedure of the prediction. This procedure first wants to load the file "basketball.csv"

to the environment. This requires the *Player* as a scheme. Next, we apply the generic function to compute the ordinary least square and Average based on the defined feature.

```
predictStatistics :: Player → GPF (ScorePrediction,
    ScorePrediction)
predictStatistics p = do
  ds <- loadCsvFromFile "data/basketball.csv"
  (resultAvg,_) :: (ScorePrediction, Mdls Avg) <- genericResult p
    ds
  (resultOls,_) :: (ScorePrediction, Mdls OLS) <- genericResult p
    ds
  return  (resultAvg, resultOls)
```

The actual execution of this code is performed in the following way:

```
main :: IO ()
main = do
    (_avg,_ols) <- DSL.evalIO $ predictStatistics (Player 5.8 160 0
     0 0)
    print $ "Average: " ++ show _avg
    print $ "ols: " ++ show _ols
> main
Average: ScorePrediction {
    fieldGoalsP = 0.45,
    freeThrowsP = 0.74,
    avgPerGameP = 11.79}
ols: ScorePrediction {
    fieldGoalsP = 0.40,
    freeThrowsP = 0.79,
    avgPerGameP = 12.29}
```

2.3 Optimization

The proposed framework currently consists only of simple statistical models. This ensures that the training of the models requires relatively little time to construct. With large datasets and more complex models, it could take weeks before a model is trained. This chapter focusses on the optimization of such models.

2.3.1 Lazy evaluations

Haskell is a non-strict language. This ensures that expressions are not evaluated until they are needed. This has, in addition to advantages, also many disadvantages. Many of these benefits are unfortunately not applicable in the current version of the framework. Reserving these thunks requires a lot of memory for executing Haskell code. These thunks are only evaluated if the argument is needed. To reduce the memory usage, we have to force evaluate the data structures.

The complete evaluation of the structure can be arranged by the package deepseq. This package contains various functions for evaluating. For evaluating

an ADT to normal form we need an instance of the NFData type class. With more complex structures (eg. tree structures) this can give a performance boost. However, laziness makes Haskell not the most suitable language for carrying out these operations. It might be better to work with stricter languages. In addition to performance, laziness also offers less control over how expressions are evaluated. This control is necessary for measuring the time required for computing operations. We, therefore, added as a constraint the NFData to the following constructors of the Instruction:

1. Train
2. TrainWithFilter
3. Predict

Each model datatype requires, therefore, an instance of the NFData. Fortunately, this type class has an instance of Generic. This makes it easy to create an instance for the NFData type class.

The following two remaining parts have not been added to the library, but serve as a proof of concept.

2.3.2 Online learning

If a lot of new data points tend to be uncorrelated, it is likely that the models become less accurate. This can be improved by retraining the model. In the case of batch algorithms, the entire data set plus the extra collected data will have to be trained. Training models can take a lot of time. It could potentially take weeks for certain complex models to be trained. A more efficient option is to use online learning algorithms. These type of algorithms stream incoming data points directly to the model and update the model without having to look at the original dataset. Suppose we use the average as a model. Then we want to update the average with an online learning algorithm. First, the average is represented in the following way:

```
data Avg dp = Avg {
  _avg :: dp,
  _length :: dp
} deriving (Show, Read, Eq, Generic)
onlineTrain :: Avg dp → dp → Avg dp
```

The *_avg* argument contains the current average and the *_length* contains the number of rows provided as input. This information is enough to calculate the new average. An example of the *onlineTrain* function is shown in the following code:

```
> onlineTrain (Avg 24 5) 3
Avg 20.5 6
```

We have to modify various components of the framework to support online learning algorithms. First of all, the interface will be adjusted such that it is

able to merge the same type of models. This allows the interface to combine the above example of *(Avg 45 5)* and *(Avg 3 1)* to *(Avg 20.5 6)*. We solved this by adding as a constraint that all of the *Model* should consist of the Monoid instance:

```
class Monoid (model datapoint) => Interface model datapoint where
  mkTrain :: TrainingStructure datapoint → model datapoint
  mkPredict :: PredictStructure model datapoint → datapoint
```

This gives the advantage that we can automatically derive online algorithms from batch algorithms:

```
mkTrainOnline :: (Monoid (model datapoint),Interface model
    datapoint)
    => TrainingStructure datapoint → model datapoint → model
    datapoint
mkTrainOnline ts me = mkTrain ts <> m
```

Next, we have to adjust all the current models such that they comply with the new interface. For example, the average would have the following Monoid instance:

```
instance Fractional a => Monoid (Avg a) where
  (Avg a b) `mappend` (Avg a' b') =
    let total' = (a * b) + (a' * b')
        length' = b + b'
    in (Avg (total' / length') length')
```

This allows us to give the following instance of the *Interface*:

```
instance Fractional a => Interface Avg a where
    mkTrain ms =
      let values = yData ms
          length'= fromIntegral $ Vector.length values
          avg' = Vector.sum values / length'
      in Avg avg' length'
    mkPredict = _avg . model
```

This instance can then be used for both batch and online learning algorithms.

```
TrainOnline :: (Interface model datapoint,ToJSON (model datapoint),
                NFData (model datapoint))
        => MetaInfo
        → Dataset schema
        → Feature schema datapoint
        → (schema → datapoint)
        → (model datapoint)
        → (Model schema datapoint model → next)
        → Instruction next
```

Finally, to support the online learning algorithms, all models will have to change into a form, from which the original value can be retrieved from the model.

However, as stated at the beginning. We chose not to support this functionality because the current interface is mainly intended for supporting simple

models. Training these models do not take much time, and it is only annoying for the users to give an extra Monoid implementation.

2.3.3 Parallel training

The current version of the framework only supports the usage of 1 core for training a model. With large datasets or/and complex models, training can take a long time. It can, therefore, be better to spread the work across multiple cores. This section covers the modification we have to apply to the framework to support parallel training. However, the final version of the framework does not support this feature. These modifications will be applied under the assumption that the framework has support for online learning algorithms. This makes it easier for merging different models into one model. The idea is that we divide the dataset into X parts, and then we train each of these parts in parallel. The parallel training is performed by the following function *parallelTraining*:

```
parallelTraining ::
  (Interface model datapoint,
  Monoid (model datapoint),
  (NFData (model datapoint)))
  => Int → TrainingStructure datapoint → model datapoint
parallelTraining n (TrainingStructure x y) =
  let _intervals = intervals 0 ((Matrix.nrows x) `div` n) (Matrix.
    nrows x)
      xVars = map (splitMatrix (Matrix.ncols x) x) _intervals
      yVars = foldr splitVectors (y,[]) _intervals
  in fold $ parMap rdeepseq mkTrain
          $ map (uncurry TrainingStructure) $ zip xVars $ snd yVars

intervals :: Int → Int → Int → [(Int,Int)]
intervals a step max'
  | a <= max' = (a,a+step): intervals (a+step) step max'
  | otherwise = [(a,max')]

splitVectors ::
  (Int,Int)
  → (Vector.Vector a, [Vector.Vector a])
  → (Vector.Vector a, [Vector.Vector a])
splitVectors (_min,_max) (_residual, xs) =
  let (a,b) = Vector.splitAt _max _residual
  in (b,a : xs)

splitMatrix ::
  Int
  → Matrix.Matrix a
  → (Int,Int)
  → Matrix.Matrix a
splitMatrix _ncols ms (_min,_max) = Matrix.submatrix _min _max 0
    _ncols ms
```

We use parMap for the parallelization of the structures. This is the parallel version of the map function. This function is from the parallel package.

2.3.4 Error prevention

Making mistakes leads to unexpected behavior. This section describes the measures that have been taken to prevent errors as much as possible that are related to this library. For the underlying code of the framework, we added various unit tests. These are described in the next section.

Different parts of the framework could go wrong. The following points provide an overview of what could go wrong:

1. Provide wrong structure of the database (When you read it from the file system)
2. Provide wrong structure of the model(When you read it from the file system)
3. Provide an invalid path of either a database or model
4. Evaluate IO operation using a non-IO evaluator
5. The supplied dataset does not meet the condition of the model.
6. The supplied row(independent attributes) does not meet the condition of the model
7. Additional rules of the models should be followed
8. Wrong feature

Many of the above problems are caused by side effects. These problems are often difficult to solve during compile time. However, these type of problem can often be dealt with at runtime. The first four points are automatically caught at runtime by the current framework. Loading various files into the environment is managed by the readFile function. This function throws an error when it receives an invalid input. The remaining problems are not caught by the current design. Point three is determined by the evaluators. The remaining points are not automatically caught by the framework. These points depend on the implementation of the models. The framework supports the implementation of an arbitrary amount of models. Each model has it owns conditions that it has to satisfy. For example, the average could only allow a non-empty Matrix. Such a condition is not known in advance. We, therefore, extended the interface with the following two functions:

```
1 class Interface model datapoint where
2   mkTrain :: TrainingStructure datapoint → model datapoint
3   mkPredict :: PredictStructure model datapoint → datapoint
4
5   trainConstraint :: TrainingStructure datapoint → model datapoint
6   trainConstraint = mkTrain
7
8   predictConstraint :: PredictStructure model datapoint →
      datapoint
9   predictConstraint = mkPredict
```

This allows us to check for these conditions. Both functions call the corresponding train or predict functions as the standard implementation, but could be used to define any additional constraint that should be satisfied. Next, we extend the MetaInfo to provide users the option to check whether the provided information of the prediction or training function satisfies the defined constraints:

```
data MetaInfo = MetaInfo{
    pathObject :: Maybe FilePath,
    timingEnabled :: Bool,
    validateCorrectness :: Bool
}
```

Then the evaluators are adjusted such that when the validateCorrectness argument is true, it will call the corresponding constraint function.

2.4 Tests

This section discusses the tests that are defined to validate the correctness of the code. There are several testing methods. We chose to focus on unit testing because this allows us to test independent parts of the code. This has the advantage that when we modify the code, we can verify whether the code works according to the defined behavior. The HSpec library is a testing framework that was used to implement these unit tests. All of the models are validated using fixed values. This includes training and using the model to predict a value. We also validate whether it is possible to write a model to a file and later load the model to the environment. The feature selection is validated whether the selected columns are correctly selected. Furthermore, we added a unit test to validate the evaluators on loading a CSV file.

3 Case study

For the case study, we investigated the resource usage of the jobs at Channable. The result of this chapter is used during the scheduling of the jobs.

3.1 Business Understanding

This phase provides an overview of the hardware usage of the jobs. We have conduct research into the behavior of the jobs at Channable. A job is a process in the operating system. The hardware usage of the jobs depends on the behavior of the job and the environment(eg. parallel jobs) where the job runs. The execution of the jobs is determined by the Jobmachine, which is the scheduler developed by Channable. It is written in Haskell. The scheduler choices between an X number of workers to distribute the jobs. Each of these workers has a capacity of Y slots that can be filled by the jobs. All the workers use Debian X64. The scheduler uses a worker that has enough slots available. The management of all the processes is controlled by the CPU (Central Processing Unit) in the

computer. This component reads the instructions of the process and performs the operations by communicating with the other hardware components. The capacity of these components determines the time needed for performing the operation. Channable consists of seventeen different types of jobs. The behavior of each of these jobs is determined by the Requestmachine. It's written in Python. From the literature, we found that finding similarities in the jobs improves the accuracy of the estimates. We will, therefore, apply the idea of templates[11, 27, 9] to Channable's jobs. The behavior of the job depends on the job type. It has therefore been chosen to subdivide the jobs into seven categories, each of which has its own template.

Group Name	**Description**
API	Jobs to perform API operations
Adwords	Adwords jobs(ie. Google)
Export	Job to export feeds
Import	Jobs to receive new feeds
Shopping	Jobs that deals with the job operations for shopping
Project	Merge the result of each of the jobs.
Analytics	Jobs that deal with analytical operations(eg. google analytics)
No meta information	Internal / less frequent jobs

Table 1: Job type categories

A computer roughly consists of the following components:

1. CPU
2. Storage
3. Network

These components are also used by the workers at Channable during the execution of the jobs. Therefore, we investigated which factors were of importance to the jobs. These factors indicate the dependent variables in the model phase. A dependent variable serves in the statistics as the variable that is estimated. After determining the dependent variable, we investigated which variables could be interesting to make a prediction about the dependent variables. These variables are described as the independent variables.

3.1.1 Dependent variables

For the dependent variables, we examined various hardware component which information affects the resource utilization of the jobs. This information is described in the following sections.

Storage The storage of a computer is divided into the primary storage and secondary storage. The primary storage is also called memory. Operating Systems requires memory to execute a process. The operating system uses virtual memory to manage the memory of these processes. The virtual memory is a memory management system that connects the memory location of a process (virtual address) to the physical addresses. The physical address refers to an address in the RAM or hard disk. The memory of a computer is divided into blocks. Virtual memory allows the operating system to allocate more memory than physically available. It then uses secondary storage to store it. The physical memory stored on the hard disk is called swap space. A hard disk is significantly slower than RAM memory. When the memory of a process exceeds the storage of the swap space, it will be killed by the operating system. In order to avoid the swap space, we chose to the investigate the maximum RSS usage of the processes. The RSS(Resident set size) is the amount of memory allocated for a process in the ram memory.

CPU The CPU fetches instructions from the memory and executes those. It is not relevant to the study how these will be performed. However, an important part of the CPU is how it deals with the virtual memory. Modern CPU's divides the virtual memory into two modes: kernel mode and user mode. Processes that affect the operating system(eg. drivers) are mainly executed in kernel mode. The other processes(eg. applications) are performed in user mode. An operating system distinguishes between these two modes to prevent dangerous operations that can affect the operating system. The time that the processor takes to fully execute a process can be useful when we have to schedule a job. This type of information can, for example, be used as an indicator of the time that we have to take into account the resource usages of another job. Channable already expected that not all the jobs are active the entire time that the job is running because some jobs rely on external tools to respond. This means that the total running time of a job is not a realistic indicator of the actual time a job is busy and uses resources. Therefore, in order to get a better idea of the time that the job spends in non-idle mode, we need to measure the time that the process uses the kernel and user space. The $kernel + user$ represents the time that a job spends in a non-idle mode. The factor to improve the resource utilization is most likely the amount of the time a process takes to be executed and the usage of the process. This study, therefore, chose to collect the following data:

Wall clock time	The total amount of time a job takes to be terminated.
User time	The total amount of time a job spends in the user space.
System time	The total amount of time a job spends in the kernel space.
CPU usage	The percentage of CPU time in non-idle modus.

Table 2: Dependent variables description

Network Nowadays, almost everything goes over the internet. In agreement with Channable, we have, however, decided that a network component is not an interesting resource factor for performing the jobs. Channable has a high network connection, which means we cannot optimize much in network usage. Also, the jobs do not send extremely high amounts of data across the network.

Summary This phase investigated the network, storage and CPU usage of a job. For each component, we discussed which attributes are interesting for the scheduling of a job. We call these the dependent variables. The following table provides an overview of dependent variables:

CPU	wall-clock time
	User time
	system time
	CPU usage
Storage	Maximum resident set size
Network	N/A

Table 3: Dependent variables

The goal of this chapter is to obtain a method that predicts the dependent variables of new incoming jobs. However, when a new job arrives, we have no information about the resource usage. Therefore, we have to make predictions about the dependent variables. The literature has shown that finding similarities in the data improves the prediction accuracy. Therefore, in the next section, we go into more detail in finding attributes that assist in making the data more similar. These attributes are called independent variables.

3.1.2 Independent Variables

This phase provides more details about the attributes that assist in finding similarities in the data. These type of attributes are called independent variables. We investigated this by looking at the behavior of the jobs. There are in total seventeen different types of jobs. The type of the job determines the behavior of the job. We, therefore, used the type of a job as an independent variable. Each of the types consists of its own behavior, however, during the execution of the jobs, many of the jobs also access similar additional independent variables. Channable provides information to the jobs using a database. This type of information affects the behavior of the job and can help to find similarities in the data. Different job types use the same kind of information. Therefore, this analysis chose to categorize the different types of jobs that use the same meta information into six different categories.

Group Name	Jobs
API	clear_api, export_api
Adwords	adwords_generate, adwords_pause_old, adwords_reset, adwords_update
Export	export_feed
Import	download_feed, sequential_download_feed
Shopping	shopping_generate, shopping_reset, shopping_update, show_shopping_factory
Project	merge_imports

Table 4: Job categories

The table above does not show all of the jobs. We have decided to only classify the jobs where we have access to additional meta information about the job. From these jobs, the following attributes were chosen:

Attribute	Description
job type	Type of the job
project id	Reference to the corresponding project
company id	Reference to the corresponding company
#products	The total amount of products that the project contains
#categories	The number of categories that are used to categorize the products
#affected_products	The number of products sent to a particular channel.
#affected_products_initial	The number of products sent to a particular channel of the previous job run.
#rules	Channable provides a rule-based product filtering system. This attribute determines the total amount of rules.
#campaigns	The total amount of campaigns for adwords
#adgroups	The total amount of group for adwords
#keywords	For Adwords jobs, these are the total amount of keywords that will trigger a product when these are inside the query.
#negative_keywords	For Adwords jobs, these are the blacklisted keywords which will not show the product when the keyword is inside the query. These keywords are determined dynamically.
#negative_keywords_static	For Adwords jobs, these are the blacklisted keywords which will not show the product when the keyword is inside the query. These keywords are determined static.
#templates	The total amount of templates for creating new adwords campaigns.
#adgroups_template	The total of templates for creating defining adgroups
#negative_keywords_template	The total amount of templates for defining negative keywords.

Table 5: Independent attributes description

Each category has its own unique subset of attributes. The grouping of the categories is shown in the image below.

Job type	API	adWords	Export	Import	Shopping	Project
total_products	X	X	X	X	X	X
affected_products_initial	X		X	X		X
affected_products	X		X	X		X
nr_rules	X	X	X		X	X
nr_categories	X		X			
campaigns		X				
adgroups		X				
keywords		X				
negative_keywords		X				
negative_keywords_static		X				
templates		X				
adgroups_template					X	
templates					X	

Table 6: Grouping of the categories

3.2 Data Understanding

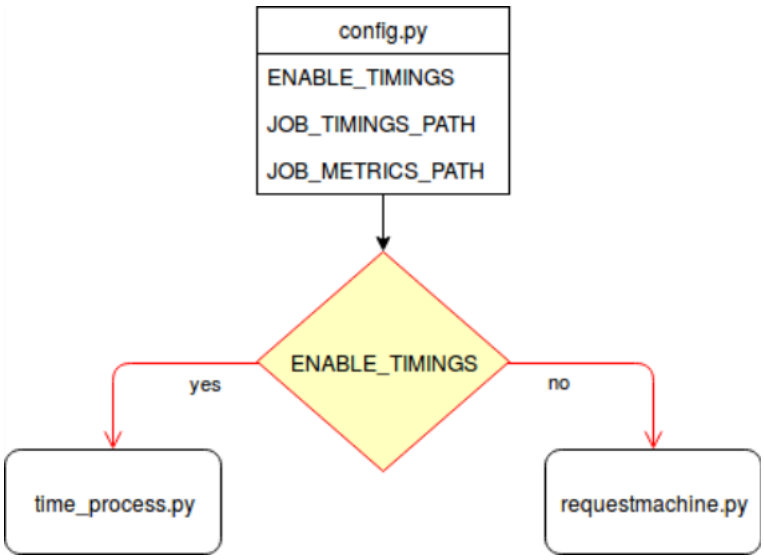
A number of factors have emerged from the previous section that might be useful for improving resource utilization. At this stage, we discuss the strategy we took to gather all of the relevant attributes of the jobs. All of this information is used to construct a historical dataset that contains the resource usage of an X number of jobs. Each of the dependent variables was measured using the built-in command-line tool Time[1] of Ubuntu 16.04. It is chosen due to its ease of use and keeping a record of various relevant information of the resource usage of a job. The tool itself spawns a new child process which executes the command that it is given. In our case, this is the command for executing a job. When the child process terminates, it returns a summary of the various resource usages of the child process. An advantage of this tool is that format of the summary can be defined by the user of the tool and automatically be appended to a file. Which is very convenient for our case to store additional information about the independent variable of the job to the same format and store all of the records to one single file. As a format, we chose JSON(JavaScript Object Notation) to store all of the relevant information of the jobs. The time tool was used to collect the following information:

- Clock time
- User time
- System time
- Percent of CPU this job got
- Maximum Resident Set Size
- Exit status

[1]Not all fields of the time are maintained. This means that some fields always result in the value 0 (e.g., average resident set size). The time used for this thesis can be found in Ubuntu 16.04 under /usr/bin/time.

3.2.1 Collecting of the data

To gather the data, we added an option to the Requestmachine that collects the variables of this chapter when executing a job. Collecting all of the attributes is performed in two steps. First of all, the corresponding independent variables are extracted from a database, except for total_products and affected_products. Then the job is executed as a child process of the time tool. After termination of the child process, it automatically appends the measured information with the corresponding independent attributes to a file. The latter step is performed when the time tool is terminated. This step collects all of the additional attributes total_product and affected_products and appends the result as a JSON format to a different file. Then, both of the files were merged to construct the final dataset. This option is visualized using the following graph:



3.2.2 Insights

This section discusses various insights about the dataset that are obtained by applying several simple statistical models on the dataset. We enabled the measurement of the jobs for 3 days in a row, which resulted in a total of 228k jobs. There were in total measurements of 10 different types of jobs. Figure 2 shows the distribution of each of the jobs over the entire collection.

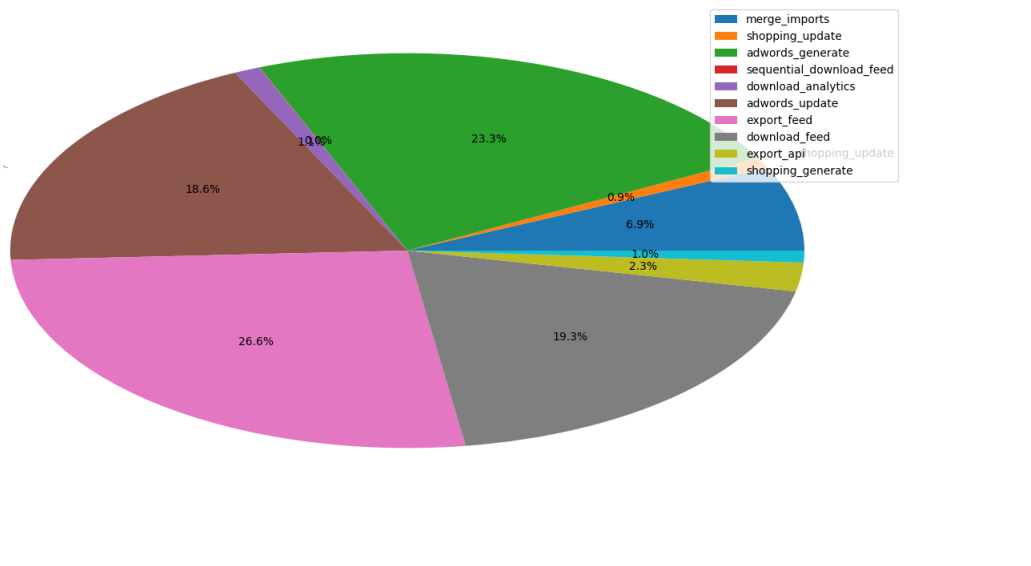


Figure 1: Distribution

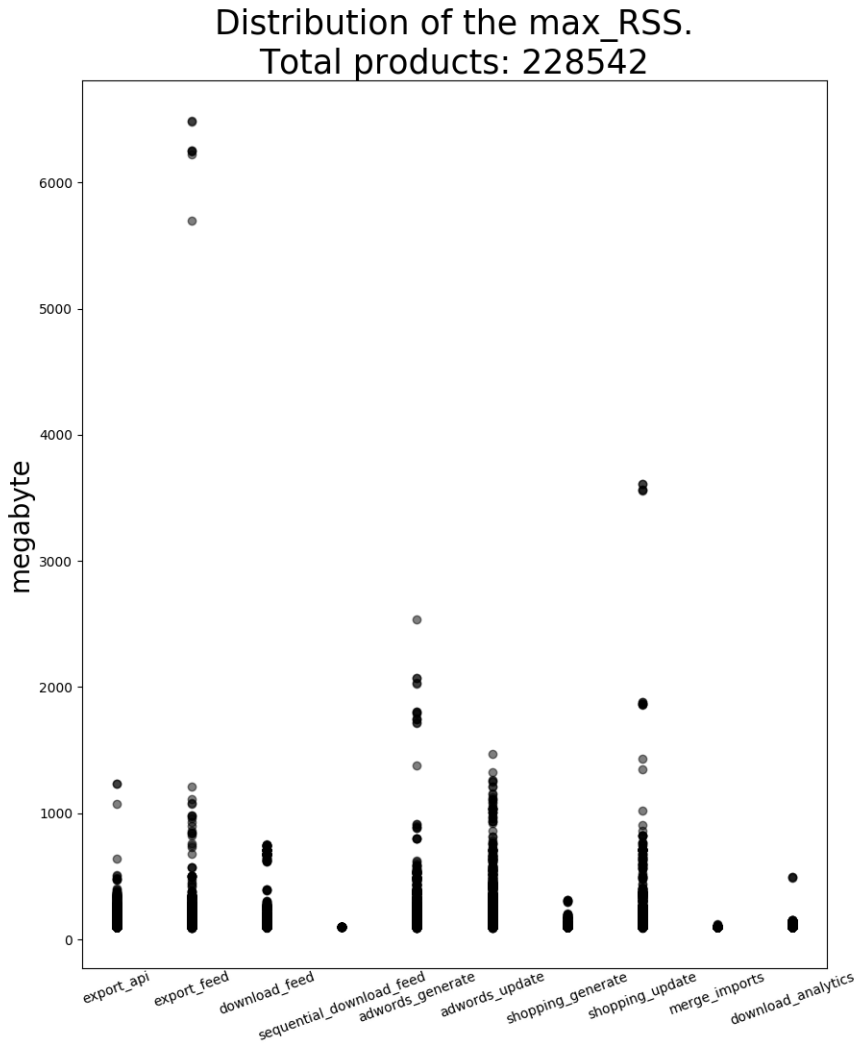
Almost ninety percent of the collected data has been filled by the job types export_feed, download_feed, adwords_update, and adwords_generate. These jobs have the highest probability consisting of a realistic distributed data. To gather more insights about the data, we applied various statistical methods on the dataset and created a scatterplot for each of the dependent variables. The result of the maximum RSS will be discussed in the next section. The insights of remaining dependent variables are added to the appendix.

Maximum RSS For the first insight, we focused on finding patterns in the maximum RSS. On average the maximum RSS of a job is 111.62 megabytes. Each of the jobs consists of its own behavior. We, therefore, looked at the distribution of the maximum RSS per job. We did this by creating a scatterplot that adds a point for each of the jobs in the dataset to the graph, where the x-axis is equal to the job type and the y-axis is the corresponding value of the RSS. Furthermore, the following table shows a number of basic statistics:

job_type	min	max	avg	median	stddev	count
merge_imports	95.87	118.59	96.92	97	0.54	15758
shopping_update	95.14	3613.46	186.38	132	229.93	2041
adwords_generate	91.54	2540.16	102.87	98	39.81	53205
sequential_download_feed	96.64	99.04	97.66	97	0.79	43
download_analytics	95.89	494.20	108.77	99	19.59	2432
adwords_update	94.67	1466.84	129.55	129	44.29	42516
export_feed	92.32	6489.71	115.12	105	71.53	60785
download_feed	91.71	757.34	100.56	98	24.97	44175
export_api	96.32	1233.58	126.57	105	52.90	5365
shopping_generate	96.04	314.84	104.53	99	17.90	2222

Table 7: Maximum RSS - Basic statistics

The average of each of the jobs lays between 96.92MB and 186MB, which is a rather high difference given that the average is 111 megabytes. However, looking at the maximum values, a number of jobs consist of extreme high maximum RSS. For example, a job of the type export_feed has a maximum of 6489 megabytes, which is very high given that the average only consists of 115.12 MB. Such wide range of values makes it hard to make accurate predictions. We, therefore, created a scatterplot that shows the maximum RSS per job type. The x-axis is subdivided into the jobs and the y-axis is divided into the maximum RSS usage.



The scatterplot clearly shows that most of the jobs use less than 1000MB as maximum RSS. However, there are few extreme high outliers. Particularly, the jobs adwords_generate, adwords_update, and shopping_update consist of extreme outliers, which indicates that the data is not a normal distribution. The above table also shows that the maximum RSS of these jobs is much higher than the other jobs. Due to these high maximum values, it was surprising that the average was quite close to each other. The highest average has a value of 186.38MB and the median 132MB, which is a relatively high average given the

median. We, therefore, wanted to investigate the usage further. This allowed us to determine a boundary such that at least 95% of the jobs fit inside this boundary. In total 225636 of the 228542 jobs used less than 200MB RSS. This is $\frac{225636}{228542} = 98.73\%$ of the jobs. This means that without any prediction method, we are able to correctly predict 98.73% of the jobs. The maximum RSS of each of the jobs differs greatly from each other. We, therefore, applied the 200-megabyte limit to each of the individual job types. The result is visible in the next table:

Jobtype	**Total jobs <= 200MB**	**Total jobs >200MB**	**Percentage jobs <= 200MB**	**Total jobs**
merge_imports	15758	0	100%	15758
shopping_update	1650	391	80.84%	2041
adwords_generate	52637	568	98.93%	53205
sequential_download_feed	43	0	100%	43
download_analytics	2429	3	99.88%	2432
adwords_update	41917	599	98.59%	42516
export_feed	60074	711	98.83%	60785
download_feed	44006	169	99.62%	44175
export_api	4908	457	91.48%	5365
shopping_generate	2214	8	99.64%	2222

Table 8: Maximum RSS

The percentage column gives a solid indication of how well the jobs fit inside the dataset based on the 200MB boundary. Almost all of the individual job types consist of a percentage higher or equal to 98.59%. Except for the shopping_update and export_api. We, therefore, determined the maximum RSS boundary of these job types for obtaining the percentage of 95% or higher. Shopping_update requires a limit of 390 MB and export_api 230 MB. The results of the other dependent variables are added to the appendix.

3.3 Data Preparation

This section discusses the approach we took to utilize the construction of the final dataset. Most of the raw data was already suitable to our needs, but we applied a few modification to the original raw data. These modifications are described in the first subsection. The follow-up section provides details about the relevant attribute selection for the modeling phase.

3.3.1 Feature selection

Feature selection is an important approach for finding interesting patterns in the provided dataset. Let us say we have a dataset that consists of the attributes: id, name, gender, and hobbies of a person. Our goal is to predict the gender of a person. Using prior knowledge, we think that it is likely that the name and

hobbies could improve the prediction accuracy. However, it's not very likely that the id assists in improving the accuracy. We, therefore, are only interested in the attributes name and hobbies to predict the gender of a person. Predicting the resource usage of a job had a similar problem. Fortunately, it turns out that for this problem the amount of attributes is relatively small. There are in total 21 different attributes, but not all of the attribute are related to each other. We determined per job, which attribute could improve the prediction accuracy. This was shown in the previous section Independent Variables.

3.3.2 Database setup

The accumulated dataset of the previous phases is divided into two parts. This allows us to determine how well a model performs. The initial part(80%) of the dataset is used to train the models. This sample is randomly selected without the replacement of each of the rows. The remaining (20%) is to validate how well the trained models perform in a theoretical scheduling simulation. This result is validated in the next chapter.

3.4 Modeling

This section covers all of the models that determine the resource usage of new incoming jobs. From the insights of the previous phase, we also determined fixed boundaries for the resource usages such that at least 95% of the jobs are below this boundary. The disadvantage of a fixed limit is that this value does not depend on the supplied data. We could be unlucky in choosing a sequence of jobs that consist of high resource usages. This could result in that the workers require more capacity than available. It might be better to include the independent variables while making a prediction. For example, let's say we have two jobs of the same type. The first job delivers 1 million products to a particular channel and the seconds delivers 1 product to the same channel. It likely that the initial job requires more resources. Making estimates based on the supplied data can be arranged by supervised learning algorithms. We have therefore chosen to focus on regression algorithms. This is a subpart of supervised learning algorithms. The advantage of a regression problem is that the outcome is a real value. Another component of the supervised learning algorithm is the classification of new incoming data points. This has the similarity with a regression problem that the outcome of the model depends on the supplied data. Classification algorithm work mainly with a fixed number of categories. A finite number of categories has the disadvantage that it is hard to take into account huge outliers. Furthermore, these types of models are often more complex and often takes more time to calculate. We do not expect that the estimates will become much more accurate. This is also the reason that we only focused on the Ordinary Least Square. This method is described in the section below.

3.4.1 Ordinary least squares

The disadvantage of the average is that it is a fixed number. This makes it hard to take into account the variability of the data. From the insights, we observed that some of the dependent variables had a lot of variation in the data. Linear Regression models the relationship between different variables. The aim of Linear Regression is to minimize the difference between the observed value and the predicted value, which is also described as minimizing the sum of squares. This allows us to take into account the variability. The predictions are based on multiple parameters. This has the advantage that the estimated value of the dependent variable depends on the supplied data. We will, therefore, use the Ordinary Least Square to predict the resource usages.

3.4.2 Cross validation

The constructed dataset consists of fifteen different independent attributes. The relevance of the independent variables depends on the job type. None of the job types uses all 15. Computing the ordinary least square is also not a heavy computation. We, therefore, chose to generate all possible combinations of attributes based on the relevance for a particular job type. This means that we had to train a total of 640 models. Ultimately, the best 45 (job types * labels) models are used to estimate resource usages. Cross-validation is used to determine the 45 best suitable models. This is used to measure how well a model performs in practice. The idea of cross-validation is to divide the dataset into k parts. Then each run it trains one of the parts and validates this model against the remaining (k-1)parts. This process will be repeated until all of the individual parts have been trained once.

3.4.3 Root Mean Squared Error

The purpose of the cross-validation is to determine the best suitable models per job type for each of the dependent variables. We use the RMSE(Root Mean Squared Error) as the validation measure. This means that we calculate the average of the total difference between the predicted value and the observed value. The model with the lowest RMSE score was chosen as the most suitable model. Formally, this validation technique is written down using the following formula:

$$MSE = \frac{1}{n}\Sigma(\hat{Y}_i - Y_i)$$

3.5 Evaluation

The 45 best OLS models were determined in the previous phases. The result is shown in the appendix. Some of these models result in a rather high RMSE. This indicates that the observed value differs a lot from the predicted value. The theoretical simulation cannot cope with the need for more resources than available in a worker. This would indicate that the jobs require more time to

finish than the observed time. To prevent this, we use a different prediction model when the RMSE is relatively high. In this case, we chose to use the average plus the standard deviation to take into account a buffer. This approach could be too safe, but it keeps the simulation fair. A too high RMSE is a vague concept, but this depends on the variables. For the maximum RSS, 10 is not a high value, but for system time 10 is a high value. Determining these values is not discussed because there are too many models to discuss.

4 Result

This section compares the original scheduling algorithm with the new proposed algorithm. The proposed algorithm uses the estimated workload of incoming jobs to validate whether the worker has sufficient resource capacity to execute the job. In the case there is sufficient resource capacity, we send the job to the worker. However, when the remaining capacity is insufficient, we wait till the worker has enough resource capacity with respect to the estimated workload. The original algorithm validates this based on the available slots.

The jobs at Channable produce various side effects. This makes it difficult to execute a job twice. Let's say we have two jobs. The first job deletes a few products at a particular channel. The next adds the same products to that channel. Re-executing the first job results in fewer products than specified at the settings of the customer. This kind of behavior is not desirable. The result is, therefore, based on a theoretical simulation. Both of the algorithms will be compared through a simulation that should produce results as close as possible to the results from the workers running the actual jobs. The drafted simulation, unfortunately, has several limitations. These limitations are discussed in the next section.

4.1 Constraint

The jobs are only allowed to be executed once due to the side effects that a job produces. This currently makes it impossible to have a precise scheduling simulation. This section describes per dependent variable which constraints we have added to the simulation.

4.1.1 Maximum RSS

The maximum RSS provides misleading information about the RSS usage in general. Let's say we keep track of a particular process. The maximum RSS is 2GB. However, the average is 50MB. This is a huge difference. Choosing the maximum RSS is, however, a safe option. The actual RSS is difficult to obtain. This depends on many different factors, such as the behavior of a process. These types of factors are difficult to take into account while planning a job. It should be emphasized that the maximum RSS is perhaps a too safe option and that in practice probably a less safer option will result in a better utilization. For this

theoretical simulation, we used the maximum RSS to minimize the chance that a process must use the swap space to be fully executed.

4.1.2 CPU usage

The second attribute that is used in this simulation is the percentage that a process spends in non-idle modes. This is measured by the (user_time + system_time) / wall_clock_time. However, this dependent variable also consists of various constraints when it's used in the theoretical simulation. We don't know the exact moments that a process is in non-idle modus. We have therefore chosen to further simplify the simulation. We did this by adding the assumption that we do not take into account the variations of the exact moments. Such variations occur, for example, when several processes at the beginning are in idle modus. This type of information is virtually impossible to include in a theoretical simulation. The insights have shown that this happens frequently at Channable. Subsequently, if a lot of these processes simultaneously change their modus to non-idle, it would probably require the workers to use more capacity than available. This extends the wall-clock time of the processes. The result may, therefore, deviate from the truth.

4.1.3 Wall-clock time

The predicted wall-clock time of a job could be used to further optimize the utilization. For example, it may be of interest to assign a higher probability to jobs that take less time to terminate. This could possibly improve the utilization. However, we decided to not use the predicted wall-clock time, because it is hard to simulate. The observed wall-clock time is used to determine whether a job is finished.

4.1.4 User time and System time

The user time and system time can potentially be used to optimize the resource utilization by introducing job priorities. Unfortunately, these attributes are quite difficult to validate in a theoretical simulation. Prioritizing the jobs would probably result in a change in the order of the jobs. As a result, it can be stated with a lesser probability that the observed wall-clock time remains the same. These two variables are not used in the simulation.

4.2 Scheduling simulation

The simulation is developed in Haskell. The choice is based on the GPF framework, such that it can be used to make estimates. This also has the advantage that it uses the same language as the Jobmachine. We have developed the following data type for representing a worker:

```
1 data WorkerD = WorkerD{
2     jobs :: !(Int, [(UTCTime, (Job,Job))]),
3     logic :: (Job,Job) → WorkerD → Bool,
```

```
    workersCapacity :: !(Capacity, Capacity),
    statistics :: ![(UTCTime, (Capacity, Capacity))]
}

data Capacity = Capacity {
    ram :: !Double,
    cpu_power :: !Double,
    total_jobs :: !Int
} deriving (Eq, Show)

data Job = Job{
    cpu_usage :: Double,
    max_RSS :: Double,
    real_time :: Double,
    user_time :: Double,
    system_time :: Double
} deriving Show
```

The *WorkerD* data type consists of two different types of jobs. The different job types are needed to demonstrate the empirical evidence between the different jobs. Both of the jobs types also consist of their own corresponding capacity. These are updated when the number of jobs in the worker is modified. The number of jobs that can be executed simultaneously depends on the defined property of the logic argument. When adding a new incoming job, it will be validated whether the worker meets the requirements of this property. As long as the worker does not meet the requirements it will keep deleting jobs. When the jobs require more resources than available in the worker, it will throw an error message. Furthermore, the changed capacity is added to the fourth argument. This collection will be compared with each other to determine the empirical evidence. The results of the experiments will be displayed in a line graph.

4.3 Experiment

The originally proposed scheduling algorithm is based on a fixed number of slots available in the worker for executing the jobs. During this experiment, we decided to reserve 13 slots for the execution of the jobs. This number maximizes resource usages without going too much over the capacity. Such an experiment is more likely to have a similar outcome in the real world. The proposed scheduling algorithm uses the estimated resource usages for scheduling a job. We added the code for obtaining the estimates of the resource usage to the appendix. The GPF framework was used to make these estimates.

For the experiment, we used a worker that consist of 8 cores and 8GB ram. During the experiment, the resource usage of external processes was not included (eg. os services, etc). The next section evaluates the result.

4.4 Evaluation

The result of this experiment is evaluated on the RSS, CPU, and utilization of the jobs. Each of these components is evaluated using a line graph that visualizes the empirical evidence. The red line represents the old situation and the green

being the new one. The results are discussed in the following three paragraphs. The x-axis of the first two graphs refers to the number of occurrences of the jobs.

4.4.1 CPU

The graph below uses as y-axis the remaining CPU capacity of the worker. This means that below zero the worker requires more CPU capacity to perform the jobs that are currently running in the worker. Both situations have a large amount of variation in the CPU capacity. The graph shows that the new situation tends to be too cautious in the estimates of the CPU usage. This ensures that it does not need extra CPU capacity to carry out the jobs. However, it is likely that we could obtain a higher utilization by reducing the value of the predicted CPU usage. The original approach uses a few times more CPU capacity than available. This indicates that the original situation would require more wall-clock time to perform the job.

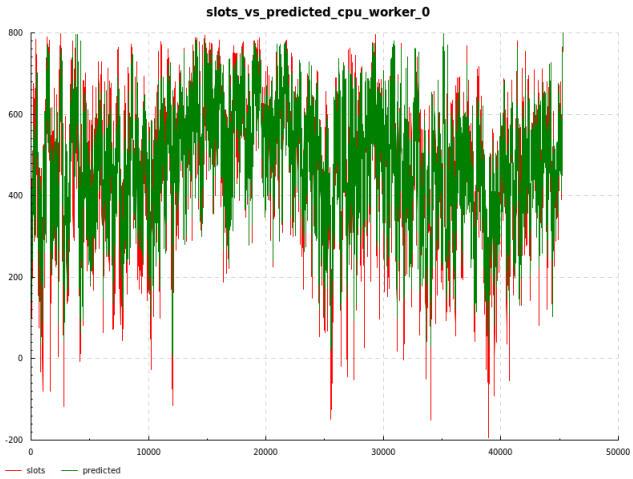


Figure 2: cpu result

4.4.2 RSS

The y-axis describes the remaining RSS of the worker. With a negative RSS, an Operating System would automatically use the designated swap space. This greatly reduces the performance. For both situations, we have tried to minimize this as far as possible. There are not many differences between the original situation and the new situation. Both of algorithms consists of two outliers that would possibly require the use of the swap space to carry out these jobs. Because of these minimal occurrences, we do not think it has any major consequences to the utilization.

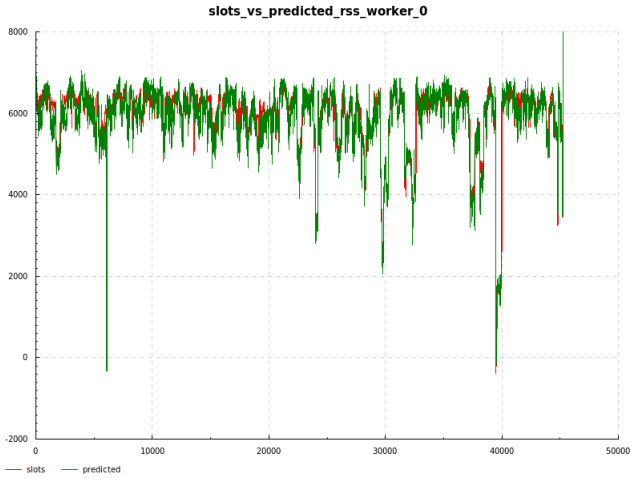


Figure 3: rss result

4.4.3 Utilization

This section compares the utilization based on the above results. It turns out that the new situation improves the utilization. This is rather surprising given that the above graphs indicate that the original approach uses several times more CPU than available. The x-axis shows the time of when the jobs were being executed at the worker. The image shows that this is a period from 7 December to 9 December. The y-axis shows the number of jobs that are executed at a particular time. The difference in time is shown in the following table:

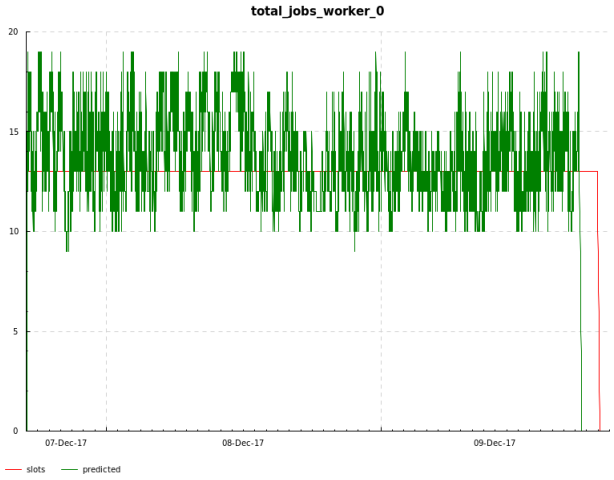


Figure 4: Utilization

Algorithm	Time
Original	50:03:53
New	48:29:47

Table 9: Completion time

This indicates that the proposed algorithm improves the original algorithm with a percentage of 3.1 based on the theoretical simulation. This is a relatively low difference, but still a surprising outcome when looking at the previous two comparisons. The original approach seems to use more or an equal amount of CPU and RSS. In a realistic situation, this approach would probably delay the executions of the jobs when they used more resources than available. The improvement is therefore expected to be higher than 3.1 percent. Due to the variations in the CPU usage, we cannot state with certainty that this estimated result corresponds to the observed scheduling. However, we don't expect much difference in the variation of the theoretical simulation, because these are mainly caused by external parties. For example, many of the jobs often have to wait for the result of an external party. The result also shows that the proposed approach doesn't maximize the CPU usage. The remaining capacity can be used for dealing with unexpected CPU usage variance. We, therefore, expect to obtain a similar utilization in reality.

4.4.4 Conclusion

Based on the collected resource information and theoretical simulation, we can state that our proposed algorithm brings a positive change to the utilization of the jobs at Channable. Both of the approaches used a few times more RSS than physically available on the worker. Due to the low number of occurrences of this, we do not think that this will affect the result.

The CPU usage is a less secure indicator because it does not take into account the variation of when the process is needed. The experiment shows that the original approach requires various times additional CPU capacity, while the proposed approach never request extra CPU capacity.

The outcome of both of the indicators is surprising given that the utilization also improved. We expect from this result that the proposed method will also improve the actual utilization. Unfortunately, due to the different constraints, we cannot state this with certainty.

5 Related work

The structure of the related work is divided into two components. Both of these components are independent of each other. The first component discusses related work about improving the resource utilization. The latter section discusses the related work for building a generic prediction framework in Haskell.

5.1 Improving resource utilization

This section discusses various related work for predicting the resource consumption of future jobs. Previous work has shown that having knowledge about the resource usage of a job is useful for scheduling future jobs. Richard Gibbons presented a technique in [11] to gather knowledge about the workload of jobs on parallel jobs. This paper studies the log files of three parallel servers and examines how to categorize these jobs into categories. He proposes a historical profiler that stores information about the program usage. This information is used to predict the execution time of future jobs. He then uses the estimates to improve the resource utilization of the servers. The input of the historical profiler determines the accuracy of the estimations. For estimating the executing time, he uses the executable name, the user that executed the process, number of processors, wall clock time and memory usage. For computing the estimated execution time he used statistical methods like mean and confidence interval.

Downey [9] presented a more advanced technique for predicting the execution time of parallel jobs. He did this by looking for similarities in Jobs. For finding these similarities he defined templates that consist of a subset of attributes that could be useful for predicting the resource consumption of a job. A template is a tuple that could, for example, be defined as (username, application name, submission time). When the dataset consists of similar jobs, it applies a statistical method to estimate the execution time of these jobs. The execution time is defined as a confidence interval. The idea of a template is to categorize jobs into several categories. According to their technique when a template of a job falls into the same category, they are considered to be similar. W. Smith et al also uses the idea of templates in [27], but rather in a more advanced manner. Instead of an easy definition or just one definition, they defined a number of templates. Then they applied a search technique that results in the template with the best outcome. The authors applied this technique to four workload records of the Cornel Theory Center. They show that this approach improves the accuracy of the prediction comparing to the ones of Downey[9] and Gibbons[11] when they compared it to the four workloads. The author discusses two algorithms for this search technique to identify the template that performs the best and uses that one to categorize the job. The first technique is a Greedy and the second is a Genetic algorithm. Both of these algorithms are divided into three phases: Initialization, prediction, and incorporation of historical information. According to the authors, it is not trivial to determine how many templates are needed to choose the best one. Too few means they will group unrelated jobs, but too many means that there are too few jobs to make accurate predictions. According to the experiments of this paper is that the Genetic approach performed the best.

All of the previous papers only focused on predicting the wall clock time. The paper[24] extended this to multiple predictions of a job workload. This could aid in a better resource utilization. The authors of this paper performed several experiments for predicting the RSS, CPU time and maximum virtual memory. The purpose of this paper was only to show that it is possible to

estimate multiple attributes of a job using a historical dataset. For this reason, they did not use search technique described in the previous for optimizing the result.

All of the previous work only used either linear regression or the mean to compute the predictions. The authors of [18] use machine learning to classify the workload of a job. Several experiments have been performed to evaluate which algorithm results in the best accuracy. From these experiments, the PQR2 algorithm performed the best. This algorithm combines a number of classifiers. The experiments are performed on two historical datasets that consist of job knowledge.

Not all techniques are directly applicable in the case of Channable. For example, none of the papers make use of additional meta-information about a particular job. Every job at Channable has its own meta information. This kind of information will be used to improve the prediction accuracy.

Research has also shown that by having more control over the required hardware we can reduce the energy usage of the servers. The papers [5, 6, 19, 23, 15, 28] investigated the relationship between the resource utilization and energy usage. In [23] they show that we could save a lot of energy by improving the resource utilization. For example, we can use this information to determine on demand how many workers are needed in a cluster. The papers [15, 28, 8, 6] have investigated the workload of the server in a virtualized server environment, where it is easy to adjust the capacity of the hardware. Finding the right amount of capacity is a tradeoff between efficiency and costs. If the capacity is too low, it takes longer to compute a process. The authors of [28] discussed a technique that automatically allocates resource capacity. This is often based on the available hardware, power-cost and application utilities. The paper[26] investigated the use of estimated workload to improve the resource utilization. The industry is also interested in reducing the energy usage. Google investigated[19, 20] the workload of Google Cloud to reduce the energy usage. They described a method for classifying the workload. This estimated resource usage can then be used to improve the resource utilization, which also results in a better energy usage. The authors of [20] discussed how to derive the models for estimating the workload. They also discuss how to deal with various utilization patterns.

5.2 Generic Prediction Framework

Various literature has been consulted for the development of the library. The library will consist of an EDSL that can be used to structurally implement and apply statistical models. The host language in which the EDSL will work is Haskell. Literature shows that Haskell is very suitable for implementing an EDSL. This is because of Haskell consist of various powerful features (e.g Type classes and Algebraic Data Types). Two papers that make use of these features for creating a DSL are the papers [13] Concurrent Orchestration in Haskell and [16] Paradise: A two-stage DSL embedded in Haskell. The authors of [13] have developed a DSL that was built as a higher layer on the current concurrent/parallel primitives of Haskell. This helps the programmer to use concurrent

programs without having to worry about certain annoying background effects of concurrent programming. The authors of [16] have developed a DSL for defining pricing models. These models are used to value financial products across the security trading division of a bank. Their DSL helps to build the pricing tools as reusable components.

The book [21] discusses two possibilities for training models that can be used to make estimates. The traditional way of training a model is performed by offline supervised algorithms. These types of algorithms convert a set of data to a particular model. The second form is training through online algorithms. In this form, incoming data points are streamed to the existing model. These types of algorithms are often much more complex than standard offline algorithms. However, online algorithms are often more efficient in re-training the model. In practice, offline algorithms are not feasible to use as large data sets.

Machine Learning in the world of functional programming is less mature than in the imperative world. This is visible from the number of available libraries and academic papers. Machine Learning is often used in a higher level imperative languages like R or Python. The disadvantage of higher level languages Is that the programmer has less control over certain operations. This often causes the language to be slower. Therefore, these languages use the services of a lower level language often to make predictions. Using these services requires most of the time an expert level knowledge of that language. Understanding the models is often hampered by the fact that there is no standard interface for implementing models. This problem is also known in the Machine Learning community[30]. Little effort has been made to solve this problem. Due to the many possibilities in Haskell, the authors of the papers [2, 3, 12] performed research in construction a standard interface in Haskell that can be used to develop machine learning models. The papers [2, 3] discusses the semantics of such an interface. Both of the papers use type classes and datatypes to construct the interface.

M.Izbicki published in 2015 the paper [12] "HLearn: A Machine Learning Library for Haskell". This paper discusses the techniques behind the Haskell library HLearn. This library consists of a DSL that could be used for structural defining various machine learning models. It is possible for this library to train datasets through offline as well as online algorithms. Furthermore, training these models can be performed in parallel. Unfortunately, at the time of writing this proposal, the library was deprecated and was no longer maintained. Also, it does not support feature selection and the DSL cannot be used to make multiple estimates without using lets and wheres.

Therefore, we proposed the GPF library that solves this problem. Training a prediction model requires the use of a database. The concept the design and implementation of a database in a functional programming language are discussed in [25]. The authors argue that a functional programming language is suitable for this purpose.

The paper "Data Mining the yeast genome in a lazy functional language"[4] presents a data mining application in Haskell. In this paper, they discuss the pros and cons of a lazy functional language for Data Mining. Laziness causes a lot of problems in controlling the heap space. However, they solved this using

various methods. Reading data required huge amount memory usage. They argue that tools like Happy produce a better performance and a reduction in memory usage.

Another drawback of the current design of the GPF library is the limited use of the type system of Haskell. The paper "Strong types for Relation Databases"[1] addresses a similar problem. This paper exploits the type level of Haskell for defining a strongly typed model of relational databases and operations. They did this by embedding a subset of SQL in Haskell. For dealing with arbitrary-length columns they used heterogeneous lists to indicate on type level what kind of type will be expected of the database fields.

Haskell also consists of various libraries that solved similar problems. The package postgresql-simple[7] is a client library for PostgreSQL. It uses a tuple based approach to determine the types on the type level. Cassava[29] is a library for parsing CSV's. This library lets the programmer define its own datatypes for storing the columns of the CSV. Exploiting the type-level reduces the chance of making errors at run-time. Most of the error will, therefore, already be caught at compile time.

6 Discussion

The current interface mainly focusses on supporting simple models. This can be extended to more complex models. The current form of the interface does not support all of the possible models. Let's say we want to implement one of the classification algorithms. Most of these algorithms require a predefined collection of categories. Such a requirement does not comply with the current interface. Therefore, research will have to be performed in finding other interfaces. Hlearn already offers support for more complex models, but this library is no longer maintained and is not well documented.

Training the more complex models often takes a lot of time. This could be improved by using multiple cores for constructing these models. The optimization section discusses an approach fo parallel training. However, it might be better to use an existing methodology. A known method for parallel training is the map reduce. Map reduce is an approach that divides data into multiple parts, such that we can compute an operation in parallel on each of the different parts. Finally, each of the computations will be reduced to one.

Several other parts of the DSL can be expanded. Currently, the framework only offers support for reading CSV files. We expect that a similar approach as the Cassava library also supports other data formats. Furthermore, expanding the evaluators could be useful. For example, evaluating the DSL in parallel could improve the runtime.

The estimated workload is based on various simple statistical models. Complex models could possibly improve the accuracy. For example, classification algorithms allow us to define various labels. However, due to their complexity, it also takes more time to compute. The proposed framework also does not support the use of classification algorithms.

The number of resource variables can also be expanded with the average RSS and network usage. The former can give a relatively better view of the RSS than only the maximum RSS. Let's say that a job uses for 1 percent of the time 1000MB, and the remaining time 100MB. This indicates that reserving the maximum RSS might be too safe.

Jobs can only be executed once due to the side effects that they cause. This forced us to introduce constraints on the theoretical simulation. Unfortunately, therefore, we cannot state with certainty that the introduced algorithm improves the utilization. Nevertheless, the theoretical simulation shows that the introduced algorithm has a positive effect on the utilization. We, therefore, expect that it also has a positive effect on reality. The actual determination of whether it is improved is a difficult problem. Determining the utilization depends on various factors. For example, the number of jobs may vary per day or the moment of executing a job also makes a lot of difference. Let's say we have to schedule a lot of memory intensive jobs simultaneously. This can lead to that the processes are performed more often in the swap space, which is not fair when this does not happen in the other algorithm. It is therefore difficult to determine which algorithm provides a better utilization. We, therefore, recommend analyzing both situations for a longer period of time.

7 Conclusion

We have developed a generic prediction framework in Haskell. As a proof of concept, we elaborated a case study using this framework. In this case study, we investigated the question: "Can we improve the resource utilization by using estimated resource usage of a job as meta-information for scheduling a Job at Channable". This question has been answered by the following three subquestions.

Is it possible to define a framework in Haskell to implement and use various prediction models? We have shown that the interface of the proposed framework can be used for implementing various models. This interface is focused on models that predict contiguous labels. Currently, the proposed framework consists of a few simple models, but this can be easily extended. The DSL of the framework assists programmers in making predictions using the interface. The examples of DSL shows the use of various prediction models. This DSL is based on the datatype Instruction. The Instruction consists of several constructors to describe the primitives of a prediction program. Each of these constructors is lifted to a Free monad using smart constructors. This allows us to define arbitrary ways to interpret the DSL. The next question uses this framework to predict the resource usages.

Can we predict the resource usage of a job with sufficient accuracy? Most of the OLS models allow us to predict the resource usage with a sufficient accuracy. Unfortunately, a few of the final OLS models resulted in a very high

RMSE. For those models, we used the sum of the average and standard deviation as an estimate for the resource usage. By adding the standard deviation to the average, we build-up a buffer that allows us to estimate the resource usage with a sufficient accuracy. From the graphs of the Result chapter, it clearly shows that this buffer also takes into account higher resource usages.

Can we use the predictions to improve the resource utilization? We showed in our simulation that we could improve resource utilization by 3.1%. This was achieved by using the predictions from our framework about job resource usage to allocate jobs more intelligently and using the available resources more efficiently. Unfortunately, due to a number of limits of the scheduling, we cannot reproduce the scheduling exactly. The result of this simulation could, therefore, deviate from the true value.

References

[1] Joost Visser Alexandra Silva. Strong types for relational databases. 2006.

[2] Lloyd Allison. Type and classes of machine learning and data mining. 2003.

[3] Lloyd Allison. Models for machine learning and data mining in functional programming. 2005.

[4] Ross D. King Amanda Clare. Data mining the yeast genome in a lazy functional language. 2003.

[5] Luiz AndrÃĺ Barroso and Urs HÃűlzle. The case for energy-proportional computing, 2018.

[6] Anton Beloglazov and Rajkumar Buyya. Energy efficient resource management in virtualized cloud data centers, 2010.

[7] Leon P Smith Bryan O'Sullivan. Database.postgresql.simple.

[8] Michael Cardosa, Madhukar R. Korupolu, and Aameek Singh. Shares and utilities based power consolidation in virtualized server environments, 2009.

[9] A.B. Downey. Predicting queue times on space-sharing parallel computers. 1997.

[10] Daniel DÃŋaz. The matrix package.

[11] Richard Gibbons. A historical application profiler for use by parallel schedulers. *Springer, Berlin, Heidelberg*, vol 1291, 1997.

[12] Michael Izbicki. Hlearn: A machine learning library for haskell. 2013.

[13] Trevor Elliott John Launchbury. Concurrent orchestration in haskell. 2010.

[14] Edward A. Kmett. The free package.

[15] Dara Kusic, Jeffrey O. Kephart, James E. Hanson, Nagarajan Kandasamy, and Guofei Jiang. Power and performance management of virtualized computing environments via lookahead control, 2008.

[16] Howard Mansell Lennart Augustsson and Ganesh Sittampalam. Paradise: A two-stage dsl embedded in haskell. 2008.

[17] Roman Leshchinskiy. The vector package.

[18] Andrea Matsunaga and JosÃľ Fortes. On the use of machine learning to predict the time and resources consumed by application. 2010.

[19] Asit K. Mishra, Joseph L. Hellerstein, Walfredo Cirne, and Chita R. Das. Towards characterizing cloud backend workloads: Insights from google compute clusters, 2010.

[20] Ismael Solis Moreno, Peter Garraghan, Paul Townend, and Jie Xu. An approach for characterizing workloads in google cloud to derive realistic resource utilization models, 2013.

[21] Kevin P. Murphy. Machine learning: A probabilistic perspective. 2012.

[22] Bryan O'Sullivan. The aeson package.

[23] Eduardo Pinheiro, Ricardo Bianchini, Enrique V. Carrera, and Taliver Heath. Load balancing and unbalancing for power and performance in cluster based systems, 2001.

[24] Rosario M. Piro, Andrea Guarise, Giuseppe Patania, and Albert Werbrouck. Using historical accounting information to predict the resource usage of grid jobs. *Future Generation Computer Systems*, 25(5):499–510, 2009.

[25] Gary Lindstrom Robert M. keller. Approaching distributed database implementation through functional programming concepts. 1985.

[26] James Smith. Workload classification and software energy measurement for efficient scheduling on private cloud platforms, 2011.

[27] Warren Smith, Ian Foster, and Valerie Taylor. Predicting application run times with historical information. *Journal of Parallel and Distributed Computing*, 64(9):1007–1016, 2004.

[28] Ying Song, Hui Wang, Yaqiong Li, Binquan Feng, and Yuzhong Sun. Multi-tiered on-demand resource scheduling for vm-based data center, 2009.

[29] Johan Tibell. cassava: A csv parsing and encoding library.

[30] Kiri L. Wagstaff. Machine learning that matters. 2012.

8 appendices

8.1 Result

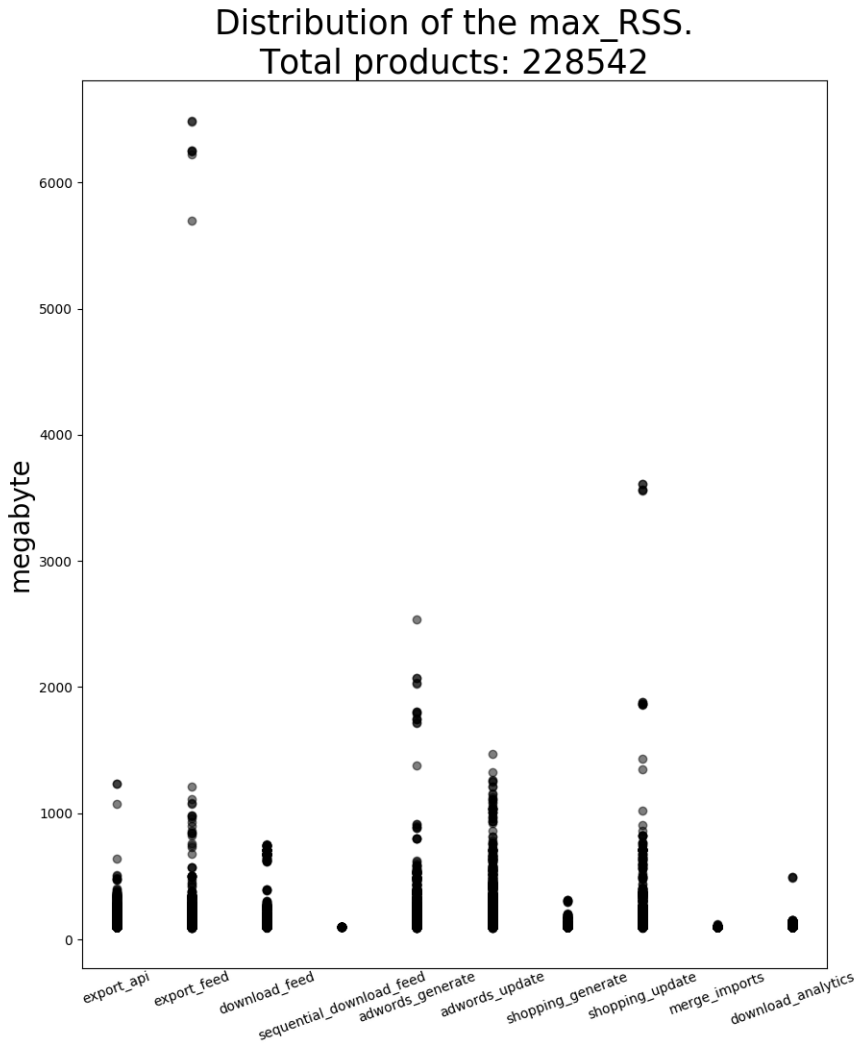
8.1.1 Maximum RSS

Jobtype	min	max	avg	median	stddev	count
merge_imports	95.87	118.59	96.92	97	0.54	15758
shopping_update	95.14	3613.46	186.38	132	229.93	2041
adwords_generate	91.54	2540.16	102.87	98	39.81	53205
sequential_download_feed	96.64	99.04	97.66	97	0.79	43
download_analytics	95.89	494.20	108.77	99	19.59	2432
adwords_update	94.67	1466.84	129.55	129	44.29	42516
export_feed	92.32	6489.71	115.12	105	71.53	60785
download_feed	91.71	757.34	100.56	98	24.97	44175
export_api	96.32	1233.58	126.57	105	52.90	5365
shopping_generate	96.04	314.84	104.53	99	17.90	2222

Table 10: Maximum RSS - Basic statistics

Jobtype	Total jobs <= 200MB	Total jobs >200MB	Percentage jobs <= 200MB	Total jobs
merge_imports	15758	0	100%	15758
shopping_update	1650	391	80.84%	2041
adwords_generate	52637	568	98.93%	53205
sequential_download_feed	43	0	100%	43
download_analytics	2429	3	99.88%	2432
adwords_update	41917	599	98.59%	42516
export_feed	60074	711	98.83%	60785
download_feed	44006	169	99.62%	44175
export_api	4908	457	91.48%	5365
shopping_generate	2214	8	99.64%	2222

Table 11: Maximum RSS



8.1.2 CPU usage

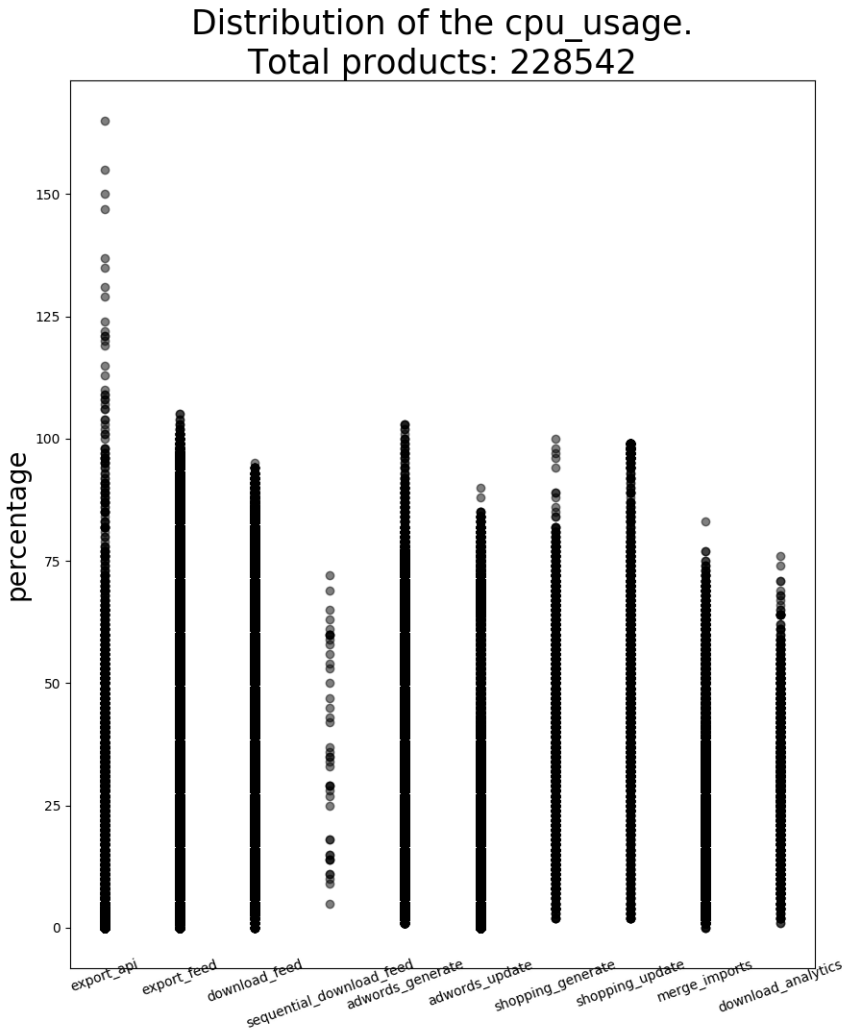
The CPU usage shows the percentage of time that a process spends in non-idle mode, which is the percentage that the process is actually busy. This information could be used to have some intuition about the time that the job is actually busy. Let's say we have a job and as wall-clock time it uses 3600 seconds, but the CPU usage states that it's only active for 0,0001% of the time. This allows us to make more sophisticated discussions about scheduling the job. The average CPU usage of all the jobs is 41%, which indicates that at least

half of the time the jobs are not doing anything. To provide a more detailed statistics about the CPU per job, we added the following table:

Jobtype	min	max	avg	median	stddev	total jobs
merge_imports	0%	83%	23.16%	20.00%	15.01%	15758
shopping_update	2%	99%	52.57%	51.00%	23.23%	2041
adwords_generate	1%	103%	46.13%	50.00%	19.85%	53205
sequential_download_feed	5%	72%	37.40%	35.00%	19.72%	43
download_analytics	1%	76%	30.38%	30.00%	11.73%	2432
adwords_update	0%	90%	21.30%	17.00%	19.49%	42516
export_feed	0%	105%	51.19%	51.00%	23.01%	60785
download_feed	0%	95%	45.40%	43.00%	19.93%	44175
export_api	0%	165%	20.96%	16.00%	20.30%	5365
shopping_generate	2%	100%	39.18%	37.00%	18.74%	2222

Table 12: Cpu usage - Basic statistics

The third column(max) shows that a few of the job types used more than 100 percentage 100%. This indicates that the process used more than one core to compute its goal. Although, most of them are still close to 100. Except for the export_api, which used as maximum 165. Additionally, the table also shows that the average is between 21.30 and 52.57 percentage is, which is a pretty big difference. The median and average of each of the individual job types are close to each other, which indicates that the data is evenly divided around the mean of the job. However, the low CPU usage jobs still got our attention. Especially the adwords_update and export_api jobs. The export_api spends on average $\frac{1}{5}$ of its time non-idle. This is quite a low percentage, given that its maximum is 165. It seems like the 165% is just a big outlier of the export_api given that the median is also just 16%. To investigate the outlier problem further, we generated a scatterplot that represents each of the jobs in the dataset with a dot. The x-axis is used to indicate which job type was used and the y-axis is used to show the percentage of CPU usage.



This scatterplot agrees with the intuition we had about the few outliers. The graph also shows that the distribution of the jobs is pretty close to each other.

8.1.3 Wall-clock time

The elapsed wall clock time of a process is the total amount of time that a process is alive in the Operation System. This is the same amount of time that when you spawn a new process, you immediately start a stopwatch and end the stopwatch after the termination of the process. This kind of information is

useful during the scheduling of the job to have some indication of the time we have to take into account the resource usages of the job. The jobs of Channable use on average 60.5 seconds to be terminated. To investigate this further, we applied various simple statistics on the dataset. This resulted in the following table:

Jobtype	min	max	avg	median	stddev	total jobs
merge_imports	3	342	23.19	17	21.97	15758
shopping_update	1	3014	50.94	14	171.80	2041
adwords_generate	2	10363	36.82	7	224.42	53205
sequential_download_feed	3	53	18.60	17	14.33	43
download_analytics	3	1688	59.21	19	156.04	2432
adwords_update	2	14283	154.64	35	468.95	42516
export_feed	2	2547	35.48	17	58.68	60785
download_feed	2	14099	34.36	13	232.31	44175
export_api	3	13055	187.82	73	503.86	5365
shopping_generate	3	179	14.72	10	15.17	2222

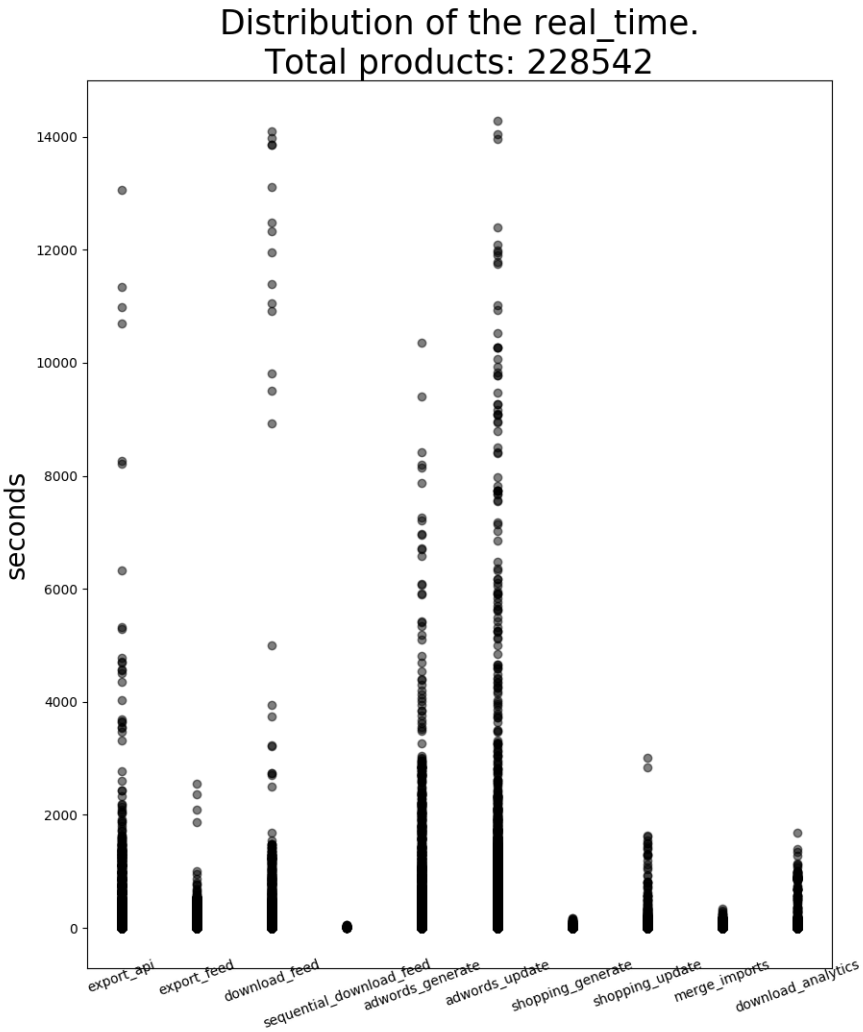
Table 13: Wall-clock time - Basic statistics

This table shows clearly the difference between the average of each of the individual job types. The average is between 14.72s and 187.64s, which is a rather big difference given that the total average is 60.5s. However, when we omit the two highest job types from the interval(export_api, adwords_update), the average of the job types change to between 14.73 and 59.21. Which is rather surprising, given that the total average is lower than the highest average of the latter interval. This shows that these two jobs have a big influence on the average time. Furthermore, the gap between the average and the median of each of the jobs are quite different. This indicates that we consist of various outliers. This could explain the extremely high values of the maximum column. The maximum value is between 53 and 14283. To investigate this further, we generated the following scatterplot:

Jobtype	Total jobs <= 200s	Total jobs >200s	Percentage jobs <= 200s	Total jobs
merge_imports	15748	10	99.93%	15758
shopping_update	1947	94	95.39%	2041
adwords_generate	51921	1284	97.58%	53205
sequential_download_feed	43	0	100%	43
download_analytics	2328	104	95.72%	2432
adwords_update	36173	6343	85.08%	42516
export_feed	59379	1406	97.68%	60785
download_feed	43306	869	98.03%	44175
export_api	4318	1047	80.48%	5365
shopping_generate	2222	0	100%	2222

Table 14: wall-clock - boundaries

The first thing that got out attention was the huge gaps between the jobs. This clearly explains the huge differences in the table. Furthermore, this figure shows that not many jobs take more than 2000 seconds. Then, when looking at the table and image, we see that many jobs need less than 200 seconds. The following table shows this:



8.1.4 System time

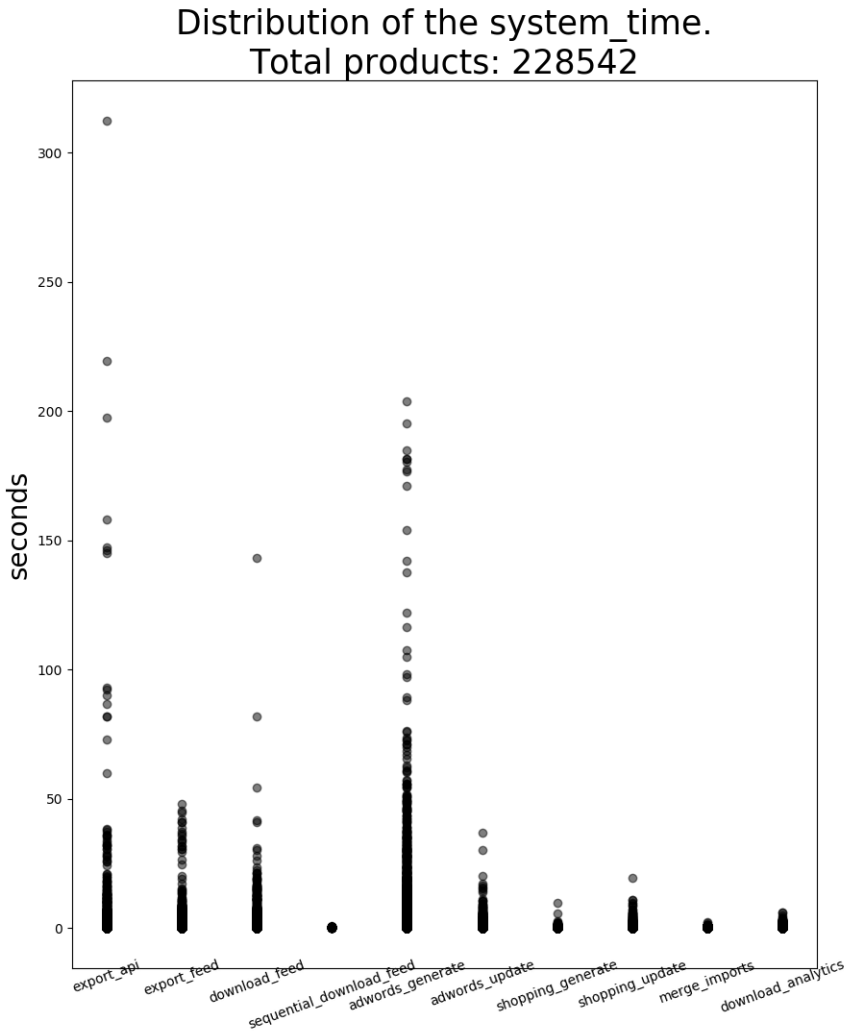
The wall clock time does not provide information about the time that the job spends as non-idle. Let's say that a job takes 3000 wall-clock time seconds. This means we have to take into account all the other resources for this amount of seconds. However, this could be a huge restriction due to the minimum usages of resources most of the time. For example, when the CPU only spends 1 second in non-idle modus, it would make no sense to take into account it for the entire time. We, therefore, were, also interested in the system and user time of a

process. Another advantage of having this kind of information for Channable is that it allows them to optimize their jobs contextually. However, these problems will not be addressed in this thesis. From the wall-clock time, we have seen that adwords_update, download_feed, export_api consists of extreme outliers. Therefore, it is likely that the system time or user time will also consist of the various outliers. The jobs spend on average 0.52 seconds in the system space, which is rather low given that the jobs spend on average 60.5 seconds as elapsed wall-clock time. The following table provides more details about each of the individual job:

Jobtype	min	max	avg	median	stddev	total jobs
merge_imports	0.06	2.16	0.26	0	0.13	15758
shopping_update	0.08	19.56	0.50	0	0.93	2041
adwords_generate	0.06	203.95	0.64	0	3.96	53205
sequential_download_feed	0.11	0.9	0.31	0	0.15	43
download_analytics	0.07	6.08	0.47	0	0.46	2432
adwords_update	0.06	36.9	0.33	0	0.45	42516
export_feed	0.06	48.14	0.58	0	1.01	60785
download_feed	0.07	143.12	0.47	0	1.21	44175
export_api	0.07	312.3	1.41	0	8.24	5365
shopping_generate	0.08	9.73	0.28	0	0.30	2222

Table 15: System time - Basic statistics

The table is drastically different from the wall-clock. None of the statistics have a relationship to these values. Although, It is noteworthy that the jobs that consisted of the outliers in the wall-clock, also have the highest values for the system time. However, it is in nowhere close to the wall-clock. This indicates that the process does not use their resources to their maximum potential or they spent a lot of time in userspace. The next section, therefore, discusses the insights of the user time.



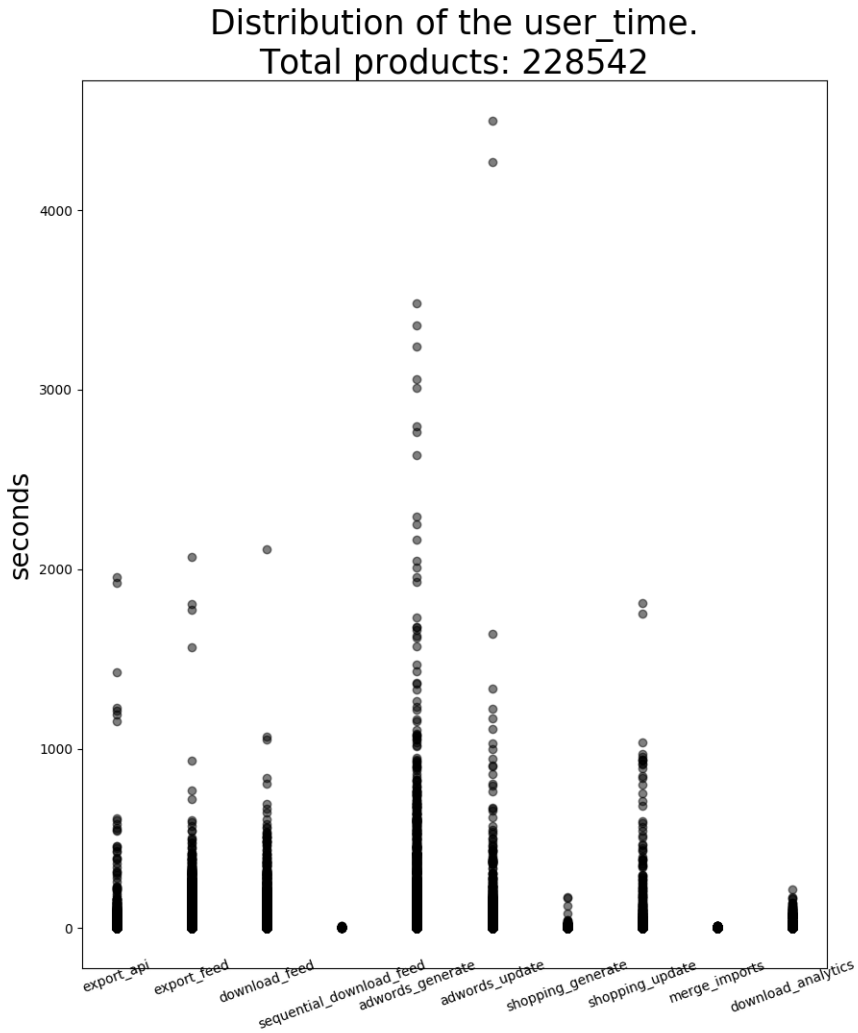
8.1.5 User time

The jobs spend on average 11.95 of their time in userspace. Given the average of the system time(0.52s) and wall clock time(60.5s), we know that on average at least $\frac{3}{4}$ of the time the jobs spend in idle mode. This indicates that the jobs do not fully take advantage of the resources. To investigate this further, we obtained the following statistics:

Jobtype	min	max	avg	median	stddev	total jobs
merge_imports	1.7	14.3	3.27	3	1.39	15758
zshopping_update	1.68	1810.36	27.14	5	106.63	2041
adwords_generate	1.7	3484.37	10.60	3	68.66	53205
sequential_download_feed	1.89	11.04	4.62	4	2.27	43
download_analytics	1.78	217.59	10.95	5	17.97	2432
adwords_update	1.68	4499.1	8.02	5	420.29	42516
export_feed	1.68	2070.66	17.99	6	36.89	60785
download_feed	1.71	2108.83	11.50	5	30.50	44175
export_api	1.81	1956.61	15.06	5	62.68	5365
shopping_generate	1.74	171.47	4.60	3	6.90	2222

Table 16: User time - Basic statistics

The table also shows that the maximum values in the userspace are many times higher than the kernel space. Although, it's by far not close to the maximum jobs of the wall-clock time. However, these high values stand out due to the surprisingly low average values. The median also differs quite a bit. We, therefore, expect that these huge values are due to a few high outliers. The below scatterplot shows, therefore, the distribution of the job.



This clearly agrees with our expectation of having a few outliers. The picture also indicates that the biggest part of the jobs uses less than 200 seconds. Which we could use as a fixed boundary when we cannot determine a prediction model with a sufficient accuracy.

8.2 Using the GPF framework to obtain estimates of the resource usage

```
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE DeriveAnyClass #-}

module Examples.JobAnalys where

import Structure.DSL (
    feature,
    trainFilterWithMetaInfo,
    loadModelFromFile,
    predict,
    allMeta,
    GPF,
    )

import Structure.Interface (Interface)
import Data.Aeson (ToJSON, FromJSON)
import Data.Csv (FromRecord, FromField(..))
import GHC.Generics (Generic)
import Models.Linear (OLS)
import Structure.Dataset (readCSV)
import Data.Vector (filter, Vector, map, zip)
import Data.Map (Map, fromList, keys, empty, insert, member, lookup
    )
import Data.Maybe (fromJust)
import Data.Time (UTCTime, parseTimeM, defaultTimeLocale)
import Control.DeepSeq (NFData)
import Models.Std (Std(..))
import Models.Avg (Avg(..))
import Structure.Evaluators (evalIO, eval)
import Structure.Primitives (Model(..),Dataset(..),MetaInfo(..))
import Prelude hiding (filter, map, zip, lookup)


type JobType = String
-- Reference type for each of the job types
mergeImports, shoppingUpdate, adwordsGenerate,
    download_analytics_label, sequentialDownloadFeed,
    adwordsUpdate, exportFeed, downloadFeed, exportApi,
    shoppingGenerate:: JobType
mergeImports = "merge_imports"
shoppingUpdate = "shopping_update"
adwordsGenerate = "adwords_generate"
sequentialDownloadFeed = "sequential_download_feed"
download_analytics_label = "download_analytics"
adwordsUpdate = "adwords_update"
exportFeed = "export_feed"
downloadFeed = "download_feed"
exportApi = "export_api"
```

```haskell
shoppingGenerate = "shopping_generate"


instance FromRecord Job
instance FromField UTCTime where
    parseField = parseTimeM True defaultTimeLocale "\"%Y-%m-%d %H:%M:%S\"" . show


data Job = Job{
    templates :: !Double,
    systemTime :: !Double,
    cpuUsage :: !Double,
    affectedProductsInitial :: !Double,
    nrRules :: !Double,
    negativeKeywordsStatic :: !Double,
    negativeKeywordsTemplate :: !Double,
    jobType :: JobType,
    nrCategories :: !Double,
    campaigns :: !Double,
    negativeKeywords :: !Double,
    affectedProducts :: !Double,
    jobExecutionDate :: !UTCTime,
    maxRSS :: !Double,
    adgroups :: !Double,
    keywords :: !Double,
    clockTime :: !Double,
    adgroupsTemplate :: !Double,
    userTime :: !Double,
    totalProduct :: !Double
} deriving (Generic, Show, NFData, Eq)

-- Type sysonym to store information about the resources of a job
type Resources a = [Resource a]
data Resource a = Resource{
    _rss :: a,
    _cpuUsage :: a,
    _wallClockTime :: a
}


-- wall-clock time feature for each of the jobs
clockTimeFeatures :: Map JobType [Job -> Double]
clockTimeFeatures = fromList
    [
    (adwordsGenerate, [negativeKeywordsStatic , templates ]),
    (shoppingGenerate, [adgroupsTemplate , negativeKeywordsTemplate , nrRules , totalProduct ]),
    (mergeImports, [affectedProducts ]),
    (downloadFeed,[affectedProducts , totalProduct ]),
    (sequentialDownloadFeed, [affectedProductsInitial , affectedProducts , totalProduct ]),
    (adwordsUpdate, [nrRules , campaigns , adgroups , keywords , negativeKeywordsStatic , templates , totalProduct ]),
    (exportFeed, [affectedProductsInitial , nrRules , totalProduct ]),
    (exportApi, [nrCategories , affectedProducts ]),
```

```
104     (shoppingUpdate, [adgroupsTemplate , totalProduct ])
105     ]
106
107
108 --cpu usage feature for each of the jobs
109 cpuUsageFeatures :: Map JobType [Job -> Double]
110 cpuUsageFeatures = fromList
111     [
112     (adwordsGenerate, [nrRules , campaigns , adgroups ,
        negativeKeywords , negativeKeywordsStatic , templates ,
        totalProduct ]),
113     (shoppingGenerate, [adgroupsTemplate , nrRules , totalProduct
        ]),
114     (mergeImports, [affectedProducts ]),
115     (downloadFeed, [affectedProducts , totalProduct ]),
116     (sequentialDownloadFeed, [affectedProductsInitial ,
        affectedProducts , totalProduct ]), --totalProduct
117     (adwordsUpdate, [nrRules , campaigns , keywords ,
        negativeKeywords , negativeKeywordsStatic , templates ,
        totalProduct ]),
118     (exportFeed, [affectedProductsInitial , nrRules , nrCategories
        , totalProduct ]),
119     (exportApi, [nrCategories , affectedProductsInitial ,
        affectedProducts , nrRules , totalProduct ]),
120     (shoppingUpdate, [adgroupsTemplate , negativeKeywordsTemplate ,
         nrRules , totalProduct ])
121     ]
122
123 --maxRSS feature
124 maxRSSFeatures :: Map JobType [Job -> Double]
125 maxRSSFeatures = fromList [
126     (adwordsGenerate, [adgroups , keywords , negativeKeywords ,
        negativeKeywordsStatic , templates , totalProduct ]),
127     (shoppingGenerate, [negativeKeywordsTemplate , nrRules ,
        totalProduct ]),
128     (mergeImports, [affectedProducts ]),
129     (downloadFeed, [affectedProducts , totalProduct ]),
130     (sequentialDownloadFeed, [affectedProducts ]),
131     (adwordsUpdate, [nrRules , campaigns , adgroups ,
        negativeKeywords , negativeKeywordsStatic , templates ,
        totalProduct ]),
132     (exportFeed, [affectedProductsInitial , nrRules , nrCategories
        , totalProduct ]),
133     (exportApi, [nrCategories , affectedProductsInitial ,
        affectedProducts , nrRules , totalProduct ]),
134     (shoppingUpdate, [adgroupsTemplate , negativeKeywordsTemplate ,
         nrRules , totalProduct ])
135     ]
136
137 ----------------------------
138 -- predicts Map
139 ----------------------------
140 type PredictMap = Job
141     -> (Map JobType (Resource (Mdl OLS))
142     , Map JobType (Resource (Mdl Avg))
143     , Map JobType (Resource (Mdl Std)))
144     -> GPF Double
```

```
145
146 -- This function describes for the wall-clock time how to make the
        prediction.
147 clockTimePredicts :: Map JobType PredictMap
148 clockTimePredicts = fromList
149     [
150     (adwordsGenerate, avgPred adwordsGenerate _wallClockTime),
151     (shoppingGenerate, predLm shoppingGenerate _wallClockTime),
152     (mergeImports, avgPred mergeImports _wallClockTime),
153     (downloadFeed, avgPred downloadFeed _wallClockTime),
154     (sequentialDownloadFeed, predLm sequentialDownloadFeed
        _wallClockTime),
155     (adwordsUpdate, avgPred adwordsUpdate _wallClockTime),
156     (exportFeed, predLm exportFeed _wallClockTime),
157     (exportApi, avgPred exportApi _wallClockTime),
158     (shoppingUpdate, avgPred shoppingUpdate _wallClockTime)
159     ]
160
161
162 -- This function describes for the cpuUsage time how to make the
        prediction.
163 cpuUsagePredicts :: Map JobType PredictMap
164 cpuUsagePredicts = fromList
165     [
166     (adwordsGenerate, avgPred adwordsGenerate _cpuUsage),
167     (shoppingGenerate, avgPred shoppingGenerate _cpuUsage),
168     (mergeImports, avgPred mergeImports _cpuUsage),
169     (downloadFeed, avgPred downloadFeed _cpuUsage),
170     (sequentialDownloadFeed, avgPred sequentialDownloadFeed
        _cpuUsage),
171     (adwordsUpdate, avgPred adwordsUpdate _cpuUsage),
172     (exportFeed, avgPred exportFeed _cpuUsage),
173     (exportApi, avgPred exportApi _cpuUsage),
174     (shoppingUpdate, avgPred shoppingUpdate _cpuUsage)
175     ]
176
177 -- This function describes for the maximal RSS time how to make the
         prediction.
178 maxRSSPredicts :: Map JobType PredictMap
179 maxRSSPredicts = fromList [
180     (adwordsGenerate, avgPred adwordsGenerate _rss),
181     (shoppingGenerate, predLm shoppingGenerate _rss),
182     (mergeImports, predLm mergeImports _rss),
183     (downloadFeed, predLm downloadFeed _rss),
184     (sequentialDownloadFeed, predLm sequentialDownloadFeed _rss),
185     (adwordsUpdate, predLm adwordsUpdate _rss),
186     (exportFeed, avgPred exportFeed _rss),
187     (exportApi, predLm exportApi _rss),
188     (shoppingUpdate, avgPred shoppingUpdate _rss)
189     ]
190
191 ---------------------------------
192 -- end predict maps
193 ---------------------------------
194
195
196 -- Type sysonym to use a model with represent a model with a Double
```

```
197 type Mdl mdl  = Model Job Double mdl
198
199
200 -- All jobtypes
201 jobtypes :: [JobType]
202 jobtypes = keys clockTimeFeatures
203
204
205
206 -- Train all the different models(avg,ols,std) such that we can
        load it later from the filesystem
207 trainAllModels :: FilePath -> IO (Resources (Mdl Avg), Resources (
        Mdl OLS), Resources (Mdl Std))
208 trainAllModels filePath = do
209     ds :: Vector Job <- readCSV filePath
210     x <- trainModel "avg" ds
211     y <- trainModel "lm" ds
212     z <- trainModel "std" ds
213     return (x,y,z)
214
215 -- Train a model and write the model to the filesystem.
216 trainModel ::
217     (Interface mdl Double, ToJSON (mdl Double), NFData (mdl Double)
        )
218     => String
219     -> Vector Job
220     -> IO (Resources (Model Job Double mdl))
221 trainModel suffix ds = foldr (go (trainJobs suffix)) (return [])
        jobtypes
222     where go f job b =
223             (evalIO $ f (Dataset ds) job) >>= (\x -> fmap (x:) b)
224
225
226 -- description of how the job should be trained.
227 trainJobs ::
228     (Interface mdl Double, ToJSON (mdl Double), NFData (mdl Double)
        )
229     => String
230     -> Dataset Job
231     -> JobType
232     -> GPF (Resource (Mdl mdl))
233 trainJobs suffix ds job = do
234     maxRSSFt     <- feature $ unsafeLookup job maxRSSFeatures
235     cpuUsageFt   <- feature $ unsafeLookup job cpuUsageFeatures
236     clockTimeFt   <- feature $ unsafeLookup job clockTimeFeatures
237     mMaxRss     <- trainFilterWithMetaInfo (wrapper job ("maxRSS_"
        ++ suffix)) ds maxRSSFt maxRSS ((==job) . jobType)
238     mCpuUsage   <- trainFilterWithMetaInfo (wrapper job ("cpuUsage_
        " ++ suffix)) ds cpuUsageFt cpuUsage ((==job) . jobType)
239     mclockTime   <- trainFilterWithMetaInfo (wrapper job ("
        clockTime_" ++ suffix)) ds clockTimeFt clockTime ((==job) .
        jobType)
240     return $ Resource mMaxRss mCpuUsage mclockTime
241
242
243 -- Load the models from the filesystem
244 loadJobs ::
```

```
245     (ToJSON (model Double),FromJSON (model Double), Interface model
         Double)
246     => String
247     -> JobType
248     -> GPF (Resource (Model Job Double model))
249 loadJobs suffix job = do
250     maxRSSFt     <- feature $ unsafeLookup job maxRSSFeatures
251     cpuUsageFt   <- feature $ unsafeLookup job cpuUsageFeatures
252     clockTimeFt   <- feature $ unsafeLookup job clockTimeFeatures
253     mMaxRss     <- loadModelFromFile (modelLocation job $ "maxRSS_"
             ++ suffix)  maxRSSFt maxRSS
254     mCpuUsage   <- loadModelFromFile (modelLocation job $ "
        cpuUsage_"   ++ suffix) cpuUsageFt cpuUsage
255     mclockTime   <- loadModelFromFile (modelLocation job $ "
        clockTime_"   ++ suffix) clockTimeFt clockTime
256     return $ Resource mMaxRss mCpuUsage mclockTime
257
258
259 toJobCapacity :: Job -> Resource Double
260 toJobCapacity j = Resource (cpuUsage j) (maxRSS j) (clockTime j)
261
262 -- Load all the models to a map, such that it easy to access each
        of the different models
263 jobsToMaps :: IO (Map JobType (Resource (Mdl OLS)), Map JobType (
        Resource (Mdl Avg)), Map JobType (Resource (Mdl Std)))
264 jobsToMaps = do
265         x <- loadJobsToMap $ loadJobs "lm"
266         y <- loadJobsToMap $ loadJobs "avg"
267         z <- loadJobsToMap $ loadJobs "lm"
268         return (x,y,z)
269
270 -- Load all the model of one type to a map
271 loadJobsToMap :: (JobType -> GPF a) -> IO (Map JobType a)
272 loadJobsToMap mdls = foldr go (return empty) jobtypes
273     where go job env = do
274                             env' <- env
275                             value <- evalIO $ mdls job
276                             return $ insert job value env'
277
278 --Combine the observerd with the predicted job data
279 getJobsCombine :: IO (Vector (Resource Double, Resource Double))
280 getJobsCombine = do
281     js <- readCSV "data/jobs.test.csv"
282     mJobs <- jobsToMaps
283     let js' = filter (\x -> (jobType x) /= "download_analytics") js
284         trueJobs = map toJobCapacity js'
285         predictedJobs = map (eval . predictResource mJobs) js'
286     return $ zip trueJobs predictedJobs
287
288 -- Looks up the correct OLS and uses that model to predict new data
289 predLm :: JobType -> (forall a . Resource a -> a) -> PredictMap
290 predLm name f job (lms,_,_) = predict (f $ unsafeLookup name lms)
        job
291
292 -- Looks up the correct AVG and uses that model to predict new data
293 avgPred :: JobType -> (forall a . Resource a -> a) -> PredictMap
294 avgPred name f _ (_,avgs,stds) =
```

```
295     return $ (_avg $ _modelDef $ f $ unsafeLookup name avgs)
296             + (std $ _modelDef $ f $ unsafeLookup name stds)
297
298
299 -- predict each of the resources using it correct models
300 predictResource ::
301     (Map JobType (Resource (Mdl OLS))
302     ,Map JobType (Resource (Mdl Avg))
303     ,Map JobType (Resource (Mdl Std)))
304     -> Job
305     -> GPF (Resource Double)
306 predictResource env job =
307     let lookup' preds = unsafeLookup (jobType job) preds job env
308     in
309     do
310     rssCapacity' <- lookup' maxRSSPredicts
311     cpuCapacity' <- lookup' cpuUsagePredicts
312     terminationDate' <- lookup' clockTimePredicts
313     return $ Resource cpuCapacity' rssCapacity' terminationDate'
314
315 -- Abbreviation to use all the meta information and provide the
        correct store lcation
316 wrapper :: String -> String -> MetaInfo
317 wrapper job response = allMeta $ modelLocation job response
318
319
320 modelLocation :: String -> String -> String
321 modelLocation job response = "data/models/" ++ job ++ "_" ++
        response ++ ".mdl"
322
323 -- Throws an error when the key does not exist
324 unsafeLookup :: (Show a, Ord a) => a -> Map a t -> t
325 unsafeLookup name env
326     | member name env = fromJust $ lookup name env
327     | otherwise = error $ "The Key ("  ++ show name ++ ") does not
        exist.."
```