Optimizing ARIMA for SystemML

STUDENT RESEARCH PROJECT

from the Course of Applied Computer Science

at the Cooperative State University Baden-Württemberg Mannheim

by

Tobias Schmidt

April 31th, 2019

Time of Project	November 1st 2018 to April 30th 2019
Student ID, Course	4748088, TINF16AI-BC
Supervisor	Reinhold Hübl

Signature Supervisor _____

Contents

Abstract - English

The Autoregressive Integrated Moving Average (ARIMA) model is one of most broadly used time series models and applied to better understand and forecast time series data. As a generalization of the Autoregressive Moving Average (ARMA model, which itself is a combination of the Autoregression and MA model, it can also be used to analyze time series that are non-stationary.

The goal of this research project was to complete the implementation of the ARIMA algorithm for SystemML using the Declarative Machine Learning Language.

This report starts of with a short introduction that gives an example for the applications of large scale time series analysis to demonstrate its importance and then discusses the focus and motivation of the project in more detail. In the second chapter, the ARIMA model is explained and discussed in full with a heavy focus on the systems of linear equations which is at the center of the model. An important part of the algorithm is to find a solution for these linear systems, which is why there is a completely separate section in the theory chapter that discusses different approaches used for solving linear systems and particular the linear systems produced by ARIMA models, which have some key characteristics that can be exploited to find a solution more quickly. To give some more background on the over all training algorithm a number of optimization methods commonly used are outlined and discussed. The third chapter then explains why and which solvers for the system of linear equations were chosen and discusses the implementation of ARIMA in DML. It is also describing how exactly the performance and precision of the script will be measured and how it is tested for correctness. Afterwards, the results are presented and finally the methodology as well as the results are discussed and assessed.

1. Introduction

On the 31st of January 2015 NASA has launched the Soil Moisture Active Passive (SMAP) research satellite to measure land surface soil moisture. The satellite has been established in a near-polar sun-synchronous orbit, repeating its ground track exactly every 8 days. It is equipped with 15.000 sensors that each take an average of 2000 samples per second producing a payload of 4 bytes each. Some of them are used for measuring soil moisture and others are providing vital information to the autopilot or are used to keep track of the health of the satellite and its systems.[11] All in all, the data produced amounts to 10 terabytes *daily*. Over the course of its three-year mission the satellite will gather and return 135 Terabytes of mapping data. And almost all of this data is *time series data*, meaning each sensor value is associated with the time of its measurement.[10] Analyzing this enormous pile of time series data is a great challenge as there is a lack of tools that support both large scale machine learning capabilities as well as machine learning algorithms suited for time series analysis.

There are essentially two parts to the problem that need to be addressed simultaneously. First, we need a way to analyze the time series data itself using appropriate and well-defined models as well as a way to address the sheer amount of data making the training and prediction with these models viable.

The goal of time series analysis on its own is either to understand and interpret the underlying forces that produce the observed system in relation to time or to forecast a future state of the system.[2] This is done by building a model that represents the data as close as possible while also being as simple as possible. And there are several different kinds of models to choose from. The Autoregressive Integrated Moving Average (ARIMA) model is one of them and wildly spread in terms of practical use. This is mostly because of its ability to take into account a reasonable amount of the complexity that time series data sets can embody. This is mostly due to the fact that the ARIMA model is actually a generalization of three of the most common classes of stochastic processes. The class of Autoregression models (AR), of integrated (I) models and of Moving Average (MA) models. And of course there are many other models that could address the first issue that this paper tries to solve. However, because of the scientific work already done on ARIMA and its status as one of the most basic tools for time series analysis in general, this particular model was chosen as the first part of the solution.

Now, if there wasn't the second part of the problem still left, we would already have more than enough tools offering solutions. Unfortunately, we still need to address the second problem. Only on its own we'd again have multiple solutions ready to go. Essentially the way the "large-scale" issue has been addressed is two-fold: On one hand we have solutions for distributed computing and on the other hand we have solutions for distributed file systems. Solutions for distributed computing address questions on how to divide a given computational task between multiple instances in a way, that results in the task being solved quickly while also making sure that the overall throughput of the cluster, when considering multiple tasks, is maximized. Distributed file systems are then needed to allow a cluster consisting out of multiple servers to share a unified file system. Without this, every instance would need a full copy of all the data that this particular instance currently requires. Resulting in a drastic reduction of overall storage volume while also impacting the distributed computing power. Fortunately for us, there already are multiple frameworks that provide these capabilities. One of them is Apache Hadoop which offers a distributed file system and another one is Apache Spark which can be used for distributed computing.

It might look like we'd already be done here, but solving each one of the problems individually is not the real issue. Solving them simultaneously is. This is where Apache SystemML comes into play. Its main goal, however, is to address a much more deeply grounded issue related to implementing machine learning algorithms like ARIMA for large scale systems using Hadoop and Spark. The main issue with that is that data scientists or machine learning engineers, which are meant to develop the machine learning algorithms, are rarely Spark or Hadoop engineers as well and vice versa. To overcome the need of a second engineer that is specialized on cluster computing frameworks like Spark or Hadoop, Apache SystemML has been developed. By providing an R-Like programming language, called Declarative Machine Learning Language (DML), SystemML is able to scale any algorithm based on data and cluster characteristics by automatically compiling the DML scripts into a set of Spark API commands. This makes the need for a separate Hadoop or Spark engineer obsolete and enables the data scientist to use a familiar programming language while also having an implementation that works for big data.

Therefore the time series analysis model ARIMA was implemented for Apache's SystemML using the Declarative Machine Learning Language. Partially this implementation had already been done before this research project had started. But it had only been fully completed for the Autoregression model. Hence, the goal of this research project was to finish the implementation of the full Autoregressive Integrated Moving Average model and optimize it as much as possible in the set amount of time that was available. As will be discussed in detail in the methodology chapter, there are essentially two parts of the training algorithm, that can be optimized independently. The scoring function of ARIMA

and the optimizer used to calculate the best weights for the model. For this project, the scoring function was chosen to be optimized first, because it seemed to have more potential for optimization.

2. Theory

2.1. ARIMA Model

The Autoregressive Integrated Moving Average (ARIMA) model is used to analyze and forecast time series data and is a generalization of the Autoregressive Moving Average (ARMA) model. The ARMA model in turn is a combination of the Autoregression (AR) model and the Moving Average (MA) model.[8]

The key idea behind both the AR and the MA model is to describe each value of the time series by using linear combinations that are based on the previous values of the time series. In the case of Autoregression the previous values for the linear combination are indeed the raw values of the time series. For Moving Average on the other hand the linear combinations are based on the white noise disturbances that are present in the previous predictions. These white noise disturbances are also referred to as residuals and are simply the errors of previous predictions. So essentially an ARMA process tries to describe a given point in a time series by combining previous values and the errors that have been made when trying to predicting these previous values.

However, one restriction of ARMA models is that they can only be used for time series that are *stationary*. Meaning that their statistical properties stay constant over time. To overcome this restriction the ARMA model has been extended by an additional step called integration, which is used to transform non-stationary time series into stationary ones. The process used for these transformations is called differencing and will be discussed later.

The formal definition for each part that makes up the Autoregressive Integrated Moving Average model is going to be discussed in the following subsections. Please note that the implementation of ARIMA as it is discussed in chapter 3 will be slightly modified to match its counterpart implementation in the *stats* library of the language R.

Autoregression

To be able to define the AR process and all the other parts of ARIMA more easily, the backshift or lag operator B is defined as:

Definition 2.1.1 *The backshift operator B operates on an a given element of a time series to produce the previous element:*[8, p. 29]

$$BX_t = X_{t-1} \tag{2.1}$$

Definition 2.1.2 *Based on definition 2.1.1 the power of B is defined as:*

$$\begin{aligned} B^2X_t &= BBX_t &&= BX_{t-1} &&= X_{t-2} \\ B^3X_t &= BB^2X_t &&= BX_{t-2} &&= X_{t-3} \\ &\vdots & & & &\vdots \\ B^iX_t &= BB^{i-1}X_t & & & &= X_{t-i} \end{aligned} \tag{2.2}$$

Using the backshift operator the AR(p) process can be defined as the linear combinations of the p previous elements of the time series:

Definition 2.1.3 *Given a univariate time series X_t that is stationary with a constant mean of 0 and e_t a random variable that is independent and identically distributed with mean 0 and variance σ^2 representing the error that can occur in any prediction, the AR(p) model is defined by:*[8, p. 17]

$$\begin{aligned} X_t &= e_t + \sum_{i=1}^{p}(\phi_i B^i)X_t \\ &= e_t + \phi_1 X_{t-1} + \phi_2 X_{t-2} + ... + \phi_p X_{t-p} \end{aligned} \tag{2.3}$$

with $\phi_1, ..., \phi_p$ being the parameters of the model.

By defining the error of a prediction as the difference between the correct value from the time series and the value from the approximation $\hat{X}_t$ as:

$$e_t = X_t - \hat{X}_t \tag{2.4}$$

The AR(p) process can therefore be used to calculate an approximation or prediction for X with:

$$\hat{X}_t = \sum_{i=1}^{p} (\phi_i B^i) X_t \tag{2.5}$$

$$displaystyle \sum_{i=1}^{N} (\phi_i B^i) X_t \tag{2.6}$$

Moving Average

The Moving Average model is a linear combination of the error terms e_t as defined by equation 2.4. It's formal definition is:

Definition 2.1.4 *Given a univariate and stationary time series X_t with a constant mean of 0 and the error term $e_t \sim iid(0, \sigma^2)$ the MA(q) model is defined by:*[8, p. 50]

$$\begin{aligned} X_t &= e_t + \sum_{m=1}^{q} (\theta_m B^m) e_t \\ &= e_t + \theta_1 e_{t-1} + \theta_2 e_{t-2} + ... + \theta_q e_{t-q} \end{aligned} \tag{2.7}$$

with $\theta_1, ..., \theta_q$ being the parameters of the model.

Even though the definition of the AR(p) model and MA(q) model look similar, there is an important difference which makes forecasting with an Moving Average model harder in comparison to an Autoregression model: When calculating an AR model given the model's weights $\phi_1, ..., \phi_p$ and the time series X, the approximation $\hat{X}_t$ can be calculated in one step. This is because all the variables in equation 2.5 are already known. However, when calculating the approximation $\hat{X}_t$ for a MA model with:

$$\begin{aligned} \hat{X}_t &= \sum_{m=1}^{q} (\theta_m B^m) e_t \\ &= \theta_1 e_{t-1} + \theta_2 e_{t-2} + ... + \theta_q e_{t-q} \end{aligned} \tag{2.8}$$

The exact values of the error terms from e_{t-1} up to e_{t-q} are not known and need to be calculated first. But to calculate $e_{t-1} = X_{t-1} - \hat{X}_{t-1}$ the approximation of X_{t-1} is also unknown and needs to be calculated first. Which in turn is dependent on the values for the error terms from e_{t-2} up to e_{t-q-1}. And this goes on for $\hat{X}_{t-2}$, which will need all the values for the terms from e_{t-3} up to e_{t-q-2}. This pattern continues for all $\hat{X}_t$ up to the

last one being $\hat{X}_0$ and eventually the values for all error terms from e_{t-1} to e_1 need to be calculated.

As an example, consider the MA(3) model's approximation for $\hat{X}_t$:

$$\begin{aligned} \hat{X}_t &= \theta_1 e_{t-1} + \theta_2 e_{t-2} + \theta_3 e_{t-3} \\ &= \theta_1(X_{t-1} - \hat{X}_{t-1}) + \theta_2(X_{t-2} - \hat{X}_{t-2}) + \theta_3(X_{t-3} - \hat{X}_{t-3}) \end{aligned} \tag{2.9}$$

Assuming that the time series X is of length 4, the equations that need to be solved to calculate an approximation for $\hat{X}_5$ are:

$$\begin{aligned} \hat{X}_5 &= \theta_1(X_4 - \hat{X}_4) + \theta_2(X_3 - \hat{X}_3) + \theta_3(X_2 - \hat{X}_2) \\ \hat{X}_4 &= \theta_1(X_3 - \hat{X}_3) + \theta_2(X_2 - \hat{X}_2) + \theta_3(X_1 - \hat{X}_1) \\ \hat{X}_3 &= \theta_1(X_2 - \hat{X}_2) + \theta_2(X_1 - \hat{X}_1) \\ \hat{X}_2 &= \theta_1(X_1 - \hat{X}_1) \\ \hat{X}_1 &= 0 \end{aligned} \tag{2.10}$$

Note that terms e_t, X_t, $\hat{X}_t$ with $t \leq 0$ are implicitly defined with a value of 0.

All non trivial approximations of a Moving Average model produce such a system of linear equations to be solved and can be transformed to a system of linear equations of the form

$$\mathbf{A}\vec{\hat{x}} = \vec{b} \tag{2.11}$$

with $\mathbf{A} \in \mathbb{R}^{n \times n}$ being an $n \times n$ Matrix, $\vec{\hat{x}} \in \mathbb{R}^n$ and $\vec{b} \in \mathbb{R}^n$ both n-dimensional vectors. The vector of $\vec{b}$ representing the known terms for each equation, $\vec{\hat{x}}$ the unknown terms and $\mathbf{A}$ the coefficients of all the $\vec{\hat{x}}$ for each equation.

The known terms in case of the MA(q) process are terms that consist out of θ and X_t and unknown terms are combinations of θ with $\hat{X}_t$.

To see how $\mathbf{A}$ and $\vec{b}$ have to be constructed, the system of linear equations (2.9) can be transformed to:

$$\begin{aligned}
\hat{X}_1 &= 0 \\
\theta_1\hat{X}_1 + \hat{X}_2 &= \theta_1 X_1 \\
\theta_2\hat{X}_1 + \theta_1\hat{X}_2 + \hat{X}_3 &= \theta_1 X_2 + \theta_2 X_1 \\
\theta_3\hat{X}_1 + \theta_2\hat{X}_2 + \theta_1\hat{X}_3 + \hat{X}_4 &= \theta_1 X_3 + \theta_2 X_2 + \theta_3 X_1 \\
\theta_3\hat{X}_2 + \theta_2\hat{X}_3 + \theta_1\hat{X}_4 + \hat{X}_5 &= \theta_1 X_4 + \theta_2 X_3 + \theta_3 X_2
\end{aligned} \tag{2.12}$$

This leads directly to the general form for systems of linear equations (2.11):

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ \theta_1 & 1 & 0 & 0 & 0 \\ \theta_2 & \theta_1 & 1 & 0 & 0 \\ \theta_3 & \theta_2 & \theta_1 & 1 & 0 \\ 0 & \theta_3 & \theta_2 & \theta_1 & 1 \end{pmatrix} \begin{pmatrix} \hat{X}_1 \\ \hat{X}_2 \\ \hat{X}_3 \\ \hat{X}_4 \\ \hat{X}_5 \end{pmatrix} = \begin{pmatrix} 0 \\ \theta_1 X_1 \\ \theta_1 X_2 + \theta_2 X_1 \\ \theta_1 X_3 + \theta_2 X_2 + \theta_3 X_1 \\ \theta_1 X_4 + \theta_2 X_3 + \theta_3 X_2 \end{pmatrix} \tag{2.13}$$

Having the system of linear equations in this particular form allows the usage of numerical solvers to do the heavy lifting when calculating the values for $\hat{X}_t$. These approaches are described in detail in chapter 2.3.

Autoregressive Moving Average

The ARMA(p, q) model can now be defined by combining the AR(p) and MA(q) model as defined in 2.1.3 and 2.1.4.

Definition 2.1.5 *Given a univariate and stationary time series X_t with a constant mean of 0 and the error term $e_t \sim iid(0, \sigma^2)$ the ARMA(p, q) model is defined as:*[8, p. 55]

$$\begin{aligned}
(1 - \sum_{i=1}^{p} \phi_i B^i) X_t &= (1 + \sum_{k=1}^{q} \theta_k B^k) e_t \\
\Leftrightarrow X_t - \phi_1 X_{t-1} - \phi_2 X_{t-2} - ... - \phi_p X_{t-p} &= e_t + \theta_1 e_{t-1} + \theta_2 e_{t-2} + ... + \theta_p e_{t-q}
\end{aligned} \tag{2.14}$$

with $\phi_1, ..., \phi_p$ and $\theta_1, ..., \theta_q$ being the parameters of the model.

The approximation $\hat{X}_t$ can therefore be calculated by:

$$\begin{aligned} \hat{X}_t &= \sum_{i=1}^{p} \phi_i B^i X_t + \sum_{k=1}^{q} \theta_k B^k e_t \\ \Leftrightarrow \hat{X}_t &= \phi_1 X_{t-1} + \phi_2 X_{t-2} + ... + \phi_p X_{t-p} + \theta_1 e_{t-1} + \theta_2 e_{t-2} + ... + \theta_p e_{t-q} \end{aligned} \tag{2.15}$$

For an ARMA$(2, 3)$ model this equation would change to:

$$\hat{X}_t = \phi_1 X_{t-1} + \phi_2 X_{t-2} + \theta_1 e_{t-1} + \theta_2 e_{t-2} + \theta_3 e_{t-3} \tag{2.16}$$

Assuming that again the length of the time series X is 4, the approximation for $\hat{X}_5$ would produce a system of linear equations that can be expressed as:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ \theta_1 & 1 & 0 & 0 & 0 \\ \theta_2 & \theta_1 & 1 & 0 & 0 \\ \theta_3 & \theta_2 & \theta_1 & 1 & 0 \\ 0 & \theta_3 & \theta_2 & \theta_1 & 1 \end{pmatrix} \hat{X} = \begin{pmatrix} 0 & & & & \\ \phi_1 X_1 & & +\theta_1 X_1 & & \\ \phi_1 X_2 & +\phi_2 X_1 & +\theta_1 X_2 & +\theta_2 X_1 & \\ \phi_1 X_3 & +\phi_2 X_2 & +\theta_1 X_3 & +\theta_2 X_2 & +\theta_3 X_1 \\ \phi_1 X_4 & +\phi_2 X_3 & +\theta_1 X_4 & +\theta_2 X_3 & +\theta_3 X_2 \end{pmatrix} \tag{2.17}$$

Autoregressive Integrated Moving Average

To define ARIMA the definition of ARMA (2.1.5) is extended by the definition of the integration component. This component is supposed to generalize ARMA, restricted to the class of stationary time series, to also incorporate the classes of non stationary time series.

Generally there are two approaches that can be used so the restricted ARMA model can be used for non stationary time series as well. Both are based on transforming the non stationary time series X_t to a stationary time series X'_t. Given that the time series X_t does not also exhibit seasonal behavior, one approach would be to fit a polynomial trend by for example using least squares, to the time series X_t and then subtracting the fitted trend from X_t, resulting in a stationary time series X_t‘. The second option is to remove the trend by differencing the time series X_t. The obvious advantage of the second method

being the computational complexity on the one hand and that it can also remove trends that don't remain constant over time on the other hand. [8, p. 25]

Definition 2.1.6 *The differencing operator* Δ *is defined by:*

$$\Delta X_t = X_t - X_{t-1} = (1 - B)X_t \tag{2.18}$$

Furthermore the j-th power of the differencing operator is $\Delta^j X_t = \Delta(\Delta^{j-1} X_t)$ with $\Delta^0 X_t = X_t$. Polynomials with Δ and B are manipulated in precisely the same way as polynomial functions of real variables.[8, p. 29] For example:

$$\begin{aligned} \Delta^2 X_t &= \Delta(\Delta X_t) \\ &= (1 - B)(1 - B)X_t \\ &= (1 - 2B + B^2)X_t \\ &= X_t - 2X_{t-1} + X_{t-2} \end{aligned} \tag{2.19}$$

To show that differencing can be used to remove the trend of a time series, lets assume that the time series can be described with a linear trend function $m_t = c_0 + c_1 t$. Then applying the differencing operator results in the constant function:

$$\begin{aligned} \Delta m_t &= m_t - m_{t-1} \\ &= c_0 + c_1 t - (c_0 + c_1(t-1)) \\ &= c_1 t - c_1(t-1) \\ &= c_1 \end{aligned} \tag{2.20}$$

It can be shown that the same holds truth for any polynomial trend of degree k: $m_t = \sum_{j=0}^{k} c_j t^j$. And if a time series can be described with such a linear trend function by $X_t = m_t + Y_t$ and Y_t is already stationary with zero mean, the application of Δ^k gives a stationary process with mean $k! c_k$:

$$\Delta^k X_t = k! c_k + \Delta^k Y_t \tag{2.21}$$

Applying the differencing operator once more yields a stationary process with mean zero.

This suggest that given any time series X_t, there is a k for which the time series $\Delta^k X_t$ can be modelled using a stationary process. Often the order of differencing k is quite small, usually one or two, as many functions can be approximated well using a polynomial of reasonable low degree. [8, p. 30]

Based on this we can finally define the ARIMA process:

Definition 2.1.7 *Given a univariate time series X_t with a constant mean of 0 and the error term $e_t \sim iid(0, \sigma^2)$ the ARIMA(p, d, q) model is defined as:*[8, p. 55]

$$(1 - \sum_{i=1}^{p} \phi_i B^i) \quad \Delta^d X_t \quad = \quad (1 + \sum_{k=1}^{q} \theta_k B^k) e_t \tag{2.22}$$

with $\phi_1, ..., \phi_p$ and $\theta_1, ..., \theta_q$ being the parameters of the model. The parameter p is known as the order of the Autoregression model, the parameter q as the order of the Moving Average model and the parameter d as the order of difference.

2.2. Optimization Algorithm

To forecast using an ARIMA model, the AR and MA coefficients ϕ and θ have to be estimated first. This is done by solving an optimization problem:

$$\begin{gathered} \max_{\phi_{1..p}, \theta_{1..q} \in R} \; g(\phi_{1..p}, \theta_{1..q}) \\ or \\ \min_{\phi_{1..p}, \theta_{1..q} \in R} \; g(\phi_{1..p}, \theta_{1..q}) \end{gathered} \tag{2.23}$$

with the function g being the Maximum Likelihood, Yule-Walker, or Conditional Sum of Squares estimator. However, the Yule-Walker estimator is usually only used for pure Autoregression models.[8, p. 139] It is therefore not discussed any further.

In case of the Maximum Gaussian Likelihood (ML) estimator, the data is assumed to be a realization of a stationary Gaussian time series and one tries to maximize the likelihood with respect to the parameters $\phi_1, ..., \phi_p$ and $\theta_1, ..., \theta_p$. The maximum of the likelihood function can not be calculated by solving a system of linear equations algebraically and is instead found by using numeric approaches. However, the algorithms commonly used for this need initial values for the coefficients to start the search from. The closer the initial guess is to the real coefficients the faster the search will be. [8, p. 138]

An alternative to the Maximum Likelihood estimator is the estimation using the Conditional Sum of Squares. This is a least squares method and the idea is to minimize the the sum of the squared residuals.

Definition 2.2.1 *Given a univariate time series X_t of size T the $ARIMA_{CSS}$ function is defined as:*

$$ARIMA_{CSS}(\phi_{1..p}, \theta_{1..q}) = \tfrac{1}{2}\sum_{t=1}^{T}(e_t)^2 = \frac{1}{2}\sum_{t=1}^{T}(X_t - \hat{X}_t)^2 \tag{2.24}$$

The CSS method is said to be slower then the Maximum Likelihood estimator. But on the flip side its performance is also not as dependent on the initial guess as the ML estimator's performance is. This is why the Maximum Likelihood estimator is commonly preferred over the CSS method, even though the CSS method is still in use. Either to quickly find a "good guess" for the ML estimator to start with or completely on its own. That it is also used on it's own is mostly due to the fact that it is the simpler estimator and more easily implemented.

But no matter which one of the functions to be optimized over is chosen, none of them can be maximized or minimized by solving a system of linear equations algebraically. They all need a numeric optimization algorithm to be solved.

There are different algorithms that have been developed to solve such a optimization problem. Some of the most common ones that are also provided by the general-purpose optimization function *optim*() of R are called:[15]

- Nelder-Mead method,
- Conjugate Gradient (CG) method,
- Broyden-Fletcher–Goldfarb-Shanno (BFGS) method,
- Limited-memory Broyden-Fletcher–Goldfarb-Shanno (L-BFGS) method, and
- SANN method

The Nelder-Mead method also known as downhill simplex method is a direct search method known to be robust but slow. As a direct search method it only uses the function values and doesn't need a gradient, making it work reasonably well for non-differentiable functions as well.[15]

The SANN method is a variation of simulated annealing given in Belisle (1992)[3] and is a part of the class of stochastic global optimization methods. Like the Nelder-Mead method it also uses function values only and is fairly slow.[15]

The BFGS algorithm on the other hand belongs to the class of the *Newton* methods or more specifically to the class of *Quasi-Newton* methods. It is said to be faster than the other methods provided by the general-purpose optimization function *optim*() and uses both the function values and the gradients. As a *Newton* method it is iteratively approximating the root $\hat{x}$ of $g(x)$ using the gradient of the function $g : R^n \to R^n$ by updating $\hat{x}$ with the inverse of the Jacobi matrix $J_g(x)$:

$$\vec{\hat{x}}_{n+1} = \vec{\hat{x}}_n - \frac{g(\vec{\hat{x}}_n)}{J_g(\vec{\hat{x}}_n)} = \vec{\hat{x}}_n - g(\vec{\hat{x}}_n) \cdot J_g^{-1}(\vec{\hat{x}}_n) \tag{2.25}$$

When using the *Newton* method to optimize a function $f : R^n \to R$ instead of finding its root this is equivalent to finding the root of the *gradient* Δf, which means that the approximation $\vec{\hat{x}}_{n+1}$ for an extrema of f is given by[15]:

$$\begin{aligned} \vec{\hat{x}}_{n+1} &= \vec{\hat{x}}_n - \frac{\Delta f(\vec{\hat{x}}_n)}{J_{\Delta f}(\vec{\hat{x}}_n)} \\ &= \vec{\hat{x}}_n - \frac{\Delta f(\vec{\hat{x}}_n)}{H_f(\vec{\hat{x}}_n)} \\ &= \vec{\hat{x}}_n - \Delta f(\vec{\hat{x}}_n) \cdot H_f^{-1}(\vec{\hat{x}}_n) \end{aligned} \tag{2.26}$$

With $H_f(\hat{x}_n)$ being the Hessian, a square matrix of all *second-order* partial derivatives.

As a *Quasi-Newton* method the BFGS algorithm is not calculating the Hessian or Jacobi matrix directly but instead tries to approximate it as well. This is important because just calculating the Hessian takes $O(n^2)$ function evaluations and inverting any matrix as an additional computational complexity of $O(n^3)$ using standard techniques.

As an improvement of the BFGS method the Limited-memory Broyden-Fletcher–Goldfarb-Shanno algorithm has been developed. It allows to limit the memory usage by introducing additional constraints on the coefficients of the function to be optimized over. To be more specific, it requires to add box constraints to each variable, giving them an lower and/or upper bound. [15]

Another alternative that is also relying on both the function values and the gradients is the Conjugate Gradient method. It works similar to the L-BFGS algorithm and has a similar convergence speed as well. But it doesn't need to store a matrix like the BFGS algorithm and can therefore be much more successful for larger optimization problems. However, the L-BFGS algorithm outperforms in term of speed for more computationally expensive functions because it needs less function evaluations than CG.[1]

To be able to use either one of the last two methods the gradient of $ARIMA_{CSS}$ needs to be calculated first. This can be done by using the finite differencing approach, recalculating

the approximation of the gradient for each new set of coefficients, or by working out the gradients algebraically and providing a way to calculate the gradient directly. Obviously the advantage of the first approach is its simplicity and that it requires less implementation work. It is also less prone to implementation errors but this comes at a cost. Using the finite differencing approach for an ARMA(p, q) process requires $2 * (p + q)$ function evaluations where the direct calculations of the gradient functions requires only one. Therefore the direct calculation of the gradients is preferred to increase the performance and the gradients for ϕ_n and θ_n for the $ARIMA_{CSS}$ function (2.24), given the approximation function for $\hat{X}_t$ in equation 2.15, are given by:

$$\begin{aligned} \frac{\partial}{\partial \phi_n} ARIMA_{CSS}(\phi_{1..p}, \theta_{1..q}) &= \frac{1}{2} \sum_{t=1}^{T} 2 \cdot (X_t - \hat{X}_t) \cdot (\frac{\partial}{\partial \phi_n}(X_t - \hat{X}_t)) \\ &= \sum_{t=1}^{T} e_t \cdot (-B^n X_t) \end{aligned} \tag{2.27}$$

$$\frac{\partial}{\partial \theta_n} ARIMA_{CSS}(\phi_{1..p}, \theta_{1..q}) = \sum_{t=1}^{T} e_t \cdot (-B^n e_t) \tag{2.28}$$

2.3. Solvers for System of Linear Equations

No matter which optimization algorithm is chosen and how good it is, the computational complexity of the function calculating $\hat{X}_t$ (2.15) will be the most important factor when it comes to determining the efficiency of the training algorithm for ARIMA. And the most computational heavy part of this function is solving the system of linear equations as described by equation 2.17. The problem with this particular system of linear equations is that its size growths quadratically with the size of the time series X. So if one wished to train an ARIMA model with a time series of size T, then in each function call to $ARIMA_{CSS}$ requires a system of linear equations with T free variables and a coefficient matrix of T^2 entries to be solved. This is why solving the system of linear equations in an efficient manner is important.

Fortunately, there are many algorithms that have been developed to accomplish this task. But to choose the most promising ones the properties of the system need to be discussed first.

Lets first consider the example of the system of linear equations for the ARMA$(2,3)$ process given in chapter 2.1. And to make it easier to see the patterns lets assume that instead of a time series of size 4 we have a time series with 7 samples and are trying to calculate the approximation $\hat{X}_8$. Remembering that the formula of $\hat{X}_t$ for the ARMA$(2,3)$ process was

$$\hat{X}_t = \phi_1 X_{t-1} + \phi_2 X_{t-2} + \theta_1 e_{t-1} + \theta_2 e_{t-2} + \theta_3 e_{t-3} \tag{2.29}$$

the system of linear equations would look like:

$$\begin{aligned}
\hat{X}_1 &= 0 \\
\hat{X}_2 &= \phi_1 X_1 + \theta_1 (X_1 - \hat{X}_1) \\
\hat{X}_3 &= \phi_1 X_2 + \phi_2 X_1 + \theta_1 (X_2 - \hat{X}_2) + \theta_2 (X_1 - \hat{X}_1) \\
\hat{X}_4 &= \phi_1 X_3 + \phi_2 X_2 + \theta_1 (X_3 - \hat{X}_3) + \theta_2 (X_2 - \hat{X}_2) + \theta_3 (X_1 - \hat{X}_1) \\
\hat{X}_5 &= \phi_1 X_4 + \phi_2 X_3 + \theta_1 (X_4 - \hat{X}_4) + \theta_2 (X_3 - \hat{X}_3) + \theta_3 (X_2 - \hat{X}_2) \\
\hat{X}_6 &= \phi_1 X_5 + \phi_2 X_4 + \theta_1 (X_5 - \hat{X}_5) + \theta_2 (X_4 - \hat{X}_4) + \theta_3 (X_3 - \hat{X}_3) \\
\hat{X}_7 &= \phi_1 X_6 + \phi_2 X_5 + \theta_1 (X_6 - \hat{X}_6) + \theta_2 (X_5 - \hat{X}_5) + \theta_3 (X_4 - \hat{X}_4) \\
\hat{X}_8 &= \phi_1 X_7 + \phi_2 X_6 + \theta_1 (X_7 - \hat{X}_7) + \theta_2 (X_6 - \hat{X}_6) + \theta_3 (X_5 - \hat{X}_5)
\end{aligned} \tag{2.30}$$

Of course, trying to examine the system in this form would be more tedious than it has to be and therefore we'll examine the system in its general form and as a matrix equation $A\vec{\hat{x}} = \vec{b}$:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \theta_1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \theta_2 & \theta_1 & 1 & 0 & 0 & 0 & 0 & 0 \\ \theta_3 & \theta_2 & \theta_1 & 1 & 0 & 0 & 0 & 0 \\ 0 & \theta_3 & \theta_2 & \theta_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & \theta_3 & \theta_2 & \theta_1 & 1 & 0 & 0 \\ 0 & 0 & 0 & \theta_3 & \theta_2 & \theta_1 & 1 & 0 \\ 0 & 0 & 0 & 0 & \theta_3 & \theta_2 & \theta_1 & 1 \end{pmatrix} \vec{\hat{X}} = \begin{pmatrix} 0 & & & & \\ \phi_1 X_1 & & +\theta_1 X_1 & & \\ \phi_1 X_2 & +\phi_2 X_1 & +\theta_1 X_2 & +\theta_2 X_1 & \\ \phi_1 X_3 & +\phi_2 X_2 & +\theta_1 X_3 & +\theta_2 X_2 & +\theta_3 X_1 \\ \phi_1 X_4 & +\phi_2 X_3 & +\theta_1 X_4 & +\theta_2 X_3 & +\theta_3 X_2 \\ \phi_1 X_5 & +\phi_2 X_4 & +\theta_1 X_5 & +\theta_2 X_4 & +\theta_3 X_3 \\ \phi_1 X_6 & +\phi_2 X_5 & +\theta_1 X_6 & +\theta_2 X_5 & +\theta_3 X_4 \\ \phi_1 X_7 & +\phi_2 X_6 & +\theta_1 X_7 & +\theta_2 X_6 & +\theta_3 X_5 \end{pmatrix} \tag{2.31}$$

Note that the vector $\vec{\hat{X}}$ represents the unknowns of the system, the vector $\vec{b}$ the known constant terms and the matrix A the coefficients. Determining the properties of A will give us insight about the system of linear equations and tell us which solving algorithms are viable and help to determine the effectiveness.

First of all, the coefficients matrix A is a squared matrix and has only 1s on its main diagonal. Its upper triangle is zero and the only other non zero values are on the diagonals beneath the main diagonal. The matrix A can therefore be described as a lower triangular matrix. Being a triangular matrix the matrix has a determinant that is equal to the product of its main diagonal. Hence, $det(A) = 1$ which means the matrix is invertible as any matrix that as a non zero determinant is invertible. Furthermore it is diagonally dominant if $\theta_1 + \theta_2 + \theta_3 < 1$.

Another important aspect to note is that the distribution of the values θ_1, θ_2 and θ_3 are creating a distinct pattern in the matrix as they are all having their own diagonal. And not just that, they are positioned in a clear pattern on the lower diagonals right under the main diagonal with θ_1 on the first diagonal beneath, θ_2 on the second one beneath and θ_3 on the third. This implies that the same characteristics of A for the ARMA$(2, 3)$ process might also be generalized to any coefficient matrix A for an ARMA(p, q) process.

Consider the formula for $\hat{X}_t$ for ARMA(p,q) and imaging to calculate $\hat{X}_t$ for all t from 0 up to the the size of the time series l:

$$\begin{aligned}
\hat{X}_1 &= 0 \\
\hat{X}_2 &= \sum_{\substack{i=1 \\ i<2}}^{p} \phi_i B^i X_2 + \sum_{\substack{k=1 \\ k<2}}^{q} \theta_k B^k (X_2 - \hat{X}_2) \\
\hat{X}_3 &= \sum_{\substack{i=1 \\ i<3}}^{p} \phi_i B^i X_3 + \sum_{\substack{k=1 \\ k<3}}^{q} \theta_k B^k (X_3 - \hat{X}_3) \\
&\vdots \\
\hat{X}_{l-1} &= \sum_{\substack{i=1 \\ i<(l-1)}}^{p} \phi_i B^i X_{l-1} + \sum_{\substack{k=1 \\ k<(l-1)}}^{q} \theta_k B^k (X_{l-1} - \hat{X}_{l-1}) \\
\hat{X}_l &= \sum_{\substack{i=1 \\ i<l}}^{p} \phi_i B^i X_l + \sum_{\substack{k=1 \\ k<l}}^{q} \theta_k B^k (X_l - \hat{X}_l)
\end{aligned} \tag{2.32}$$

Notice the second condition of each sum defining that the index i and k are bound by the index of $\hat{X}$. This has only been implicit before because X_t was defined only for $t \in \mathbb{N}^+$ and it was therefore assumed that $\forall t \in \mathbb{Z} \wedge t \leq 0 : BX_t = 0$.

From this form we can now infer that the characteristics found for the matrix A of equation 2.31 are for the most part also the characteristics of any other system of linear equations that is produced by an ARMA(p,q) process. In particular these characteristics are:

1. Main diagonal with constant 1
2. Upper triangle is zero
3. Lower triangle is non zero
4. Non zero values are on the diagonals right below the main diagonal.
5. Each diagonal has a constant value
6. There are exactly q diagonals filled with non zeros under the main diagonal. Each θ_i being a constant of one of the diagonals for all $1 \leq i \leq q$

The first one is quite easy to see by looking at the equations in 2.32. Each equation starts with the unknown $\hat{X}_t$ on the left side and no other occurrence of this unknown on the other side. Therefore it doesn't cancel out at any time and keeps its coefficient of 1. In the transformation to the notation as a matrix equation this therefore translates into the multiplication of the $\vec{\hat{X}}$ vector with 1 for each equation. Resulting in the main diagonal of 1s.

Next up is the upper triangle of the matrix. For this to have any non zero value, one of the equations would need to include an unknown $\hat{X}_t$ on the right hand side of the equation with t greater than the $\hat{X}_t$ on the left hand side. Considering the equations this can not be. As any $\hat{X}_t$ on the right hand side exists only in combination with the backshift operator B^j with $j \geq 1$ resulting in any $\hat{X}_t$ to be transformed to $\hat{X}_{t-j}$ with $t - j < t$. Therefore the upper triangle is zero.

The lower triangle on the other hand has non zero values precisely for this reason. That these non zero values are occurring in the distinct patterns that has been described before can be more easily seen when transforming the last equation of 2.32 into:

$$\sum_{\substack{k=1 \\ k<l}}^{q} \theta_k B^k \hat{X}_l + \hat{X}_l = \sum_{\substack{i=1 \\ i<l}}^{p} \phi_i B^i X_l + \sum_{\substack{k=1 \\ k<l}}^{q} \theta_k B^k X_l \tag{2.33}$$

Now the left hand side describes the l-th row of the matrix. With a one in the l-th column of the l-th row and the value of θ_k in the $l-k$-th column. And because decreasing l by one means that θ_k shifts one column to the left when compared to the previous row l, the pattern that has been described emerges. As there are q different θ_k there are always exactly q diagonals filled with the values of θ_1 to θ_k with θ_1 on the first diagonal under the main diagonal, θ_2 on the second and in general with θ_k on the k-th diagonal beneath the main diagonal.

Having shown that the matrix A is indeed a lower triangular matrix with a main diagonal with constant 1 we can conclude that it is invertible. Furthermore, the matrix is diagonal dominant if $\sum_{k=1}^{q} \theta_k < 1$.

And as mentioned before these properties play an important role in choosing one of the many algorithms that can efficiently solve this system of linear equations. In the second part of this chapter some of these methods will be discussed. Including the Gauss-Seidel, the Successive Over-Relaxation and the Conjugate Gradient method as well as the Jacobi and forward substitution method. Additionally there will also be an introduction of a more direct approach, that is calculating the inverse of A to solve the equations.

The Gauss-Seidel method is an iterative approach, updating the solution matrix x of the linear system of equations $Ax = b$ by solving the equations of the system one at a time. The update formula for the k-th approximation of x is:

$$x_i^{(k)} = \frac{b_i - \sum_{j<i} a_{i,j} x_j^{(k)} - \sum_{j>i} a_{i,j} x_j^{(k-1)}}{a_{i,i}} \tag{2.34}$$

For this method, however, the convergence is only proven for matrices that are diagonally dominant, symmetric and positive definite. Furthermore, because $x_i^{(k)}$ relies not only on the previous iteration $x^{(k-1)}$ but also on all the components of the same iteration $x_j^{(k)}$ with $j < i$ the updates can not be done simultaneously.[4] Note that $j > i$ of $a_{i,j}$ are the components of the upper triangular and for out system of linear equations this means that the second sum will therefore be zero. Later we will see that for out particular case this makes the update of the Gauss-Seidel method equal to the update of the Jacobi method.

The Successive Over-Relaxation method is derived from the Gauss-Seidel method and was proven to be able to converge faster by an order of magnitude.[5] It does this by extrapolating the Gauss-Seidel method and approximates x by updating $x_i^{(k)}$ with the weighted average of $x_i^{(k-1)}$ and the Gauss-Seidel iterate $\bar{x}_i^{(k)}$:

$$x_i^{(k)} = \omega \bar{x}_i^{(k)} + (1 - \omega) x_i^{(k-1)} \tag{2.35}$$

It was shown that choosing an appropriate value for ω accelerates the rate of convergence. However, it has also been proven that Successive Over-Relaxation fails if ω is outside the interval $(0, 2)$.[6] And because the update of the Successive Over-Relaxation method is relying on the calculation of the Gauss-Seidel iterate $\bar{x}_i^{(k)}$ it has the same dependencies on the previous components Gauss-Seidel iterate $\bar{x}_j^{(k)}$ with $j < i$ and can therefore also not calculate the updates in parallel.

The Conjugate Gradient method on the other hand is known for its applications in parallel computing. And it can not only be used to solve optimization problems as described in chapter 2.2 but is also useful for solving systems of linear equations. The convergence matrix, however, needs to be symmetric and positive definite.[12, p. 3] And the algorithm is not robust for ill-conditioned systems.[12, p. 19] This makes it especially hard to apply the CG method to the problem at hand. Because one could argue that even though the coefficient matrix A is not symmetric, a solution vector x could still be found if one were to solve the system $\bar{A}x = \bar{b}$ with $\bar{A} = A^T A$ instead. $\bar{A}$ would then be symmetric and it can be shown that the solution x stays the same when choosing $\bar{b} = A^T b$:

$$\begin{aligned} & Ax &&= b \\ \Leftrightarrow\ & A^T Ax &&= A^T b \\ \Leftrightarrow\ & \bar{A}x &&= \bar{b} \quad \wedge \quad \bar{A} = A^T A \quad \wedge \quad \bar{b} = A^T b \end{aligned} \tag{2.36}$$

Unfortunately it is not as easy as that. Multiplicating the coefficient matrix A with A^T would result in an ill-conditioned system. If the condition number of A would be K then

the condition number of $\bar{A}$ would be K^2. And because the Conjugate Gradient method is not suited for ill-conditioned systems this approach would fail.

Even though there are ways to reduce the condition number by introducing a so called preconditioner, the preferred alternative is to use the Biconjugate Gradient method, which has been developed so the algorithm could also be applied to non-symmetric systems.[12, p. 19] However, the trade off for this is a higher numerical instability leading to an explosion in terms of its computation time and that except for a few papers there is now much known about the algorithm.[12, p. 16] The next generation in the family of CG methods has been the Conjugate Gradient Squared method which improved the speed of the algorithm to converge in half the time of Bi-CG. This is partially achieved by removing the need to transpose A, which is beneficial when dealing with large matrices or matrices that are not definite. But CGS is still not addressing the issues of the numerical instability that the Biconjugate Gradient introduced and because it is using a squared function there is a build up of rounding and overflow errors.[16, p. 19] Fortunately, these issues have been mostly addressed by the newest member of CG family: The Biconjugate Gradient Stabilized method that was published in 1992. It is introducing a smoothing function that attempts to reduce the spikes of the CGS method and by doing so stabilizing the Bi-CG method while retaining the speed of the CGS algorithm.[16, p. 19]

Another algorithm that is commonly used when in need of a parallel solver for systems of linear equations is the Jacobi method. It is similar to the Gauss-Seidel method in terms of its update formula:

$$x_i^{(n+1)} = \frac{1}{a_{ii}}(b_i - \sum_{\substack{j=1 \\ j \neq i}} a_{ij} \cdot x_j^{(n)}) \tag{2.37}$$

But in contrast to the Gauss-Seidel or Successive Over-Relaxation method its update $x_i^{(n)}$ is independent of all other $x_i^{(n)}$. The drawback is that for non parallel application it converges slower than the Gauss-Seidel method. A big advantage, however, when compared to the

other algorithms is it's simplicity and robustness: The update method can be derived just from the matrix form of the system of linear equations in a few steps:[13, p. 133]

$$\begin{aligned} & \sum_{j=1}^{n} a_{ij} \cdot x_j &&= b_i \\ \Leftrightarrow \quad & a_{ii} \cdot x_i + \sum_{\substack{j=1 \\ j \neq i}} a_{ij} \cdot x_j &&= b_i \\ \Leftrightarrow \quad & a_{ii} \cdot x_i &&= b_i - \sum_{\substack{j=1 \\ j \neq i}} a_{ij} \cdot x_j \\ \Leftrightarrow \quad & x_i &&= \tfrac{1}{a_{ii}}(b_i - \sum_{\substack{j=1 \\ j \neq i}} a_{ij} \cdot x_j) \end{aligned} \tag{2.38}$$

In matrix form the Jacobi update can be expressed by assuming that $A = D + R$ where D is the main diagonal of A and R is the remainder:

$$x^{(n+1)} = D^{-1}(b - Rx^{(n)}) \tag{2.39}$$

It has been proven that Jacobi converges for any system which coefficient matrix is diagonally dominant.[9, p. 16] Fortunately for us this is only a sufficient but not necessary condition and it has also been shown that the method converges if the spectral radius of the iteration matrix $D^{-1}R$ is less than 1. [9, p. 12] The spectral radius of a matrix A is defined by:

$$\rho(A) = max\{|\lambda| : \lambda \text{ is eigenvalue of } A\} \tag{2.40}$$

In the case of the linear system of the ARMA(p, q) process, the spectral radius of the iterative matrix is indeed smaller than one because it can be shown that $\rho(D^{-1}R) = 0$. This is because R is actually a strictly lower triangular matrix, meaning its a triangular matrix with zeros on the main diagonal. Because of this the characteristic polynomial is $\mathcal{P}_R(\lambda) = \lambda^n$ for $R \in \mathbb{R}^{n \times n}$ and therefore all Eigenvalues λ_i are 0. Then obviously $max\{|\lambda| : \lambda$ is eigenvalue of $A\}$ is also zero, which proves that the Jacobian method converges for all linear systems of the ARMA(p, q) process.

Another approach that is also used to solve linear systems is the LU factorization or LU decomposition. The idea is to represent A by the product of two matrices L and U that are both triangular matrices. L a lower triangular matrix and U an upper triangular matrix. This method leverages the unique properties that systems with triangular coefficients bring to the table: They can be solved directly by substituting the unknown x_n of the $(n+1)$-th

equation with the solution acquired by solving the n-th equation. This is only possible because the n-th equation does not rely on any x_i where $i \geq n$:

$$\begin{array}{ccccccc} l_{1,1}x_1 & & & & & = & b_1 \\ l_{2,1}x_1 & + & l_{2,2}x_2 & & & = & b_2 \\ \vdots & & \vdots & & \ddots & & \vdots \\ l_{m,1}x_1 & + & l_{m,2}x_2 & + \ldots + & l_{m,m}x_m & = & b_m \end{array} \tag{2.41}$$

In this form it is obvious that the solution to the first equation is trivial and $x_1 = \frac{b_1}{l_{1,1}}$, making it also trivial to solve the second equation with $x_2 = \frac{b_1 - l_{2,1}x_1}{l_{1,1}}$ as x_1 is not an unknown anymore. This process of substituting all the unknowns x_i except for one can be continued for all the equations up to the last one by using the formula:

$$x_k = \frac{b_k - \sum_{i=1}^{k-1} l_{k,i}x_i}{l_{k,k}} \tag{2.42}$$

This method of solving a system of linear equations is called forward substitution and has as its only condition that A is a lower triangular matrix. The counter part of this method for upper triangular systems is called backward substation and as the name suggests it works analogue to the forward substitution method, but instead of starting with the first equation and iterating forward, it is starting with the last equation and iterates backwards. The biggest advantage of this algorithm is that it is relatively fast and only has a quadratic complexity in terms of multiplication and subtraction operations. Unfortunately none of the calculations can be done simultaneously.

The last alternative that is going to be discussed in this chapter is one that might also be considered the most obvious: Considering the system of linear equations $Ax = b$ the straight forward approach would be to calculate x by multiplying b with the inverse of A. This is generally considered to be a inefficient approach, because the complexity of inverting a matrix is known to be $O(n^3)$ using standard techniques. But in contrast to some of the other methods this approach can be parallelized and might therefore still be a viable option. Consider the the definition of the inverse of a squared matrix:

Definition 2.3.1 *The matrix $B \in \mathbb{R}^{n\times n}$ is the inverse of the matrix $A \in \mathbb{R}^{n\times n}$ if and only if $AB = I_n$ where I_n is the identity matrix $I_n \in \mathbb{R}^{n\times n}$*

And assume that A and B are lower triangular matrices and define them as 2×2 matrices constructed by $A_{1,1}, B_{1,1} \in \mathbb{R}^{k \times k}$, $A_{2,1}, B_{2,1} \in \mathbb{R}^{l \times k}$ and $A_{2,2}, B_{2,2} \in \mathbb{R}^{l \times l}$ with $l + k = n$ then the equation defining the inverse of A can be expressed as

$$\begin{pmatrix} A_{1,1} & 0 \\ A_{2,1} & A_{2,2} \end{pmatrix} \begin{pmatrix} B_{1,1} & 0 \\ B_{2,1} & B_{2,2} \end{pmatrix} = \begin{pmatrix} A_{1,1}B_{1,1} & 0 \\ A_{2,1}B_{1,1} + A_{2,2}B_{2,1} & A_{2,2}B_{2,2} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{2.43}$$

Considering the blockwise multiplication the following three equations can be derived:

$$\begin{aligned} A_{1,1}B_{1,1} &= 1 \\ A_{2,2}B_{2,2} &= 1 \\ A_{2,1}B_{1,1} + A_{2,2}B_{2,1} &= 0 \end{aligned} \tag{2.44}$$

Solving these gives the following formulas for B:

$$\begin{aligned} B_{1,1} &= A_{1,1}^{-1} \\ B_{2,2} &= A_{2,2}^{-1} \\ B_{2,1} &= -B_{2,2}A_{2,1}B_{1,1} \end{aligned} \tag{2.45}$$

Assuming that we have already inverted $A_{1,1}$ and $A_{2,2}$ recursively means that we can also calculating $B_{2,1}$ and by doing so calculating the last missing part of B, the inverse of A. Note that when recursively calculating $A_{i,i}$ at one point A will be a 1×1 matrix and the inverse B can simply be calculated by inverting the scalar $A_{1,1}$. And obviously every time the matrix A gets divided and the inverse of $A_{1,1}$ and $A_{2,2}$ is supposed to be calculated, they can be calculated independently of each other, allowing for a huge portion of the computation to be parallelized.

Now, to recap what has been introduced in this chapter: There are six different approaches for solving a system of linear equation that have been discussed so far. Three of them, the Gauss-Seidel, the Successive Over-Relaxation and the forward substitution method are all not parallelizable. Each of them is supposed to be faster than any of the other three methods, but when it comes to solving large systems of linear equations the solvers that can distribute their work load on multiple instances ought to be ahead in terms of

computation time. And these are the Biconjugate Gradient Stabilized method, the Jacobi method and the recursive inversion approach.

2.4. SystemML

SystemML is a Machine Learning Platform built for large scale analytics. It enables flexible and scalable machine learning while also accelerating exploratory algorithm development. It accomplishes this by providing the Declarative Machine Learning Language (DML). It can either be written in the default R-Like ("DML") or a Python-Like ("PyDML") syntax.[14] The goal of SystemML is to automatically scale any algorithm by translating all the instructions of a DML script into a set of *Spark* API calls so it can be run on a cluster in multiple nodes if necessary. Beforehand, SystemML also uses code optimization methods to remove dead code and common sub expressions. [7, p. 52]

Each script is optimized based on data and cluster characteristics, which means that the code is not only compiled and optimized once, but instead every time a script is invoked. Only at that point, all parameters the script can be run with are known and therefore the sizes and characteristics of the matrices (data) used in the script can be calculated. Using this information combined with other static characteristics of the cluster the script can be optimized further by, for example, calculating the number of nodes needed to run the script.[7, p. 52]

SystemML does this by translating a DML script through a series of transformations and rewrites into an executable program that can run in parallel on Spark or Hadoop. The script is taken through different layers of a well defined compilation chain that transforms it using 3 internal representations. The first representation being used, is a generic syntax tree that describes the original program. This representation is obtained by using parser that was produced by a parser generator that creates a parser based on a grammar describing either the R-like DML or Python-Like PyDML language. The script is parsed into the hierarchical representation of statement blocks. This step is responsible for a lexical and syntactical analysis. Afterwards a classical live variable analysis can be done as well as removal of dead code and then a semantical analysis checks for mandatory parameters of built-in functions.

At the end of the so called language layers, basic statement blocks have been identified. These basic blocks are chunk of straight line code (multiple lines) in between branches and for each of those blocks directed acyclic graph (DAGs) of high-level operations (HOPs) are created. These graphs are a representation of the data flow describing logical operations with their data dependencies and outputs. To be able to optimize the program to run

it in parallel one needs to know the characteristics of the data dependencies. And as the data in the machine learning domain typically comes in form of matrices SystemML computes the dimensions and the sparsity of the matrices for each high-level operation. These statistics are then propagated upwards to all the intermediate results - nodes of the directed acyclic graph of HOPs - and can then be used to determine whether a specific basic block can be calculated on a single node or can (or needs to) be distributed onto multiple ones. Additionally to determining the distributed operations, the HOPs layers are also responsible for rewrites of logical operations. These rewrite include algebraic simplifications, mandatory format conversions, common subexpression elimination and many more.

The third and final representation is derived by translating the HOPs DAGs into low-level operations (LOPs) DAGs which can then be used to generate run-time plans. In LOP DAGs each node represents a physical operation instead of a logical one, which, depending on the analysis of the HOPs DAGs, are either chosen to optimize parallel computing or in-memory computing one cluster node. Each low-level operation is runtime-backed-specific and SystemML offers the use of three different run-time environments. Therefore DML and PyDML scripts can be run in different modes. Either in Spark, Hadoop or Standalone mode (JVM). Additionally it can also be accessed via Scala or Python to be used in a Spark Shell, Jupyter or Zeppelin notebook. This enables easy and fast algorithm development in well established development environments commonly used for Machine Learning applications.[7, p. 54-56]

3. Methodology

The goal of this research project is to optimize the Autoregressive Integrated Moving Average training process, which can essentially be divided into two components: The optimization algorithm and the scoring function. If one wished to optimize the overall process one can either choose to improve the performance of the optimization algorithm or the performance of the scoring function. When only concentrating on one of these it is generally a better idea to improve the performance of the scoring function first, as it is usually easier. One might think that it would also be more efficient performance-wise to start with the scoring function, because its impact on the overall training algorithm might be greater. For example consider, that the complexity of both the optimization algorithm and the scoring function is $O(n^3)$. But for the optimization function $O(n^3)$ refers to the function calls of the scoring function and for the scoring function the measure refers to the number of multiplication operations. Then the complexity of the training algorithm in terms of multiplication operations would be $O(n^9)$ and when reducing the complexity of the scoring function to $O(n^2)$ would result in an overall complexity of $O(n^6)$ for the training algorithm. However, reducing the complexity of the optimization algorithm in the same fashion would have the same result. Nonetheless, the potential reduction of complexity for the scoring function is likely to be greater than that of the optimization algorithm. This is mainly because much is already known about the different optimization algorithms (see chapter 2.2). Including their convergence rates. This is why the scoring function is the focus of this research project and why the optimization and testing will be done for the scoring function only.

Of course the scoring function must be more closely defined as there are two alternatives: The Maximum Likelihood estimator and the Conditional Sum of Squares estimator. Even though the ML estimator is known to be faster, for the performance and precision testing the CSS estimator was chosen. The reason for this is twofold: First, the CSS method is much simpler to understand, implement and reason about and second, both scoring functions actually rely on calculating the approximation $\hat{X}_t$ for a given ARMA(p, q) process. This calculation is what can actually be improved and will impact both scoring function in the same way, as one call to either one of the scoring function results in exactly one call to this $\text{ARMA}_{\text{predict}}(\omega)$ function.

The final question that has to be addressed before discussing the implementation detail of the $\text{ARMA}_{\text{predict}}(\omega)$ and $\text{ARMA}_{\text{CSS}}(\omega)$ is which solver to use for the system of linear equations that needs to be solved in order to calculate $\hat{X}_t$ for any ARMA(p, q) where $q > 0$.

From the section 2.3 we know there are 6 different methods to choose from: Gauss-Seidel, Successive Over-Relaxation, Conjugate Gradient (including its variations), Jacobi, forward substitution and the direct approach using the recursively calculated inverse. We also know from the same chapter that the coefficient matrix of the system is neither symmetric nor positive definite. Therefore the Gauss-Seidel and Successive Over-Relaxation are unsuitable for the problem at hand. For the same reason the Conjugate Gradient, the Biconjugate Gradient method is not an option as well. The Conjugate Gradient Squared method is known to be unstable and therefore also not preferred. The Biconjugate Gradient Stabilized would be a valid option, but it is relatively complex compared to the other methods and therefore prone to implementation errors. Most importantly its complexity doesn't allow it to be used in the context of this research project because there is simply not enough time to implement and test this method properly. This leaves us with three methods: Jacobi, forward substitution and the direct approach using the recursively calculated inverse. Each one of these approaches is to be implemented and to be tested in terms of their performance to determine which is best suited to optimize the performance of the scoring function and by doing so the training algorithm.

Additionally to comparing the different solvers implemented in the DML language with each other, they should also be compared to the performance of the implementation of ARIMA in the *stats* library of the language R. However, the *stats* package only offers one kind of solver for the system of linear equation and is using the forward substitution method. Furthermore, the core implementation of the algorithm in the *stats* package is done not in R but in native C. Only the interface for the function is provided in R.

3.1. ARIMA Implementation

As mentioned, ARIMA is supposed to be tested using SystemML's Declarative Machine Learning Language and the implementation provided by R's statistics library. The following two sections describe how to run ARIMA on each systems respectively. The first section, however, covering the implementation in Declarative Machine Learning Language (DML), is much more detailed to the degree that it describes every aspect of the ARIMA prediction algorithm. The second section, on the other hand, is not describing the actual implementation of ARIMA at all, but only how to use the more complex ARIMA function of the *stats* package to call the ARIMA_{CSS} method in a way that is equivalent to the DML implementation.

DML Script

The goal of the *arima_css.dml* script is to calculate the Conditional Sum of Squares of ARIMA. Before diving into the details of the implementation, let's reconsider the formulas that should be implemented:

First of all, we have the formula for the Conditional Sum of Squares (definition 2.2.1):

$$ARIMA_{CSS}(\phi_{1..p}, \theta_{1..q}) = \tfrac{1}{2}\sum_{t=1}^{T}(e_t)^2 = \frac{1}{2}\sum_{t=1}^{T}(X_t - \hat{X}_t)^2$$

Where $\hat{X}_t$ for ARMA(p, q) can be calculated by solving the system of linear equations $A\hat{X} = b$ described by equation 2.33:

$$\sum_{\substack{k=1 \\ k<l}}^{q} \theta_k B^k \hat{X}_l + \hat{X}_l = \sum_{\substack{i=1 \\ i<l}}^{p} \phi_i B^i X_l + \sum_{\substack{k=1 \\ k<l}}^{q} \theta_k B^k X_l$$

where $A = I + R$ with the identity matrix I and $R_{j,i} = \theta_{j-i}$ for $i - j \leq q$ and $b = Z \cdot \omega$ is the matrix product of the weights $\omega =< \phi_1, \ldots, \phi_p, \theta_1, \ldots, \theta_q >^T$ and the auxiliary matrix $Z \in \mathbb{R}^{l \times (p+q)}$ defined by $Z_{i,j} = X_{(i \bmod p)-j}$.

To get a better understanding of R and Z and how they are defined reconsider the system of linear equations for ARMA$(2, 3)$ presented in chapter 2.3 in equations 2.30 and 2.31 and examine the matrices depicted there. The coefficient Matrix has already been discussed in detail in the same chapter and for the vector b on the right hand side of the equation note that every element $b_l = \sum_{\substack{i=1 \\ i<l}}^{p} \phi_i B^i X_l + \sum_{\substack{k=1 \\ k<l}}^{q} \theta_k B^k X_l$ can also be expressed as $b_l = \sum_{\substack{i=1 \\ i<l}}^{p} \phi_i Z_l^{(i)} + \sum_{\substack{k=1 \\ k<l}}^{q} \theta_k Z_l^{(k)}$ with $Z_l^{(i)} = B^i X_l$. As can be easily seen b_l can therefore be expressed as $b_l = \vec{\phi} \cdot \bar{Z}_l + \vec{\theta} \cdot \bar{\bar{Z}}_l$ with $\bar{Z}_l =< Z_l^{(1)}, \ldots, Z_l^{(p)} >^T$ and $\bar{\bar{Z}}_l =< Z_l^{(1)}, \ldots, Z_l^{(q)} >^T$.

This way of describing the equation 2.33 is used in the implementation of the *arima_predict* function shown in figure 3.1.

The *arima_predict* function takes five arguments: The weights ω, the auxiliary matrix Z, the non-seasonal Autoregression order p, the non-seasonal Moving Average order q, and the name of the algorithm that is to be used to solve the system of linear equations. It then calculates the vector of the constant terms b directly by multiplying Z with ω and continues to construct the coefficient matrix A before calling the function with the name of the algorithm that is supposed to solve the linear system. The construction of $A \in \mathbb{R}^{l \times l}$ starts with initializing A as the identity matrix and then iterating q times through a loop

```
arima_predict = function(Matrix[Double] weights, Matrix[Double] Z,
  ↪ Integer p, Integer q, String solver) return (Matrix[Double] x_hat){

    b = Z %*% weights

    A = diag(matrix(1, nrow(Z), 1)) #identity matrix

    if (q > 0){
        for(i in 1:q){
            A = addDiagonalToMatrix (A, as.scalar(theta[i]), nrow(A)-i)
        }
    }

    x_hat = eval(solver + "_solver", A, b)
}
```

Figure 3.1.: The *arima_predict* function

that adds an $(l-i) \times (l-i)$ identity matrix to A as described in more detail below. This smaller matrix M is multiplied with θ_i before being added to A. For the addition of the matrix M and A, the matrix A is defined as:

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \tag{3.1}$$

where $A_{1,1} \in \mathbb{R}^{(l-i)\times i}$, $A_{2,1} \in \mathbb{R}^{(l-i)\times(l-i)}$, $A_{1,2} \in \mathbb{R}^{i\times i}$ and $A_{2,2} \in \mathbb{R}^{i\times(l-i)}$. The addition of M and A is then defined by:

$$A + M = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} + M & A_{2,2} \end{pmatrix} \tag{3.2}$$

This addition of an identity matrix multiplied with a constant value is implemented in the *addDiagonalToMatrix* function shown in figure 3.2.

Next up is the implementation of the three different algorithms that are used to solve the system of linear equations. All the functions implementing on of the solvers have the same signature. They all return the solution vector x and take two arguments: The coefficient matrix A and the constant terms vector b.

```
addDiagonalToMatrix = function (Matrix[Double] M, Double diagValue,
  ↪ Integer diagSize)return (Matrix[Double] M){

    if (diagSize > 0){
        diagonal = diag(matrix(diagValue, diagSize, 1))
        startRow = nrow(M) - diagSize + 1
        endRow = nrow(M)
        startCol = 1
        endCol = diagSize
        M[startRow:endRow, startCol:endCol] = M[startRow:endRow,
         ↪ startCol:endCol] + diagonal
    }
}
```

Figure 3.2.: The *addDiagonalToMatrix* function

Starting with the simplest one - the forward substitution solver shown in figure 3.3. The update of x was defined in chapter 2.3 by equation 2.42 and adjusted to the notation used in the implementation is:

$$x_i = \frac{b_i - \sum_{k=1}^{i-1} A_{i,k} \cdot x_k}{A_{i,i}} = \frac{b_i - A_i \cdot x^{(i-1)}}{A_{i,i}}$$

Where $x^{(i)} =< x_1^{(i)}, x_2^{(i)}, \ldots, x_i^{(i)}, 0, \ldots, 0 >^T$ is the i-th update of the solution vector x with x_j for $j \leq i$ already calculated and $x_j = 0$ for $j > 0$. And this is exactly the formula that has been implemented in the *forwardsub_solver* function. Note that only the solution of the most recent update is saved, hence $x = x^{(i)}$, and that in DML there is no vector data type. Therefore both x and b are represented as $l \times 1$ matrices.

```
forwardsub_solver = function (Matrix[Double] A, Matrix[Double] b)  return
  ↪ (Matrix[Double] x){
    x = matrix(0, nrow(A), 1)
    for (i in 1:nrow(A)){
        x[i,1] = (b[i,1] - A[i,] %*% x[,1]) / A[i,i]
    }
}
```

Figure 3.3.: The *forward substitution* solver

The *Jacobi* solver's implementation shown in figure 3.4 is a little bit more complicated. Again, let's start remembering the update function for this iterative method, which is given in equation 2.37:

$$x_i^{(n+1)} = \frac{1}{a_{ii}}(b_i - \sum_{\substack{j=1 \\ j \neq i}} a_{ij} \cdot x_j^{(n)})$$

Recalling that for the Jacobi method A was defined as $A = D + R$, the update function can be rewritten in vector notation as:

$$x^{(n+1)} = D^{-1} \cdot (b - R \cdot x^{(n)})$$

```
32 jacobi_solver = function (Matrix[Double] A, Matrix[Double] b)  return
   ↪  (Matrix[Double] x){
33     tolerance = 1.0E-8
34     max_iterations = 1000
35
36     x = matrix(0, nrow(A), 1)
37     iter = 0
38     diff = tolerance+1
39     diagVector_A = diag(A)
40     rest_A = A - diag(diagVector_A)
41
42     while(iter < max_iterations & diff > tolerance){
43         x_new =1/diagVector_A * (b - rest_A %*% x)
44         diff =sum(abs(x_new-x))
45         iter = iter + 1
46
47         if (toString(diff) != "NaN") {
48             x = x_new
49         }
50     }
51 }
```

Figure 3.4.: The *Jacobi* solver

As before in the implementation of the forward substation method, x is initialized as a $n \times 1$ matrix filled with zeros. Additionally the two constants *tolerance* and *max_iterations* are introduced and initialized. Afterwards the matrices D and R are derived from A. The main part of the function is a while loop that runs as long as the solution has not converged or the maximum iteration count has been reached. In each iteration the update function (2.39) is applied, the difference between $x^{(n)}$ and $x^{(n+1)}$ calculated and the iteration counter increased. Note that, first, $D^{-1} = \frac{1}{D}$ because D is a strictly diagonal matrix, and second, that $x^{(n)} = x^{(n+1)}$ is only applied in line 48 if *diff* does not have any *NaN* values. This

sometimes occurs when there is an overflow resulting in *Infinity* or *-Infinity* values in $x^{(n)}$. This sometimes leads to SystemML trying to add *Infinity* to *-Infinity*, which is an undefined operation and results in a *NaN* value. When this happens the solver won't recover by itself as any operation with a *NaN* value results in another *NaN* value. Therefore the solver is implemented in the way that it holds on to the last $x^{(n)}$ that doesn't have a *NaN* value. Instead of letting the while loop continue to run until the maximal amount of iterations has been reached, one should stop the loop as soon as line 47 evalutes *true*. Unfortunately there is neither a *break* nor an explicit *return* operator in DML that could be used to achieve this.

In the following figure 3.5 the implementation of the "*inverse*" solver is shown. The function of the solver itself is just calling the *L_triangular_inv* function to recursively invert the coefficient matrix A and then uses this to directly calculate the solution $x = A^{-1} \cdot b$. But the *L_triangular_inv* function is actually doing the heavy lifting.

```
52 inverse_solver = function (Matrix[Double] A, Matrix[Double] b)  return
   ↪  (Matrix[Double] x){
53     invA = L_triangular_inv(A);
54     x = invA %*% b
55 }
56
57 L_triangular_inv = function(Matrix[double] L) return(Matrix[double] A) {
58     n = ncol(L)
59     if (n == 1) {
60         A = 1/L[1,1]
61     }
62     else {
63         k = as.integer(floor(n/2))
64
65         L11 = L[1:k,1:k]
66         L21 = L[k+1:n,1:k]
67         L22 = L[k+1:n,k+1:n]
68
69         A11 = L_triangular_inv(L11)
70         A22 = L_triangular_inv(L22)
71         A12 = matrix(0, nrow(A11), ncol(A22))
72         A21 = -A22 %*% L21 %*% A11
73
74         A = rbind(cbind(A11, A12), cbind(A21, A22))
75     }
76 }
```

Figure 3.5.: The inverse solver

The *L_triangular_inv* function only takes one argument: the matrix L that it is supposed to invert. As this is an recursive function, the simplest case - n equal to 1 - is calculated first with the variable n being the number of columns of the matrix L. Having the case that L is an 1×1 matrix, the inverse is trivial and given by $L^{-1} = 1/L$. For the general case of $n > 1$ the inverse is determined recursively as described in chapter 2.3 and can be calculated using the equations 2.45. Adjusted to the notation used in the implementation these are:

$$\begin{aligned} A_{1,1} &= L_{1,1}^{-1} \\ A_{2,2} &= L_{2,2}^{-1} \\ A_{2,1} &= -A_{2,2} L_{2,1} A_{1,1} \end{aligned}$$

where L and A are 2×2 matrices constructed by $L_{1,1}, A_{1,1} \in \mathbb{R}^{k \times k}$, $L_{2,1}, A_{2,1} \in \mathbb{R}^{l \times k}$, $L_{2,2}, A_{2,2} \in \mathbb{R}^{l \times l}$ with $l + k = n$ and $L \cdot L^{-1} = I$ with $L^{-1} = A$ is rewritten as:

$$\begin{pmatrix} L_{1,1} & 0 \\ L_{2,1} & L_{2,2} \end{pmatrix} \begin{pmatrix} A_{1,1} & 0 \\ A_{2,1} & A_{2,2} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

In the implementation, first, L is divided into $L_{1,1}$, $L_{2,1}$, and $L_{2,2}$ and then $A_{1,1}$ and $A_{2,2}$ are calculated recursively. Afterwards they are used to calculated $A_{2,1}$ as described by equation 2.45.

Having described the *arima_predict* function as well as each of the algorithms used for solving the system of linear equation, we can come to the final code snippets needed for the *arima_css.dml* script.

To goal of the *arima_css.dml* script was to calculate the Conditional Sum of Squares. To do so the residuals of ARIMA have to be calculated. They are defined as the difference between the values given for X_t and the approximation $\hat{X}_t$ and therefore calculated by $\text{res} = X - \hat{X}$. The function that does this is shown in figure 3.6 and takes the same arguments as the *arima_predict* function plus one additional one referred to as *ncond*. This is the number of initial observations to be ignored and is equal to the maximum lag of AR, which is p. This changes the original form of the AR model in the way that it is only defined for $\hat{X}_t$ which can be expressed using all values ϕ_i of the model. One could consider to skip this and act like *ncond* was set to zero. However, because the default value for ncond used in the implementation of R's *stats* package, this is also applied to the DML implementation. This is why in line 83 the residuals for $1 \leq t \leq$ ncond are set to 0.

```
77 arima_residuals = function(Matrix[Double] weights, Matrix[Double] X,
   ↪ Matrix[Double] Z, Integer p, Integer q,  Integer ncond, String
   ↪ solver) return (Matrix[Double] residuals){
78
79     x_hat = arima_predict(weights, Z, p, q, solver)
80     residuals = X - x_hat
81
82     if (ncond > 0)
83         residuals[1:ncond] = matrix (0, rows = ncond, cols = 1)
84
85     residuals = replaceNaN(residuals, 0)
86 }
```

Figure 3.6.: The *arima_residuals* function

Also note that any *NaN* value that might have been produced by one of the linear system solvers is set to zero as well. Because the only use of the residuals is to be summed up, this is equal to just ignoring each of the *NaN* values. Furthermore, these *NaN* values only occur if at least one other residual is *Infinity* or *-Infinity*. The sum will therefore be either on of these values anyways no matter what all the other values are.

In figures 3.7 and 3.8 two different functions can be seen that are used for calculating the Conditional Sum of Squares. The first one simply calculates the squared sum of the residuals, normalized by dividing through the number of observations that were used in the model. The normalization is also a detail of R's implementation that was incorporated into the DML script to make the results of the two comparable.

```
87 arima_sumofsquares = function(Matrix[Double] weights, Matrix[Double] X,
   ↪ Matrix[Double] Z, Integer p, Integer q,  Integer ncond, String
   ↪ solver) return (Double sumofsquares, Matrix[Double] residuals){
88
89     residuals = arima_residuals(weights, X, Z, p, q, ncond, solver)
90     sumofsquares = sum(residuals^2)/(nrow(X) - ncond)
91 }
```

Figure 3.7.: The *arima_sumofsquares* functions

The second function is the actual *arima_css* function that uses the *arima_sumofsquares* function to calculate the Conditional Sum of Squares which was defined previously as:

$$ARIMA_{CSS}(\phi_{1..p}, \theta_{1..q}) = \tfrac{1}{2} ARIMA_{SSQ}(\phi_{1..p}, \theta_{1..q}) = \tfrac{1}{2} \sum_{t=1}^{l} (e_t)^2 = \frac{1}{2} \sum_{t=1}^{l} (X_t - \hat{X}_t)^2$$

But this is not exactly what has been implemented. To ensure that the results of the R and DML implementation match up, the definition of $ARIMA_{CSS}$ was changed to:

$$ARIMA_{CSS}(\phi_{1..p}, \theta_{1..q}) = \frac{1}{2} \ln \left(\sum_{t=1}^{l} (X_t - \hat{X}_t)^2\right)$$

```
92  arima_css = function(Matrix[Double] weights, Matrix[Double] X,
    ↪  Matrix[Double] Z, Integer p, Integer q,  Integer ncond, String
    ↪  solver) return (Double css, Matrix[Double] residuals){
93
94      [sumofsquares,residuals] = arima_sumofsquares(weights, X, Z, p, q,
        ↪  ncond, solver)
95      css = 0.5 * log (sumofsquares)
96  }
```

Figure 3.8.: The *arima_css* functions

To be able to call the *arima_css* functions for a given ARIMA model, there are two steps: The first one is differencing the time series. This has to be done because *arima_predict* is only defined for an ARMA model and not for an ARIMA model. And the second one is constructing the auxiliary matrix Z that was discussed in the beginning of this section.

The differencing operator (definition 2.1.6) was defined as:

$$\Delta X_t = X_t - X_{t-1} = (1 - B)X_t$$

with $\Delta^d X_t = \Delta(\Delta^{d-1} X_t)$. Hence, the difference operator Δ^d is implemented iteratively with X_t denoted as $X[2:n]$ and X_{t-1} as $X[1:n-1]$ where n is the length of the time series. Note that the difference of $\Delta X_1 = X_1$ because $X_{1-1} = X_0$ is undefined. This is true for all d and therefore the constants *ncond* should be adjusted accordingly.

```
97  difference = function (Matrix[Double] X, Integer d) return
    ↪  (Matrix[Double] X){
98      if (d > 0){
99          for(i in 1:d){
100             X[2:nrow(X)] = X[2:nrow(X)] - X[1:(nrow(X)-1)]
101         }
102     }
103 }
```

Figure 3.9.: The *difference* function

The *constructPredictorMatrix* function shown in figure 3.10 can be used to create the auxiliary matrix Z. As described in the beginning of this section Z is a $n \times (p+q)$ matrix where n is the length of the time series X. Each column of Z is a copy of X but with an offset that corresponds to the index of the column. The idea behind the way Z is build, is to provide n vectors containing all values of X that are needed to calculate the approximation of $\hat{X}_t$. Because there are p values needed for the Autoregression model of $\hat{X}_t$ and q values for the Moving Average each row vector has the length $p+q$. Furthermore, the offset of X's copy in each column is the column index i for $i \leq p$ and $i-p$ for $i > p$.

For *ncond* greater than zero, the columns containing the X values for the Moving Average model need to be adjusted so the residuals calculated by the *arima_residuals* function are the same as the residuals of R's ARIMA. Unfortunately, the reasons for changing the definition of ARIMA by introducing the constant *ncond* are not documented by the R Core Team that implemented the *stats* package. However, this particular change made in lines 122 to 127 is necessary to ensure that there are no inconsistencies introduced by changing the definition of ARIMA. If one would only set the residuals e_t for $t \leq ncond$ to zero after all e_t have been calculated, then all $\hat{X}_t$ for $t > ncond$ would be calculated using the assumption that e_t is defined the same way for all t. Which by the changed definition it is not.

The function used to add a copy of X with a given offset to the matrix Z is called *addShiftedMatrix* and is implemented in figure 3.11. This function takes two matrices, the target and a source matrix and two additional scalar arguments, the row offset and the index of the column that the new values of the source matrix are supposed to be copied to. Assuming that the target is Z and the source is X the function would set $Z_{i,j} = X_{j-k}$ with k being the row offset and i the index of the column. But only for $j-k > 0$.

Finally all functions necessary for calculating the Conditional Sum of Squares have been discussed and the only thing left to discuss is the main body of the *arima_css.dml* script shown in figure 3.12. As can be seen, the script first tries to read the parameters that can be provided when executing the script. Only two of them being non-optional parameters: The source of the time series file to be read into X and the source of the file that contains the weights ϕ_i and θ_j for $i \in \{1, \ldots, p\}$ and $j \in \{1, \ldots, q\}$. The optional parameters are the non seasonal orders p, q and d of AR,MA and the difference. All of them are initialized with zero if not provided. Note that none of the values can be negative and either p or q has to be non zero. The solver for the system of linear equation can also be specified. It can be one of the following: "forwardsub" for the forward substitution method, "jacobi" for the Jacobi method and "inverse" for using the recursively calculated inverse. The default is set to "jacobi". The last three parameters that can be specified are "css_out", "residuals_out" and "result_format". The first two can be used to specify the

```
constructPredictorMatrix = function(Matrix[Double] X, Integer p, Integer q,  Integer ncond) return (Matrix[Double] Z){

    totalColumns = p + q
        Z = matrix(0, rows = nrow(X), cols = totalColumns)

    if (p > 0){
        for(i in 1:p){
            Z = addShiftedMatrix(Z, X, i, i)
        }
    }

    if (q > 0){
        for(i in 1:q){
            Z = addShiftedMatrix(Z, X, i, p + i)
        }
    }

    if (ncond > 0){
        Z[1:ncond,] = matrix(0, rows = ncond, cols = totalColumns)
        if (q > 0){
            for (i in 1:q){
                ignoreRows = min(ncond+i, nrow(Z))
                Z[1:ignoreRows,(p+i)] = matrix(0, rows = ignoreRows, cols = 1)
            }
        }
    }
}
```

Figure 3.10.: The *constructPredictorMatrix* function

```
addShiftedMatrix = function (Matrix[Double] targetMatrix,  Matrix[Double] sourceMatrix, Integer rowOffset, Integer nthColumn)return (Matrix[Double] targetMatrix){

    if (rowOffset < nrow(targetMatrix)){
        targetMatrix[(rowOffset+1):nrow(targetMatrix), nthColumn] = sourceMatrix[1:(nrow(targetMatrix)-rowOffset)]
    }
}
```

Figure 3.11.: The *addShiftedMatrix* function

location of the files that contain the output of the *arima_css* function and are initialized with the default values "arima-css.mtx" and "arima-residuals.mtx". The third one is used to define the format, that should be used to write the residuals and Conditional Sum of Squares into the files. These can be any of the formats supported by DML. The two most important are "csv" or "MM" which stands for the Matrix Market format.

After reading and initializing the parameters, the constant *ncond* is defined as the sum of p and q like it is for R's ARIMA. Afterwards the difference of X is calculated. Z is created based on the differenced time series and then the *arima_css* function is called with Z and the differenced time series. Finally the Conditional Sum of Squares is printed and written into the output file. The same is done for the residuals.

```
137 X = read($X)
138
139 solver = ifdef($solver, "jacobi")
140 css_out = ifdef($css_out, "arima-css.mtx")
141 residuals_out = ifdef($residuals_out, "arima-residuals.mtx")
142 result_format = ifdef($result_format, "MM")
143
144 p = ifdef($p, 0)
145 d = ifdef($d, 0)
146 q = ifdef($q, 0)
147
148 weights = ifdef ($weights_src, "")
149
150 ncond = d + p
151
152 diffX = difference(X, d)
153 Z = constructPredictorMatrix(diffX, p, q, ncond)
154 [css,residuals] = arima_css(weights, diffX, Z, p, q, ncond, solver)
155
156 print("arima_css = " + css)
157 write(css, css_out, format = result_format)
158 write(residuals, residuals_out, format = result_format)
```

Figure 3.12.: The *arima_css.dml* script's main body

R Script

Even though the *arima_css.r* script is rather long it is quite simple and pretty straight forward. As it is only a wrapper for R's *arima* function, its main task is to read the configuration parameters, measure the execution time and to handle the access to the Hadoop Distributed File System if necessary.

Unfortunately, R does not provide a native integration with the HDFS. There are a few libraries that could have been used to handle the file access, but they are deprecated and not necessarily suited for the newest version of R. Therefore the access was handled manually by executing a bash command that would copy the file from the HDFS to the local file system. Then load the data into memory and delete the temporary files from the local file system afterwards. Lines 172 to 194 show how this was implemented for the file containing the time series X and for the file containing the weights ϕ and θ. The same was done for the files containing the residuals and Conditional Sum of Squares. Only in the opposite direction from the local to distributed file system.

Because the copying of the files might take a significant amount of time, two time measurements were introduced. The first one referred to as "run_time", measures the time from the very beginning of the script until the very end. The second one called "execution_time" is only for measuring the time it takes to run the script from the point where it reads the data from the local file system to the point where it has run the *arima* function and written the results into the locally stored files.

```
159 library(Matrix)
160 start_time  <- Sys.time()
161 args = commandArgs(trailingOnly=TRUE)
162 XPath = args[1]
163 weightsPath = args[2]
164 cssoutputPath = args[3]
165 p= as.numeric(args[4])
166 d= as.numeric(args[5])
167 q= as.numeric(args[6])
168 usehdfs = args[7]
169
170 tryCatch({
171   if (usehdfs == TRUE){
172     tmp.Xfile <- sprintf("tmp_hadoop_X_%s.mtx", as.numeric(Sys.time()))
173     tmp.Wfile  <- sprintf("tmp_hadoop_W_%s.mtx", as.numeric(Sys.time()))
```

```
174     XPath = tmp.Xfile
175     weightsPath = tmp.Wfile
176     system(command = sprintf("%s dfs -cat %s | perl -pe 's/\t/,/g' > %s",
        ↪ "hdfs", XPath, tmp.Xfile))
177     system(command = sprintf("%s dfs -cat %s | perl -pe 's/\t/,/g' > %s",
        ↪ "hdfs", weightsPath, tmp.Wfile))
178   }
179   arima_css_start_time <- Sys.time()
180   data = as.matrix(readMM(XPath))
181   weights = as.matrix(readMM(weightsPath))
182 }, finally = {
183   if (usehdfs == TRUE){
184     cfile.remove(tmp.Xfile)
185     file.remove(tmp.Wfile)
186   }
187 })
188
189 fixedmodel = arima(data, fixed = weights, order = c(p,d,q), seasonal =
    ↪ list(order = c(0, 0, 0), period = 0), include.mean = FALSE,
    ↪ transform.pars = FALSE, method = c("CSS"))
190 sumofsquares = fixedmodel$sigma2
191 css = 0.5 * log(sumofsquares)
192 write.table(css, file = cssoutputPath, sep=",", row.names=FALSE,
    ↪ col.names= FALSE)
193
194 arima_css_end_time <- Sys.time()
195 if (usehdfs == TRUE){
196     # copy css output file from local cssoutputPath to hdfs
        ↪ cssoutputPath
197 }
198 end_time <- Sys.time()
199 sprintf("execution_time:  %f", (arima_css_end_time -
    ↪ arima_css_start_time))
200 sprintf("run_time:  %f", (end_time - start_time))
201 print(paste("arima_css=", css, sep=" "))
```

Figure 3.13.: The *arima_css.r* script

3.2. Performance and Precision Testing

After having implemented the *arima_css.dml* script as well as the *arima_css.r* script, we would like to know how do they compare? And even more important: How do the three linear system solvers impact the performance of the DML script and how do they compare to each other? Obviously we'd also like to make sure that the implementation of ARIMA in DML is indeed correct.

To test how precisely the *arima_css.dml* calculates the Conditional Sum of Squares compared to R's implementation, both *arima_css* scripts need to be run with the same configuration. Meaning the same time series, the same weights and the same values for p, d and q. The same has to be done, if we'd like to compare the time of the two implementations. Because the *arima_css.dml* script has one additional parameter - the linear system solver - to be specified, we would either have to compare R's implementation only to one of the solvers or run three times the amount of tests and compare each configuration of the R script with the same configuration for the DML script, but once with "jacobi" as solver, once with "forwardsub" and once with "inverse". Doing so would also give us the possibility to compare the solvers to each other, which is why this approach was chosen.

To automatically run these tests a simple shell script has been developed. It assumes that there is a configuration file in CSV format containing multiple entries of these configuration vectors that each describe how one test run should be done. This configuration is supposed to have twelve columns and an arbitrary amount of rows. These twelve columns are used to hold the following specifications:

1. X_src: The source of the time series file containing X
2. weights_src: The source of the weights file containing ϕ and θ
3. residuals_out: File name in which to put the resulting residual vector
4. solver: The solver for the linear system
5. p: Non-seasonal AR order
6. d: Non-seasonal differencing order
7. q: Non-seasonal MA order
8. P: Seasonal AR order
9. D: Seasonal differencing order
10. Q: Seasonal MA order

11. s: Seasonal period in terms of number of time-steps from one season to the next
12. XSize: The length of time series X

The shell script that can take this kind of configuration file and run two tests for each row - one with the R script and one with the DML script - is shown in figures 3.14 and 3.15. Even though both of these figures only show excerpts from the original *performanceTest.sh* they do show the relevant part of the shell script. The original one contains some more code for initializing some of the parameters the *performanceTest.sh* is executed with with default values. It does also contain a lot of boilerplate and debugging code which is not to be discussed here.

The first excerpt from the *performanceTest.sh* shows which parameters the script expects. There is a total of eighth parameters, that can be specified when executing the script:

1. "config": The path to the configuration file that was discussed earlier
2. "r_path": The path to the R script
3. "dml_path": The path to the DML script
4. "systemml": The path to SystemML.jar
5. "driver-memory": The memory dedicated to the spark driver. Used for specifying how much RAM the SystemML instance can use
6. "executor-memory":The memory dedicated to the spark executor. Used for specifying how much RAM the SystemML instance can use
7. "exec_mode": As described in the *SystemML Documentation*[14] this can be "standalone", "spark", "hadoop" or "hybrid"
8. "-usehdfs": Flag determining whether the R script should be looking for the time series and weights file on the local file system or on the HDFS (Optional)

The second excerpt, on the other hand, shows how one can iterate through the configuration file and execute the R and DML script to run a test for the given ARIMA model specifications. The outputs of both commands contains the error and debugging logs as well as the results for the Conditional Sum of Squares and most importantly the measurements for the execution time. To extract the measurements one can either choose to write the two variables containing the output into a file and use another script, maybe even of another language for greater convenience, or use another while loop to iterate through each line and look for keywords like "execution_time" or "run_time". Of course, the key words needed for extracting the measurements and results from the DML output are different then the ones from R's output.

```
for ARG in "$@"
do
  KEY=$(echo $ARG | cut -f1 -d=)
  VALUE=$(echo $ARG | cut -f2 -d=)
  case $KEY in
    "config") config=$VALUE;;
    "r_path") r_path=$VALUE;;
    "dml_path") dml_path=$VALUE;;
    "systemml") systemml_jar=$VALUE;;
    "driver-memory") driver_memory=$VALUE;;
    "executor-memory") executor_memory=$VALUE;;
    "exec_mode") exec_mode=$VALUE;;
    "-usehdfs") usehdfs=TRUE;;
  esac
done
```

Figure 3.14.: Excerpt from the *performanceTest.sh* shell script for reading the parameter

Note that the DML script is executed with the additional *-stats* flag. This prints out a detailed statistics describing the elapsed time and what it was used for. It shows how much was used for compilation, how much for execution, and of this execution time how much was used for the 10 most time consuming function calls. An example of what this would look like is shown in listing 29. The rest of the SystemML command is very straight forward. Spark is used to run SystemML with the memory available specified by the "driver_memory" and "executor_memory" parameters and all other parameters are passed as name variable arguments by using the *-nvargs* option.

In contrast to that the R script is invoked by passing the arguments by their position and calling R with the path to the script. Note that the *Rscript* does not load the "methods" package by default to save on start time. However, the "methods" package is required by the "Matrix" package used in the *arima_css.r* script and therefore needs to be added to the default packages loaded on start up.

```
SystemML Statistics:
Total elapsed time:   54.247 sec.
Total compilation time:   7.830 sec.
Total execution time:   46.417 sec.
Number of compiled MR Jobs: 40.
Number of executed MR Jobs: 0.
Cache hits (Mem, WB, FS, HDFS): 7800032/0/0/2.
Cache writes (WB, FS, HDFS):  600014/0/1.
Cache times (ACQr/m, RLS, EXP): 3.837/0.819/4.960/1.225 sec.
HOP DAGs recompiled (PRED, SB): 6/17.
HOP DAGs recompile time:  0.342 sec.
```

```
Functions recompiled:   2.
Functions recompile time: 0.058 sec.
Total JIT compile time:   54.659 sec.
Total JVM GC count:   11.
Total JVM GC time:    1.223 sec.
Heavy hitter instructions:
  #  Instruction          Time(s)    Count
  1  arima_css             40.756        1
  2  arima_sumofsquares    40.755        1
  3  arima_residuals       40.717        1
  4  eval                  34.188        1
  5  ba+*                  12.072   600001
  6  replaceNaN             6.199        1
  7  rightIndex             5.480  2400006
  8  rmvar                  5.002  6600038
  9  leftIndex              4.315   600008
 10  createvar              4.264  4800028
```

Listing 3.1: Example of *arima.dml* script run with *-stats* flag

```
while IFS=',' read -r X weights_src residuals_out solver p d q P D Q s
    ↪ Xsize
do
    dml_output=$(spark-submit --driver-memory $driver_memory
     ↪ --executor-memory $executor_memory $systemml_jar -f $dml_path
     ↪ -nvargs X=$X weights_src=$weights_src p=$p d=$d q=$q P=$P D=$D
     ↪ Q=$Q s=$s solver=$solver residuals_out=$dest_dml -exec $exec_mode
     ↪ -stats)

    r_output=$(Rscript --verbose
     ↪ --default-packages=methods,datasets,graphics,grDevices,stats,utils
     ↪ $r_path $X $weights_src "" $p $d $q $P $D $Q $s $dest_r $usehdfs)

done < "$config"
```

Figure 3.15.: Excerpt from the *performanceTest.sh* shell script showing how to execute the R and DML test runs for all configurations

In the full *performanceTest.sh* script the output of both the R and DML test runs are scanned to retrieve the relevant information like execution-, run-, elapsed- and compilation-time as well as the heavy hitter instructions and the resulting Conditional Sum of Squares. Together with their full configuration vector this information is then written into a new file to make the evaluation of the test results easier.

Now, the only piece missing for running the performance tests are the configuration files themselves. And to be able run multiple tests at once, three of these configuration files

were created. One file only containing configuration vectors for solving with "forwardsub", one for "jacobi" and one for "inverse". Each of them contained configurations for the following models: AR(3), AR(6), MA(3), MA(6), ARIMA$(2, 1, 2)$ and ARIMA$(4, 1, 4)$. And each of these models was tested with time series ranging in size from 1 to 950 in steps of 50 and from $1,000$ to $950,000$ in steps of $50,000$. So each model was tested with 40 different time series. At least if the model could be calculated for the given time series. Obviously this is not always the case, because the value of *ncond* needs to be greater than the size of the time series. Configurations for which this would have been the case were not included in the configuration files to begin with. And finally, each one of the configuration files was put through the *performanceTest.sh* script twice.

4. Results

The test runs that were done for a total of 708 different configurations with 236 different configurations for each solver. They were run twice for each configuration and the results where aggregated by calculating the mean. Nonetheless, only 649 configurations were recorded for which at least one of the runs was successful. And every single one of the 59 unsuccessful runs was a run using the "inverse" solver. In particular the ARIMA$(2, 1, 2)$ and ARIMA$(4, 1, 4)$ models were crashing for all time series with sizes starting from $50,000$ and the MA(3) and MA(6) models were *mostly* not working for time series with sizes starting from $150,000$. However, for the 649 configurations that were producing results, the maximal difference between the Conditional Sum of Squares that the R script calculated and that of the DML script was only $5e^{-13}$ and the mean difference was $4e^{-14}$. As the results of both the R and the DML script are only returning results with up to 16 decimal places, the minimal non-zero difference that could have been measured is $1e^{-16}$. Therefore there is no further need to examine the distribution of these computational errors.

However, the data collected on the performance of the algorithms is more complex by far. Hence, the following presentation of the performance measurements is structure in a way that first introduces the results in a reasonably general way and then drills further down into the details by showing the data with respect to the following six dimensions:

1. **Script type and solvers**: Whether the results are from the DML or R script and which type of solver was used in the DML script.
2. **Time measurement**: Whether the measurement was recording the execution time or the run time as described in chapter 3.2
3. **Time series category**: Whether the time series is part of the group of "small" or "large" time series. Small scale time series are defined as time series with sizes smaller or equal to $1,000$ and large scale time series are defined as time series with sizes greater than $1,000$
4. **Model class**: Whether the data describes an AR, MA or ARIMA model
5. **Model**: The exact model specified by its class and the order of the model. Data includes AR(3), AR(6), MA(3), MA(6), ARIMA$(2, 1, 2)$ and ARIMA$(4, 1, 4)$ models.
6. **Time series size**: The actual size in terms of number of observations

In its most comprehensive form the results of the performance tests are presented in figure 4.1 which shows the mean values of the execution and run time for all four script types while also differentiating the two time series size categories. From this representation alone, there are already 3 important properties of the test results that can be observed:

1. On average R is faster than any one of the three DML scripts. The only exception is the run time for small scale time series which is roughly three times slower than the fastest DML script. However, the difference of the other three values is much greater. On small scale with a factor of at least 15 and on large scale with a factor of at least 40 for run time and a factor of 140 for the execution time.

2. Generally the difference between execution and run time is greater for small scale time series than for large scale time series. For the DML scripts the difference is smaller than 1% for large scale time series and therefore insignificant. For R, on the other hand, the difference is significant independent of the time series' size.

3. Of the three DML scripts, the inverse one is the slowest by a huge margin. Both the Jacobi and Forward Substitution solver are more than twice as fast on average. Of these two, the Jacobi solver is 4.5% faster on average.

For the further presentation of the results this means that, first of all, the dimension of *time measurement* will be reduced to one element only - the execution time - for all comparisons that are focused on the model or solver dimension and secondly that the DML script with the inverse solver won't be the subject of any more detailed presentation as it is already obvious that it is considerably worse for large scale time series compared to all the other scripts and not significantly better on small scale. Note that, even though these parts of the results are not discussed in this chapter, there are additional tables and plots of a wider variety of views in appendix A.

Script	execution time	run time	execution time	run time
	small scale		large scale	
Inverse DML	1.24	3.15	1,767.3	1,769.44
Forwardsub DML	1.21	4.74	528.43	530.82
Jacobi DML	1.35	4.59	505.81	508.47
R	0.08	9.67	3.65	11.95

Table 4.1.: Comparison of **execution and run time** mean values of **DML and R** implementations for **small scale** (from 50 to 1,000) and **large scale** (from 50,000 to 950,000) time series

The next three figures 4.2, 4.3, and 4.4 show the statistical characteristics of the execution time in more detail by expanding the view on the data with the dimension of the model classes.

When comparing these tables with respect to the Autoregression models one can see that there is an average increase in execution time from small scale to large scale by factor of 5 for Jacobi, a factor of 15 for Forward Substitution and a factor of 75 for R. With the average large scale time series having 1000 times more observations than an average small scale time series. For small scale time series, Jacobi and Forward Substitution are pretty close in terms of their mean execution time, with Jacobi taking 10.8% less time. The absolute difference, however, is only 1/10 of a second. When comparing the execution time R is 13 times faster than Jacobi. However, this is not the case for the run time, because using the HDFS has a much grater impact on the performance of R than on DML. Therefore the run time of DML for small scale AR models is twice as good (see additional run time statistics A.1.6 and A.1.8). This is true for the averages of all model classes. For large scale time series Jacobi has the best average execution time. It even takes 15% less time than R to produce the results. And it is more than 3 times faster than the script using Forward Substitution

For the Moving Average models the difference between small scale and large scale is much greater than for the AR models. For the Jacobi and Forward Substitution scripts there is more than a 150 fold increase while the increase of R is only 18 fold. And both on small and large scale the R script is absolutely outperforming any of the DML implementations. For small scale it is 16 times faster than Forward Substitution, which itself takes 16% less time than Jacobi. For large scale R is even better than that with an average of 1.5 seconds to 226 seconds for Forward Substitution and 235 seconds for Jacobi, making it more than 150 times faster than any one of them. An interesting fact to note here is that the execution time of R for large scale time series with the Autoregression model is 4 times that of the Moving Average model.

For the complete Autoregressive Integrated Moving Average model there is an average increase in execution time from small to large scale by a factor of 955 for Jacobi and by 1227 for Forward Substitution. This is a huge difference. But note that for large scale time series the maximal value measured for both solvers is roughly 3 times the mean value. On the other hand, for the class of MA models the maximal execution time was only about 1.4 times the mean. The increase for R from small to large scale is only 45 fold and with an average execution time of only 3.6 seconds for large scale time series the R script is 370 times faster than any one of the DML scripts. Of the two DML scripts the script using the Jacobi solver is the faster one. But again the difference is only about 4%, making it statistically insignificant.

Stat.	AR	MA	ARIMA	AR	MA	ARIMA
	small scale			large scale		
min	0.47	0.5	0.46	1.27	20.4	10.09
max	2.32	3.25	3.16	10.64	340.35	4,091.19
mean	1.08	1.54	1.42	5.15	235.05	1,356.96
median	1.06	1.23	1.04	4.46	253.88	1,006.21
standard deviation	0.45	0.89	0.87	2.89	72.9	1,159.36

Table 4.2.: Statistical analysis of **DML execution time** for **small scale** (from 50 to 1,000) and **large scale** (from 50,000 to 950,000) time series for **Jacobi solver**

Stat.	AR	MA	ARIMA	AR	MA	ARIMA
	small scale			large scale		
min	0.58	0.5	0.52	1.98	20.42	9.04
max	2.2	4.5	2.3	42.6	315.31	4,049.64
mean	1.19	1.28	1.16	18.66	226.39	1,423.5
median	1.11	0.99	1.17	17.4	239.03	1,113.43
standard deviation	0.47	0.9	0.53	12.01	71.71	1,090.63

Table 4.3.: Statistical analysis of **DML execution time** for **small scale** (from 50 to 1,000) and **large scale** (from 50,000 to 950,000) time series for **Forward-substitution solver**

Stat.	AR	MA	ARIMA	AR	MA	ARIMA
	small scale			large scale		
min	0.05	0.05	0.05	0.57	0.24	0.4
max	0.19	0.17	0.17	13.03	3.21	7.71
mean	0.08	0.08	0.08	6.07	1.49	3.65
median	0.07	0.07	0.06	6.32	1.3	3.51
standard deviation	0.03	0.03	0.03	3.31	0.87	2.22

Table 4.4.: Statistical analysis of **R execution time** for **small scale** (from 50 to 1,000) and **large scale** (from 50,000 to 950,000) time series

So far the data was analyzed only on the first four dimensions: Script type and solvers, time measurement, time series category and model classes. In the following section of this chapter, the results are more closely looked at with regards to the last two dimensions time series size and model.

First, have a look at the scatter plots of figures 4.1, 4.2 and 4.3. They show how different models of the same model class are comparing to each other in terms of their execution time for large scale time series and the Jacobi solver.

In case of the class of Autoregression processes the two models are actually extremely closely correlated with a correlation of $r = 0.91$. What is most interesting about this is that even the measurements that look like they might be outliers, for example the observations for $450k$ and $550k$, still show this correlation. Of these 40 data points there is actually only two, the measurements for $800k$, that do not look like they are tightly correlated. Also note that the two trend lines that have been plotted show that, surprisingly, it is the AR(3) model that appears to be slightly better.

For the other two model classes MA and ARIMA there is no significant linear correlation. The correlation of the Moving Average models is only $r = 0.55$ and the correlation of the ARIMA models is even less with $r = 0.47$. However, for the class of MA processes one can clearly see, that the correlation could have been much better for most parts of the data. Especially for the measurements of time series with sizes greater than $400k$. There the correlation is 0.91. And for the time series from $1k$ to $250k$ the correlation is even 0.94. Just from looking at the graph 4.2 one can't tell that the the observations that are reducing the overall correlation so significantly are not part of the underlying pattern that seems to emerge from the rest of the data. Disregarding these abnormalities for a moment, the pattern mentioned is quite interesting as it is not simply linear. For the first few observations the measurements of each model look like they might be described with a quadratic function, but then the pattern abruptly changes to a linear model at the $150k$ mark. Note that from the linear trend lines that are shown in the plot one can clearly see that for the class of Moving Average models the higher order models take more time to compute.

In contrast to that, the correlation of the ARIMA models execution time does not seem like it might have been corrupted by a few outliers. However, the standard deviation for the ARIMA(4,1,4) model is significantly greater than the one for ARIMA(2,1,2). To be more precise, the standard deviation of the higher order model is 16 times greater (see appendix A.1.9). But this is only one factor that explains the low linear correlation. By comparing the linear trends of the two ARIMA models one can also see, that they are diverging greatly with increasing time series size, with the higher order model ARIMA(4,1,4) being the one taking longer and longer for each increase in time series size.

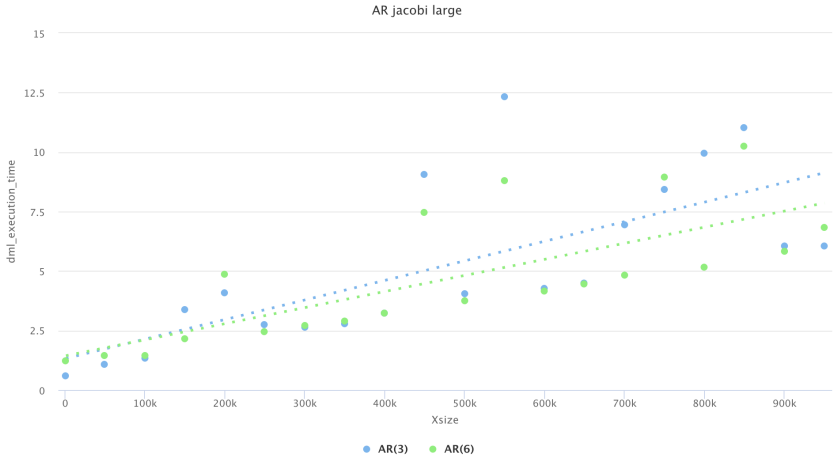

Figure 4.1.: Comparison of **Execution time** of **AR(3) and AR(6) for Jacobi-DML** for time series with sizes **from** $1,000$ **up to** $950,000$

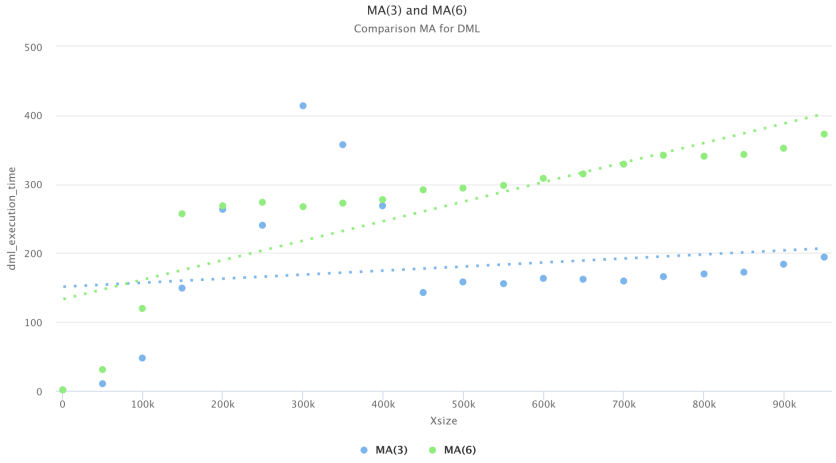

Figure 4.2.: Comparison of **Execution time** of **MA(3) and MA(6) for Jacobi-DML** for time series with sizes **from** $1,000$ **up to** $950,000$

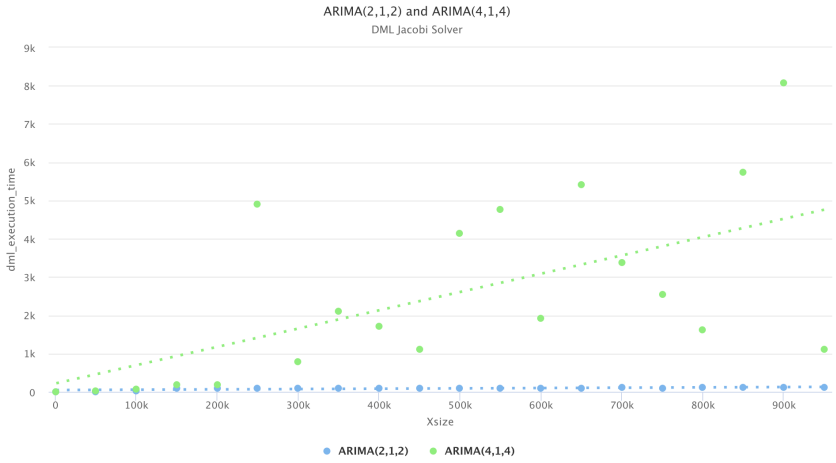

Figure 4.3.: Comparison of **Execution time** of **ARIMA(2,1,2) and ARIMA(4,1,4) for Jacobi-DML** for time series with sizes **from** $1,000$ **up to** $950,000$

The next two plots show the execution time dependent on the time series size for all four solvers and scripts. The first figure 4.4 for AR(3) and the second one, figure 4.5, for MA(3).

For the Autoregression model the data that can be seen in figure 4.4 is pretty straight forward. As was discussed in the beginning of this chapter: The inverse solver is completely off the charts, the Forward Substitution solver is slower than Jacobi and R, and the Jacobi solver is insignificantly better than R. Almost surprisingly there are no outliers or any other kind of unexpected patterns in the data.

This is definitely not the case for figure 4.5 showing the MA(3) model. As it could already be seen in one of the previous figures showing the class of Moving Average models, there is a significant set of observations that absolutely don't fit any obvious patterns. And what is most interesting about these observations ranging from $150k$ to $400k$ is that they first of all, are so much bigger than the following measurements and more importantly are occurring both for the Jacobi solver and the Forward Substitution solver, but not for the Inverse solver or for R. Setting these outliers a side again, one can see that the two approaches Jacobi and Forward Solver are equally good, but not comparable to the performance of R's implementation.

The plots describing the execution time of ARIMA(2,1,2) show similar characteristics as the one for Moving Average(3). The first figure 4.6 demonstrates the scale of the measurements and mainly shows why the mean value for the ARIMA models is so high. There are three outliers that impacted the mean disproportional. As mentioned before the maximal value for ARIMA was roughly 3 times the mean value. Removing only the three largest outliers results in a representation of the data as it can be seen in figure 4.7 and would mean an average execution time of only 90.45 seconds. As one would expect, the same pattern that could be observed for the Moving Average models is also being exhibited by the ARIMA(2,1,2) process. For the values from $1k$ up to $150k$ the patterns seems to be of a quadratic function, but then breaks and continues in the form of a linear model which also looks to be quite stable, if one is willing to disregard the outliers. These linear models are shown in figure 4.8. From looking at this last figure one can see that the Forward Substitution solver seems to be the slower one for increasing sizes in the time series. However, this difference is quite small and the trend lines based only on small set of observations. And finally, each one of these figures also shows that the implementation of R is much faster than any DML implementations.

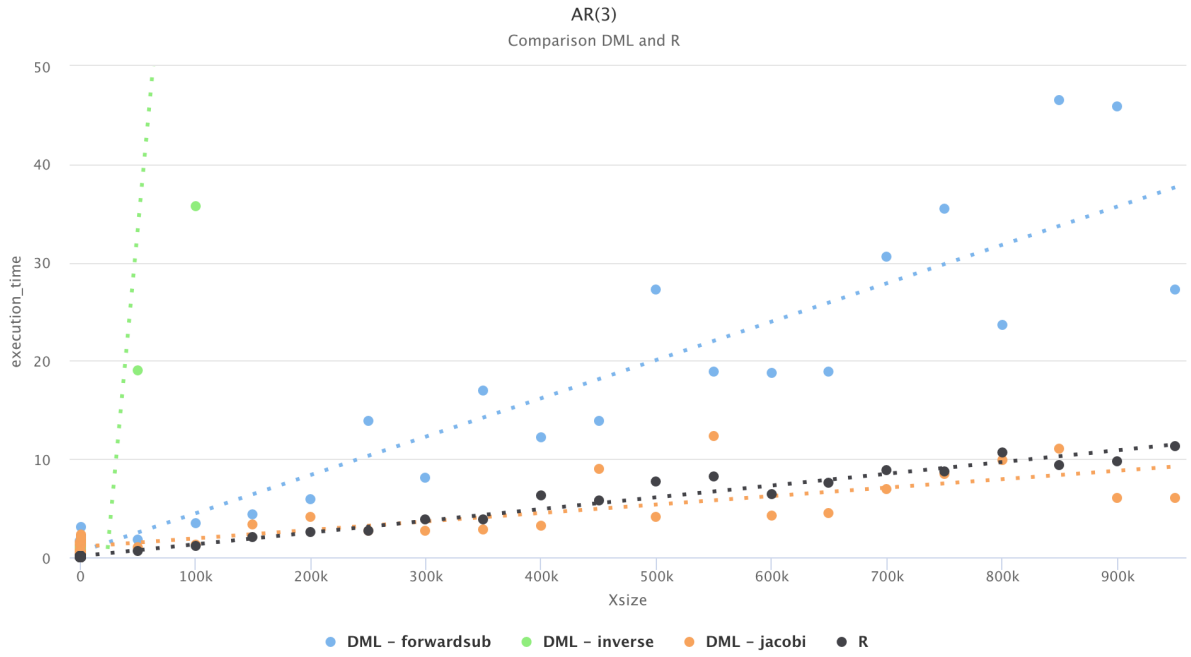

Figure 4.4.: **Execution time** of AR(3) for DML and R for time series with sizes **up to** 950,000

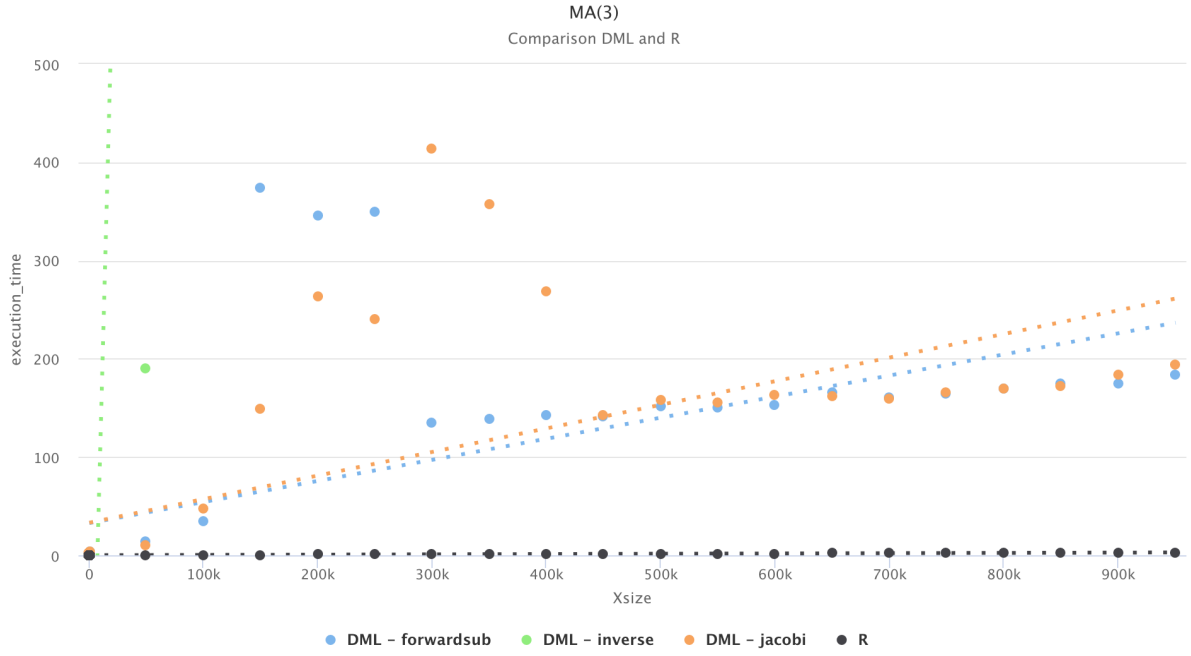

Figure 4.5.: **Execution time of MA(3) for DML and R for time series with sizes up to** 950,000

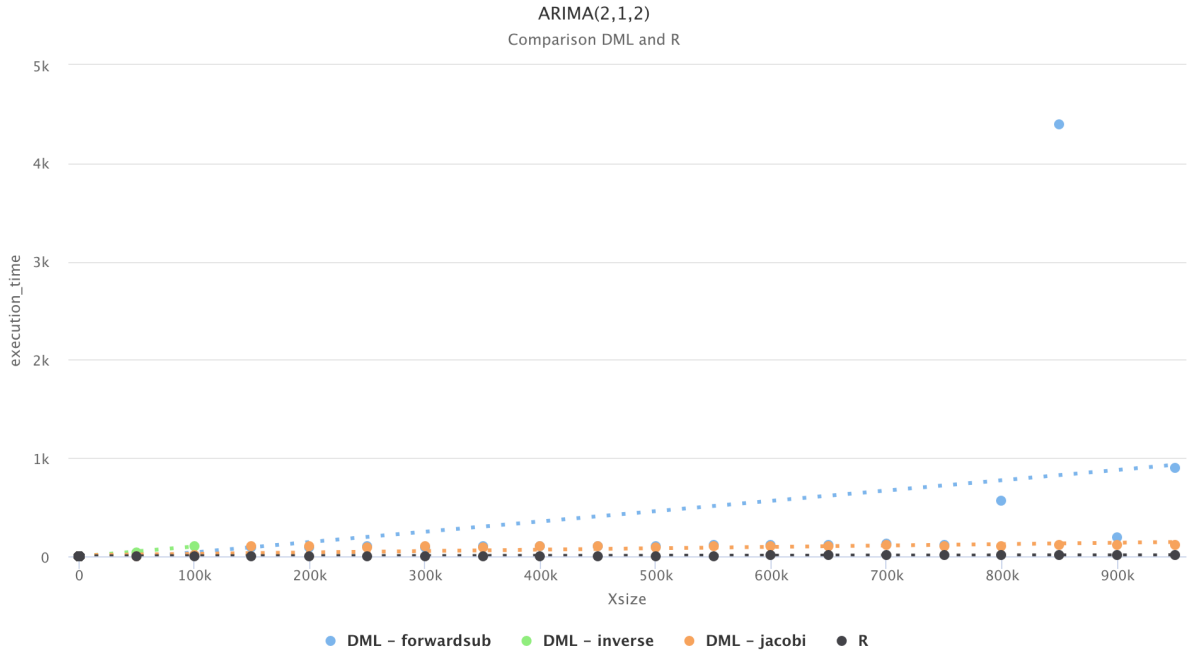

Figure 4.6.: **Execution time of ARIMA(2,1,2) for DML and R for time series with sizes up to** $950,000$

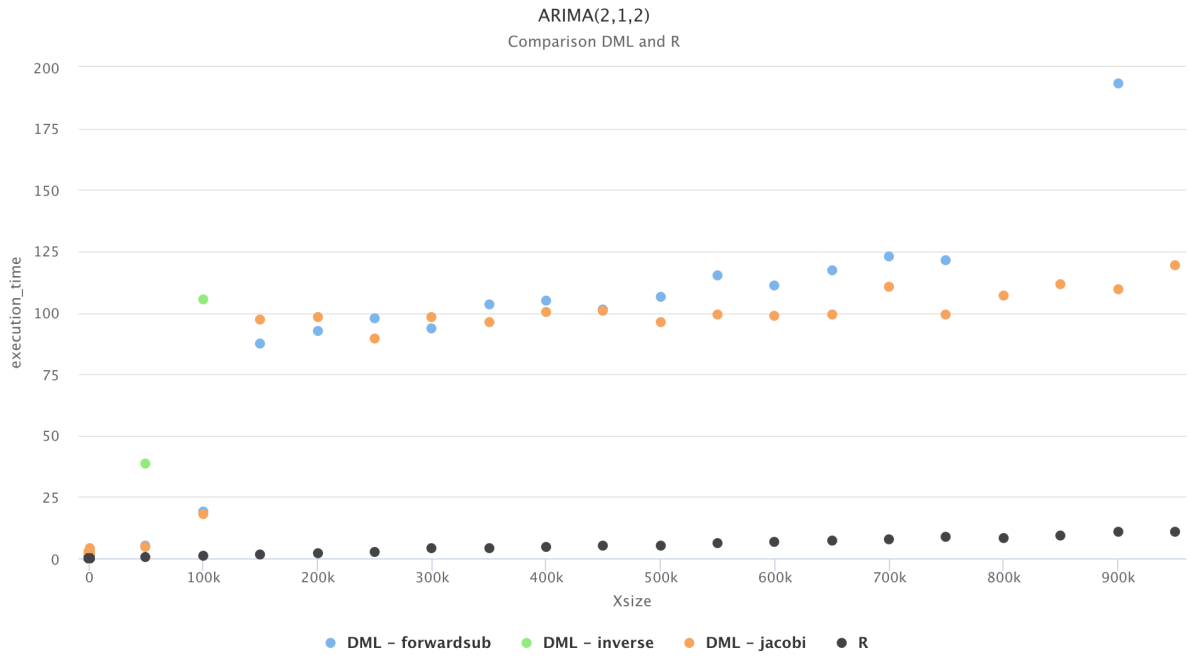

Figure 4.7.: **Execution time** of ARIMA(2,1,2) for DML and R for time series with sizes **from** $1,000$ **to** $950,000$ excluding outliers with an execution time greater than 200 seconds

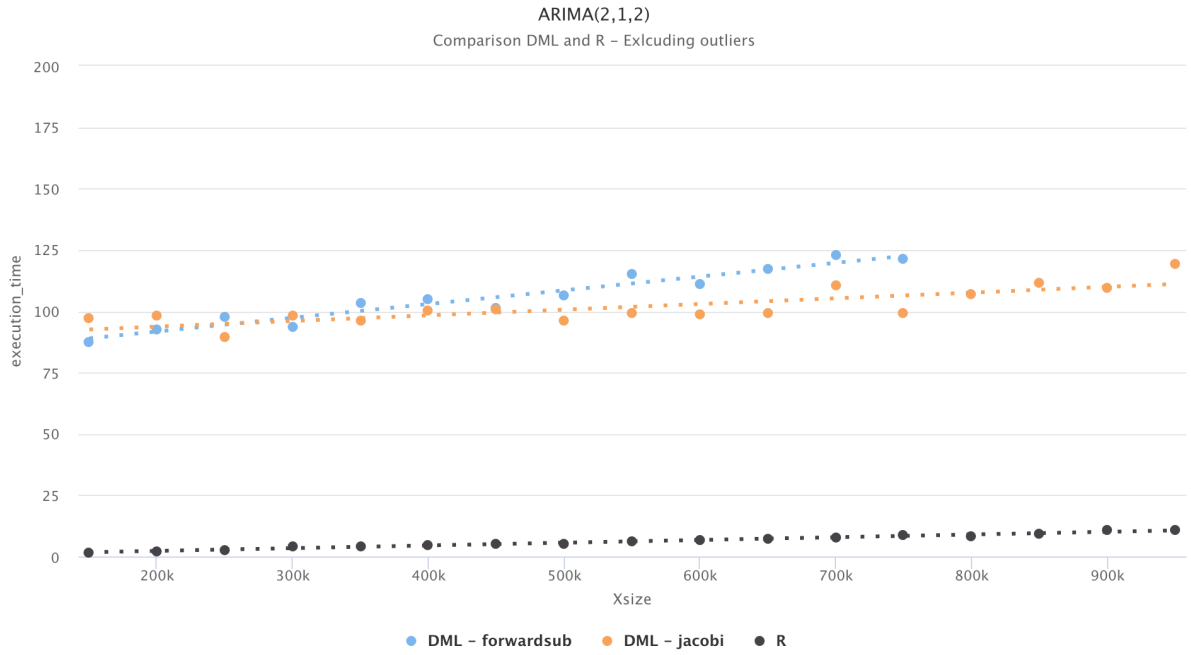

Figure 4.8.: **Execution time** of ARIMA(2,1,2) for DML and R for time series with sizes **from** $100,000$ **to** $950,000$ excluding outliers with an execution time greater than 150 seconds

5. Conclusion

The main goal of this research project was to optimize the ARIMA training algorithm for Apache SystemML. Or more precisely, to optimize the scoring function that ARIMA is based on. In chapter 2.1 the mathematical foundations of this function were discussed at length, including a closer look at the system of linear equations that needed to be solved for any process based on the Moving Average model. These linear systems were analyzed in more detail in chapter 2.3 which then also outlined and discussed a number of different approaches that could be used to solve these systems, of which the following three were finally chosen to be implemented: Namely the Jacobi solver, the Forward Substitution solver and the "Inverse" solver.

Chapter 3 explained the methods that were to be used to achieve the goal of optimizing ARIMA for SystemML. It explained why the scoring function was chosen to be optimized as well as why and how the linear system solvers where chosen. Another part of this chapter was the discussion of the implementation of ARIMA for SystemML using the Declarative Machine Learning Language and how its performance and precision should be measured. Especially the methods of measurement had a significant impact on the final results of this research project. Which is why the first section 5.1 of this chapter will be revisiting some of the decision that were made and evaluate them in terms of their effectiveness and efficiency.

The second section 5.2 will then proceed with the discussion of the results of the performance and precision tests that were presented previously in chapter 4 and try to explain why the results look the way they do, what might have caused irregularities and what the meaning of the results is with respect to the main goal of this research project.

And finally, the last section of this chapter will be addressing how the work can be continued as well as how and which parts of this project should be reworked or improved before doing so.

5.1. Methodology

The very first decision that hasn't even been documented as one in the methodology chapter was the decision to use the Declarative Machine Learning Language to implement the ARIMA training algorithm. Of course this was only because that decision had already

been made before even starting with the research project. Even though there were good reasons for making this decision (some of them described in the introduction chapter) there were also a number of downsides that started popping up as soon as the implementation work had begun. Especially having to use DML had its disadvantages: Debugging became a very painfully process, because of all the code optimizations that were going on under the hood. Additionally the debugger available could only be used from the terminal and had a rather small feature set. And even when used, always having the compiler optimize and restructure the code was making it particularly hard to understand what was happening. Of course one could also try to debug the code using print statements, but even when an error was not occurring before these statements sometimes they still weren't executed. This was a result out of the said code optimizations doing its magic, divided the code into segments which were run separately. And an error in one of those segments would stop all the other instructions in the same segment to be run too. Therefore the error messages were not only vague in the form of "error between lines 120 and 158" but all the print statements used for debugging within these lines were not executed as well and finding the cause of the error was made even more complicated. Of course this also led to the next problem when developing with DML. The optimizations were not only impacting the development in a negative manner, but they were also not well documented. And neither were the workarounds that made it possible to "cut" the sections manually into smaller pieces.

Furthermore, because the implementation of R's *stats* package was chosen to be the base line for verifying the correctness of the ARIMA implementation, using regression test to make sure that the DML implementation was indeed correct turned out to be rather complicated. One had to implement additional scripts only to separately run then read and finally compare the results. And quickly comparing intermediary results was not possible at all without adjusting both scripts. This made developing and debugging the script much harder. To overcome these issues, the script was first developed in native R and only after checking that the script is working properly it was ported to DML. Which is a process I would most definitely recommend for any further development using SystemML's Declarative Machine Learning Language. Even if one does not have to compare it to another R implementation. Being able to use R-Studio as a development environment increased the development speed significantly.

Additionally, the performance test should have been run sightly differently to make it easier to compare the results. The models that were chosen to be tested, were not considered carefully enough and most definitely not deliberately enough. The AR and MA models should have been chosen in a way that would have allowed comparisons between each one of those models with the full ARIMA model. That way one could have analyzed the influence of the AR or MA part on the ARIMA model. But because the models chosen

in the form of AR(3), MA(3) and ARIMA(2,1,2), no direct comparison could be done without relying on assumptions about the way an AR or MA model behaves in terms of execution time when increasing or decreasing the order of the model.

5.2. Results

First and foremost, the results have shown that the implementation of the ARIMA_{CSS} function with DML is correct and that it produces the exact same results as R's implementation for at least the first 12 decimal places. Of course there were some test runs that completely failed and didn't even produce any result. But first of all, these were only runs for the implementations using the Inverse solver which turned out to be the slowest anyways. And more importantly they were occurring, because the Hadoop Distributed File System had crashed and SystemML could therefore not read the test data.

With regards to the results of the performance tests for large scale time series there are three important takeaways:

First of all, *R is generally much faster than DML*, especially when it comes to calculating the ARIMA_{CSS} function for a model that needs to solve a system of linear equations. For the class of AR models this is not the case, which is one of the factors that allow the DML implementation to beat R's performance for these models. But only using the Jacobi solver. Which is particularly interesting considering that for AR models there isn't any need for solving a system of linear equations anyways. That we can still see a difference in the performance when using different solvers is due to the fact that there is no check whether we need to solve the linear system or not. The solver function is just called, even if the the coefficient matrix A is the identity matrix I and therefore b would already be equal to the solution vector x. For the Jacobi solver this doesn't have a huge impact performance-wise because the it finds the solution immediately in the first iteration. In contrast, the Forward Substitution solver runs the same amount of iterations no matter which values A has and always converges after exactly n iterations for an $n \times n$ coefficient matrix.

Second, when comparing the different solvers among each other, the Jacobi and Forward Substitution solver are ahead with a huge margin, with the Jacobi solver being slightly better than the Forward Substitution solver. But not by much and for the most part the difference isn't even statistically significant and does only appear to be better because of observations that can be clearly defined as outliers. The inverse solver on the other hand is so much slower than the other two, that it wasn't even worth discussing its performance in

detail. Using the inverse solver is therefore not recommended and it should be disregarded with completely.

The last takeaway is that there is indeed a clear pattern in the performance data for MA and ARIMA models. The execution time first starts to rise almost quadratically and then the underlying function suddenly changes and continues in the fashion of a linear model. Unfortunately the reason for this behavior is not known. It might have something to do with the way that SystemML decides at what point the algorithm is distributed on multiple nodes. Which would also explain the weird behavior of MA(3) model's execution time that can be seen in figures 4.2 and 4.5. In case of the first figure, where MA(3) and MA(6) are compared, one possible explanation for the rise and fall of MA(3)'s execution time could be that the script is first running only on one node, which is taking longer and longer to calculate the solutions for the bigger systems of linear equations, until SystemML's compiler estimates that splitting the computations on two nodes is worth the overhead making the execution time drop again.

The same explanation could also be applied to the outliers that can be seen in the comparison of the Autoregression models. However, the extremely high measurements of the Forward Substitution solver's execution time for ARIMA(2,1,2) that were shown in figure 4.6 can not be explained with this. They are most likely caused by external factors that impacted the cluster on which the tests were running directly.

5.3. Next Steps

In it self this research project was successfully finished and its goal to optimize the ARIMA training algorithm achieved. But obviously there is still room for improvement and further work that can be done to increase the performance of the script even further. A few things that should be improved have already been addressed and the first one that could also quite easily be done is re-running the performance test with a slightly different setup: The orders of AR, MA and the full ARIMA should then be set to the same set of values. And each test configuration should not only be run twice, but five to ten times, to get a more stable test result. With regards to the general set up of the tests there are also some other things that one could decide to invest more time into: For example running the tests with lower values for executor and driver memory of spark to see how the script would behave if it were to max out the memory space and at what point it does this. One could also change the execution mode of SystemML to "single-node" to see how the performance of the scripts would change if SystemML would not distribute the computation over multiple nodes.

Besides fiddling with the settings of the test environment and adding more test configurations or increasing the number of test runs, there's also the question of the optimizer still to be answered. Because even though it was pointed out in chapter 3 that optimizing the scoring function promises to give better result, choosing the right optimizer can still have a significant impact on the performance and on the precision of the results. Or even on the question whether a solution will be found at all. Answering this question alone is definitely worth investing a considerable amount of time.

And finally there are also a lot more variations of the ARIMA algorithm that could be implemented. There is the seasonal Autoregressive Integrated Moving Average model as well as the Autoregressive Moving Average with exogenous inputs model and of course one could also add an additional parameter to specify the mean or intercept term or add another parameter for the including a drift term.

A. Appendix

A collection of tables and plots describing the results of the performance tests that didn't make it into the results section, but might still be of some interest to the reader.

A.1. Tables

The following tables try to give an overview of the distribution and statistical properties of the performance measurements.

Stat.	AR	MA	ARIMA	AR	MA	ARIMA
	small scale			large scale		
min	0.67	0.56	0.6	19.53	166.25	79.16
max	1.48	2.32	1.57	1,358.25	20,409.55	505.1
mean	1.16	1.39	1.17	571.86	7,521.83	292.13
median	1.17	1.43	1.16	467.69	6,773.76	292.13
standard deviation	0.26	0.47	0.3	462.75	7,116.69	301.18

Table A.1.: Statistical analysis of **DML execution time** for **small scale** (from 50 to 1,000) and **large scale** (from 50,000 to 950,000) time series for **Inverse solver**

Stat.	AR(3)	AR(6)	MA(3)	MA(6)	ARIMA(2,1,2)	ARIMA(4,1,4)
min	0.55	1.23	1.25	0.59	0.6	0.66
max	1,518.45	1,614.59	20,409.55	11,357.28	4,392.84	8,072.99
mean	186.55	190.82	1,299.8	852.12	217.49	2,266.97
median	15.41	12.46	165.54	294.95	99.32	1,617.87
standard deviation	372.98	387.5	3,765.94	2,145.01	668.53	2,176.59

Figure A.1.1.: Statistical analysis of **DML execution time** for time series with sizes **from** 50,000 **to** 950,000 for all models and **averaged for all solvers**

Stat.	AR(3)	AR(6)	MA(3)	MA(6)	ARIMA(2,1,2)	ARIMA(4,1,4)
min	2.24	3.15	2.88	2.39	2.28	2.63
max	1,520.21	1,616.31	20,411.22	11,361.3	4,394.45	8,077.31
mean	189.3	193.34	1,302.12	854.12	219.51	2,269.98
median	20.61	15.25	167.48	296.85	101.5	1,619.6
standard deviation	372.71	387.4	3,765.92	2,145.27	668.6	2,177.28

Figure A.1.2.: Statistical analysis of **DML run time** for time series with sizes **from** 50,000 **to** 950,000 for all models and **averaged for all solvers**

Stat.	AR(3)	AR(6)	MA(3)	MA(6)	ARIMA(2,1,2)	ARIMA(4,1,4)
min	0.06	0.06	0.08	0.07	0.05	0.07
max	13.89	12.17	3.16	3.26	11.29	5.19
mean	5.87	5.66	1.35	1.46	5.01	1.78
median	6.29	5.88	1.21	1.28	5.15	1.66
standard deviation	3.65	3.43	0.88	0.96	3.38	1.39

Figure A.1.3.: Statistical analysis of **R execution time** for time series with sizes **from** 50,000 **to** 950,000 for all models

Stat.	AR(3)	AR(6)	MA(3)	MA(6)	ARIMA(2,1,2)	ARIMA(4,1,4)
min	6.44	6.61	6.36	6.4	6.22	6.25
max	32.55	23.12	15.15	12.45	22.85	33.36
mean	14.93	14.12	9.19	8.9	12.22	11.5
median	14.18	14.47	8.88	8.68	12.37	8.8
standard deviation	6.4	4.23	2.04	1.48	3.84	6.15

Figure A.1.4.: Statistical analysis of **R run time** for time series with sizes **from** 50,000 **to** 950,000 for all models

Stat.	AR(3)	AR(6)	MA(3)	MA(6)	ARIMA(2,1,2)	ARIMA(4,1,4)
min	0.48	0.44	0.47	0.44	0.46	0.47
max	3.07	3.89	3.96	5.59	3.88	4.65
mean	1.04	1.25	1.43	1.36	1.12	1.38
median	0.91	1.22	1.25	1.0	0.87	1.12
standard deviation	0.55	0.72	0.88	1.06	0.73	0.95

Figure A.1.5.: Statistical analysis of **DML execution time** for time series with sizes **from** 50 **to** 1,000 for all models and **averaged for all solvers**

Stat.	AR(3)	AR(6)	MA(3)	MA(6)	ARIMA(2,1,2)	ARIMA(4,1,4)
min	2.13	2.1	2.14	2.21	2.17	2.19
max	11.82	14.13	10.26	16.93	13.63	14.12
mean	3.68	4.69	4.26	4.19	3.87	4.26
median	3.07	3.24	3.4	3.05	2.75	2.94
standard deviation	2.0	2.84	2.19	2.79	2.76	2.97

Figure A.1.6.: Statistical analysis of **DML run time** for time series with sizes **from** 50 **to** 1,000 for all models and **averaged for all solvers**

Stat.	AR(3)	AR(6)	MA(3)	MA(6)	ARIMA(2,1,2)	ARIMA(4,1,4)
min	0.05	0.05	0.05	0.05	0.05	0.05
max	0.23	0.2	0.2	0.21	0.25	0.18
mean	0.07	0.08	0.08	0.08	0.07	0.08
median	0.06	0.06	0.07	0.06	0.06	0.06
standard deviation	0.03	0.04	0.04	0.04	0.04	0.04

Figure A.1.7.: Statistical analysis of **R execution time** for time series with sizes **from** 50 **to** 1,000 for all models

Stat.	AR(3)	AR(6)	MA(3)	MA(6)	ARIMA(2,1,2)	ARIMA(4,1,4)
min	6.31	6.4	6.36	6.4	6.22	6.25
max	29.78	25.52	24.77	24.02	25.99	20.82
mean	9.52	10.07	9.99	9.59	9.46	9.36
median	7.33	7.58	7.89	6.86	6.77	6.73
standard deviation	4.4	4.65	4.15	4.56	5.03	4.4

Figure A.1.8.: Statistical analysis of **R run time** for time series with sizes **from** 50 **to** 1,000 for all models

Stat.	AR(3)	AR(6)	MA(3)	MA(6)	ARIMA(2,1,2)	ARIMA(4,1,4)
min	0.47	0.5	0.46	1.27	20.4	10.09
max	2.32	3.25	3.16	10.64	340.35	4,091.19
mean	1.08	1.54	1.42	5.15	235.05	1,356.96
median	1.06	1.23	1.04	4.46	253.88	1,006.21
standard deviation	0.45	0.89	0.87	2.89	72.9	1,159.36

Figure A.1.9.: Statistical analysis of **DML execution time** using the **Jacobi solver** for time series with sizes **from** 50,000 **to** 950,000 for all models

Stat.	AR(3)	AR(6)	MA(3)	MA(6)	ARIMA(2,1,2)	ARIMA(4,1,4)
min	0.55	1.29	2.37	0.59	0.6	0.66
max	46.58	46.86	374.35	359.16	4,392.84	8,005.76
mean	18.73	16.81	166.46	263.84	372.65	2,332.06
median	17.84	13.25	156.63	292.08	105.56	1,568.24
standard deviation	13.62	12.74	97.42	99.62	968.93	2,102.71

Figure A.1.10.: Statistical analysis of **DML execution time** using the **Forward substitution solver** for time series with sizes **from** 50,000 **to** 950,000 for all models

Stat.	AR(3)	AR(6)	MA(3)	MA(6)	ARIMA(2,1,2)	ARIMA(4,1,4)
min	1.41	1.38	2.88	1.46	1.38	1.41
max	1,518.45	1,614.59	20,409.55	11,357.28	105.24	904.95
mean	535.68	550.99	6,935.22	4,202.66	48.46	341.97
median	427.5	435.29	5,376.61	2,710.76	38.77	119.55
standard deviation	489.11	511.07	7,184.54	4,452.18	52.61	491.12

Figure A.1.11.: Statistical analysis of **DML execution time** using the **Inverse solver** for time series with sizes **from** $50,000$ **to** $950,000$ for all models

Stat.	AR(3)	AR(6)	MA(3)	MA(6)	ARIMA(2,1,2)	ARIMA(4,1,4)
min	0.48	0.44	0.47	0.49	0.46	0.47
max	2.36	2.58	3.96	4.87	3.88	4.65
mean	0.98	1.18	1.51	1.56	1.28	1.57
median	0.57	1.04	1.05	0.93	0.69	1.03
standard deviation	0.62	0.73	1.03	1.29	1.08	1.31

Figure A.1.12.: Statistical analysis of **DML execution time** using the **Jacobi solver** for time series with sizes **from** 50 **to** $1,000$ for all models

Stat.	AR(3)	AR(6)	MA(3)	MA(6)	ARIMA(2,1,2)	ARIMA(4,1,4)
min	0.52	0.47	0.47	0.44	0.5	0.49
max	3.07	3.89	3.4	5.59	2.51	3.83
mean	1.0	1.39	1.24	1.32	0.91	1.4
median	0.66	1.1	0.74	0.66	0.59	0.99
standard deviation	0.69	0.98	0.94	1.28	0.58	0.96

Figure A.1.13.: Statistical analysis of **DML execution time** using the **Forward substitution solver** for time series with sizes **from** 50 **to** $1,000$ for all models

Stat.	AR(3)	AR(6)	MA(3)	MA(6)	ARIMA(2,1,2)	ARIMA(4,1,4)
min	0.64	0.57	0.58	0.54	0.6	0.59
max	1.5	1.54	2.98	1.72	1.58	1.57
mean	1.15	1.18	1.56	1.21	1.17	1.18
median	1.1	1.26	1.45	1.33	1.14	1.17
standard deviation	0.27	0.28	0.64	0.35	0.29	0.31

Figure A.1.14.: Statistical analysis of **DML execution time** using the **Inverse solver** for time series with sizes **from** 50 **to** 1,000 for all models

A.2. Graphical representation

The representation using scatter plots was chosen to show a more detailed view of the performance measurement data:

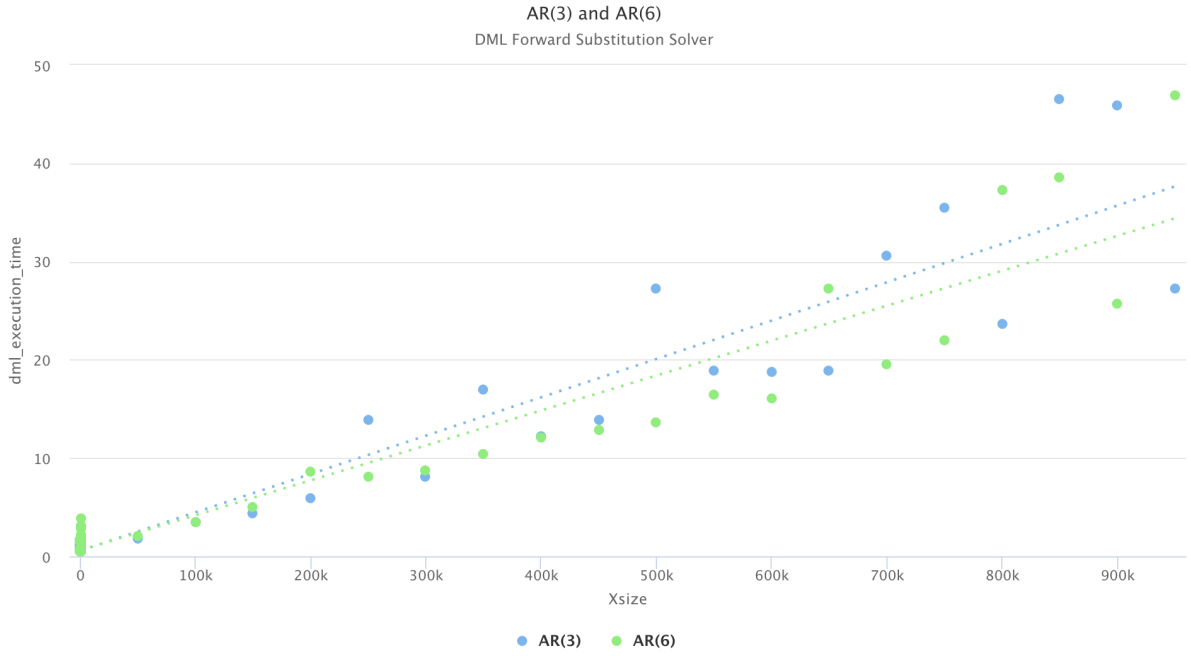

Figure A.2.1.: Comparison of **Execution time** of **AR(3) and AR(6) for Forward Substitution DML** for time series with sizes **from** $1,000$ **up to** $950,000$

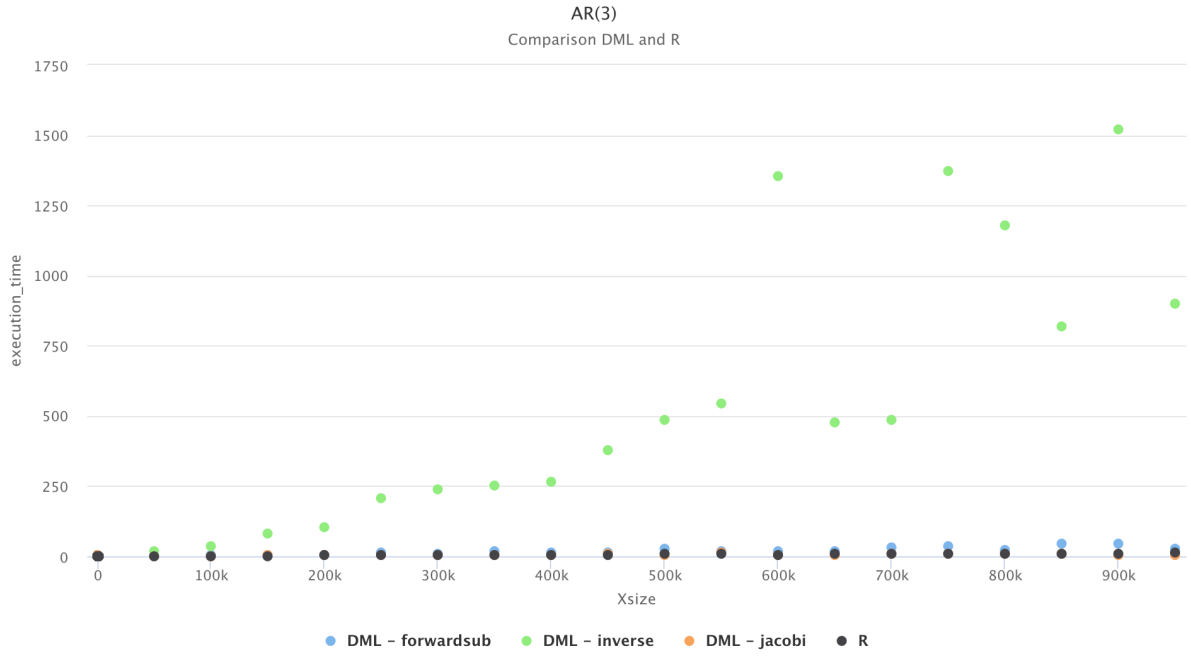

Figure A.2.2.: **Execution time** of AR(3) for DML and R for time series with sizes **up to** 950,000

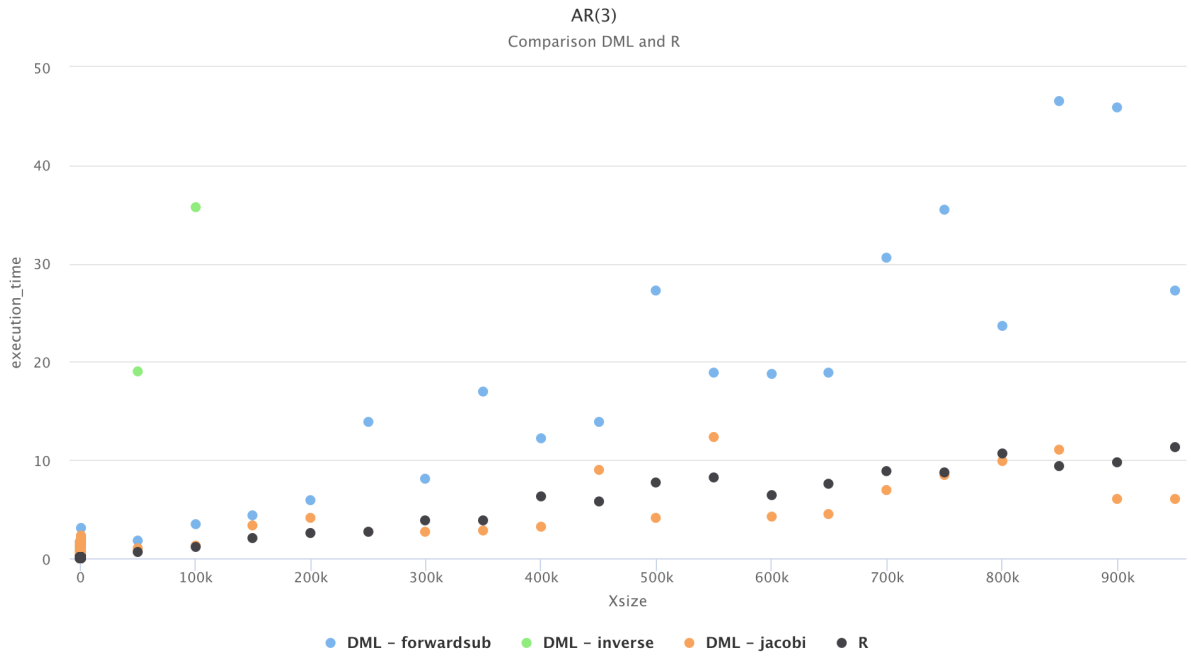

Figure A.2.3.: **Execution time** of AR(3) for DML and R for time series with sizes **up to** 950,000 but on a reduced time scale limited to

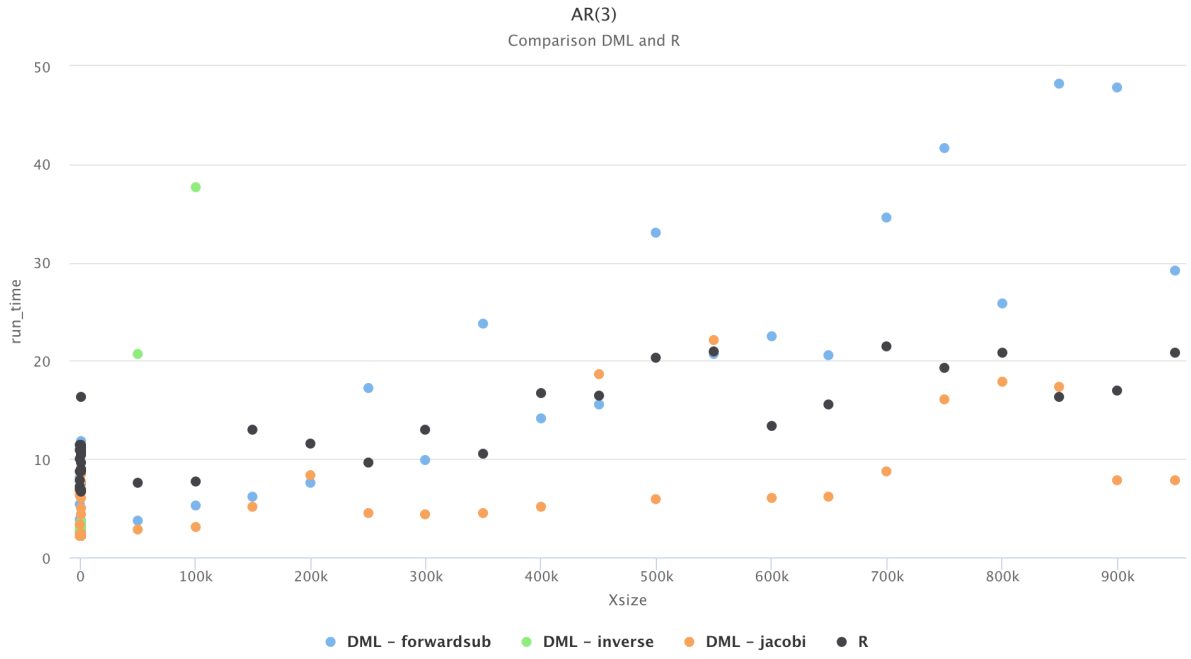

Figure A.2.4.: **Run time of** AR(3) for DML and R for time series with sizes **up to** 950, 000

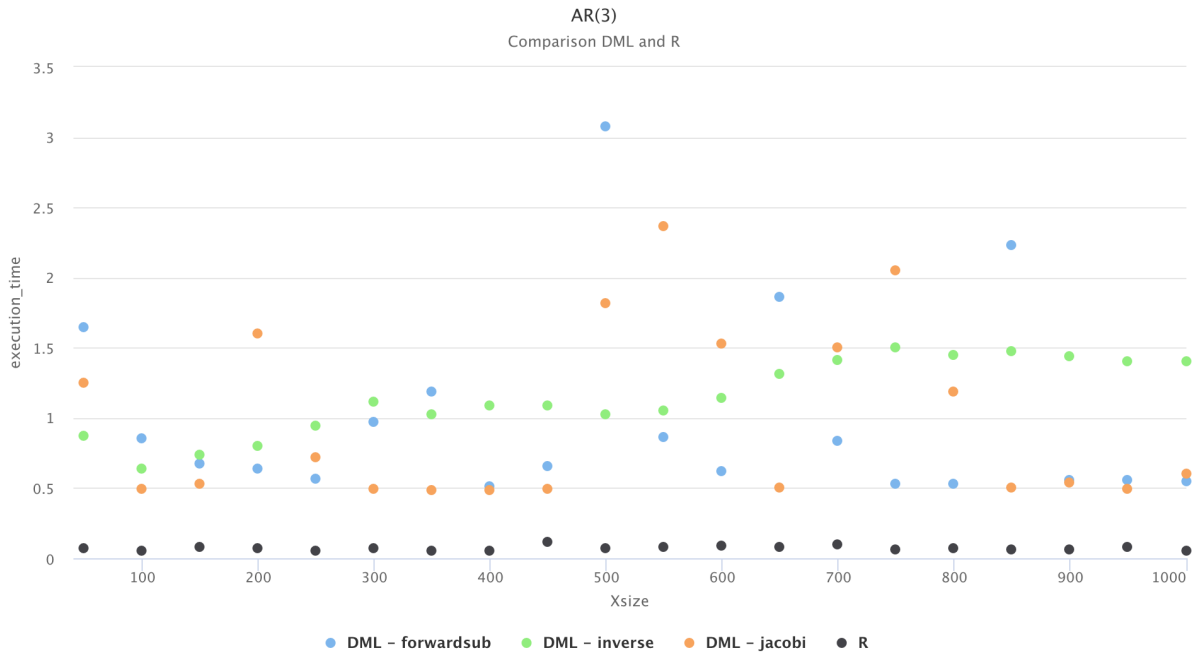

Figure A.2.5.: **Execution time** of AR(3) for DML and R for time series with sizes **from** 50 **to** 1, 000

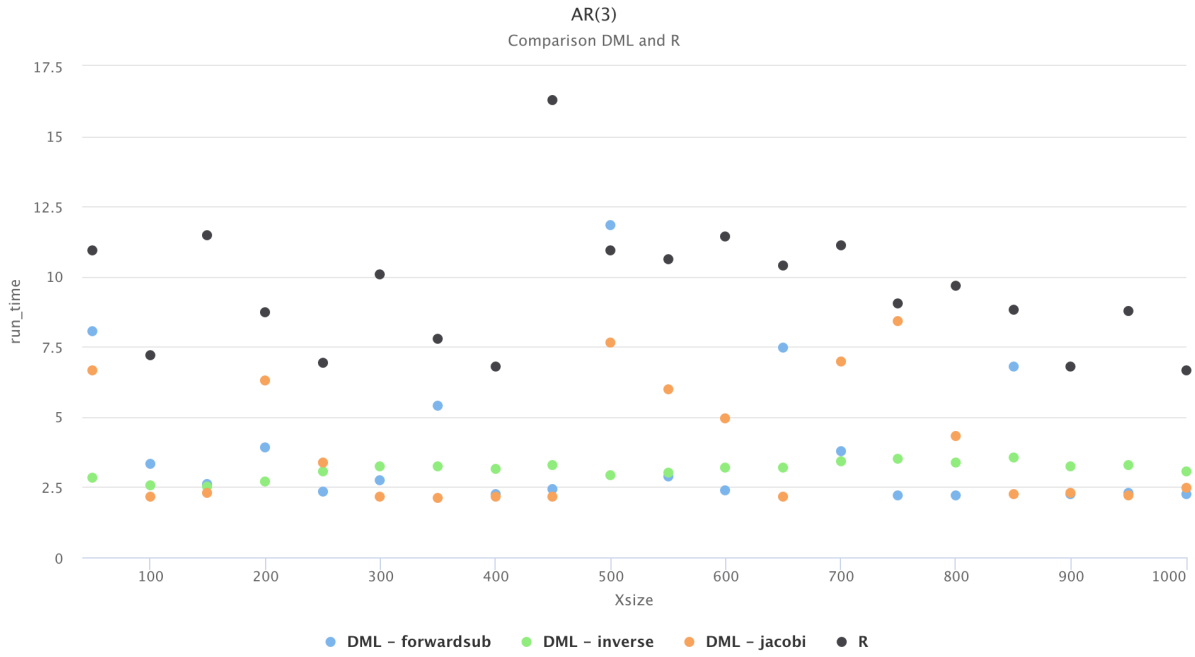

Figure A.2.6.: **Run time** of AR(3) for DML and R for time series with sizes **from** 50 **to** 1,000

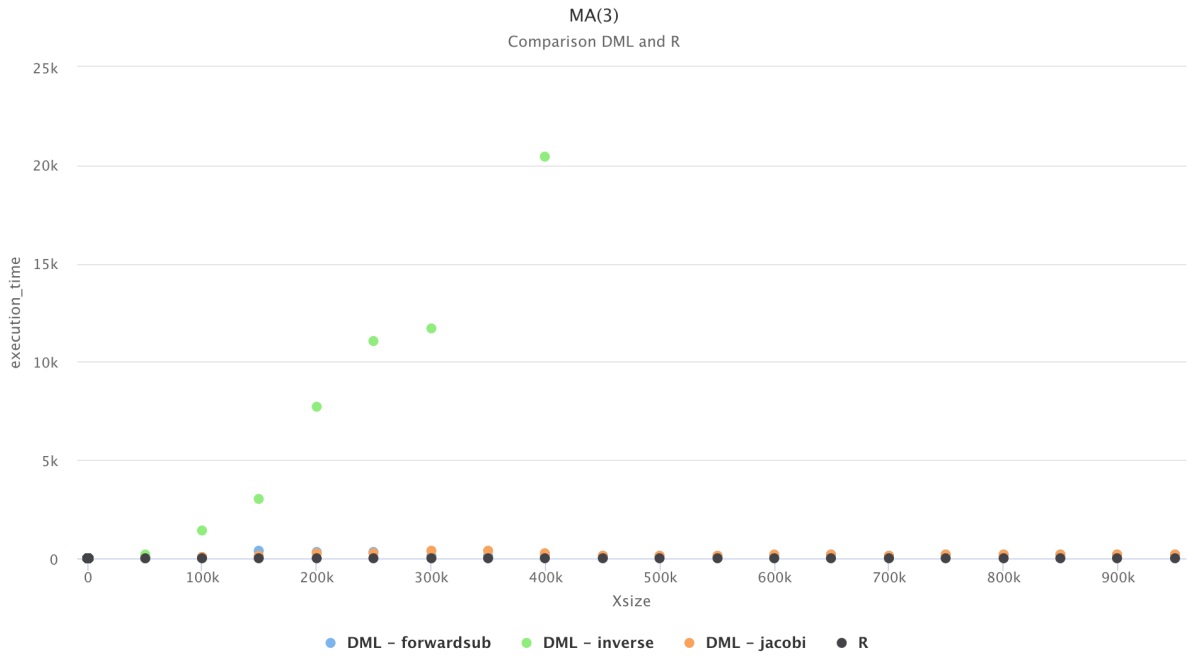

Figure A.2.7.: **Execution time** of MA(3) for DML and R for time series with sizes **up to** 950,000

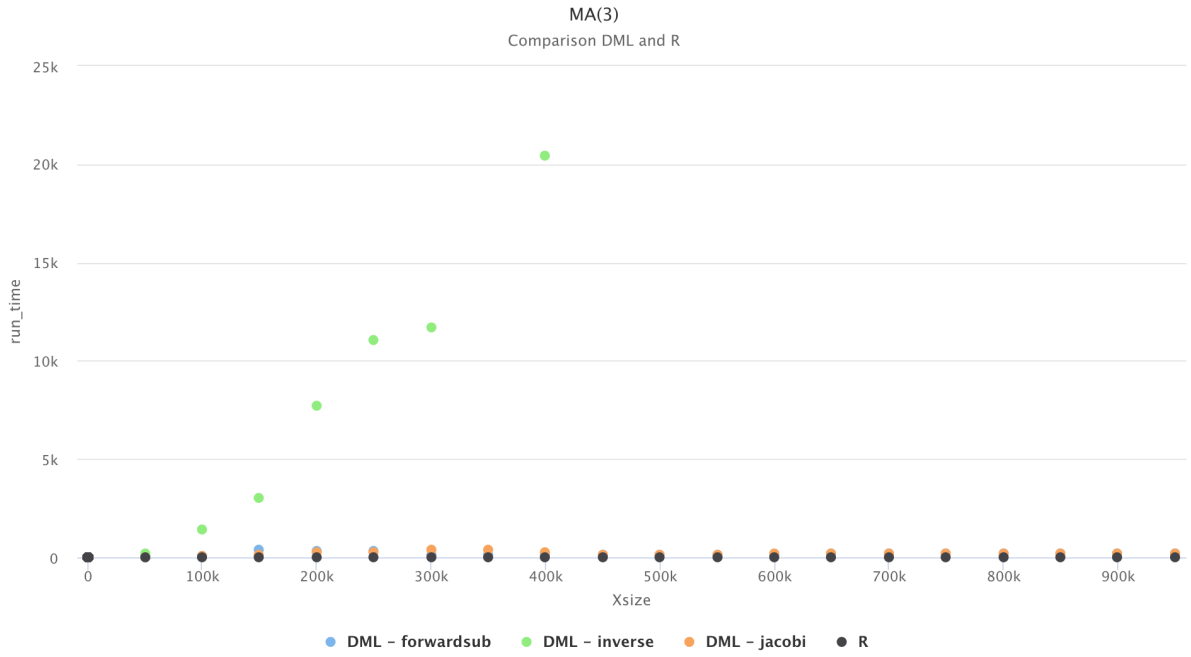

Figure A.2.8.: **Run time of** MA(3) for DML and R for time series with sizes **up to** 950,000

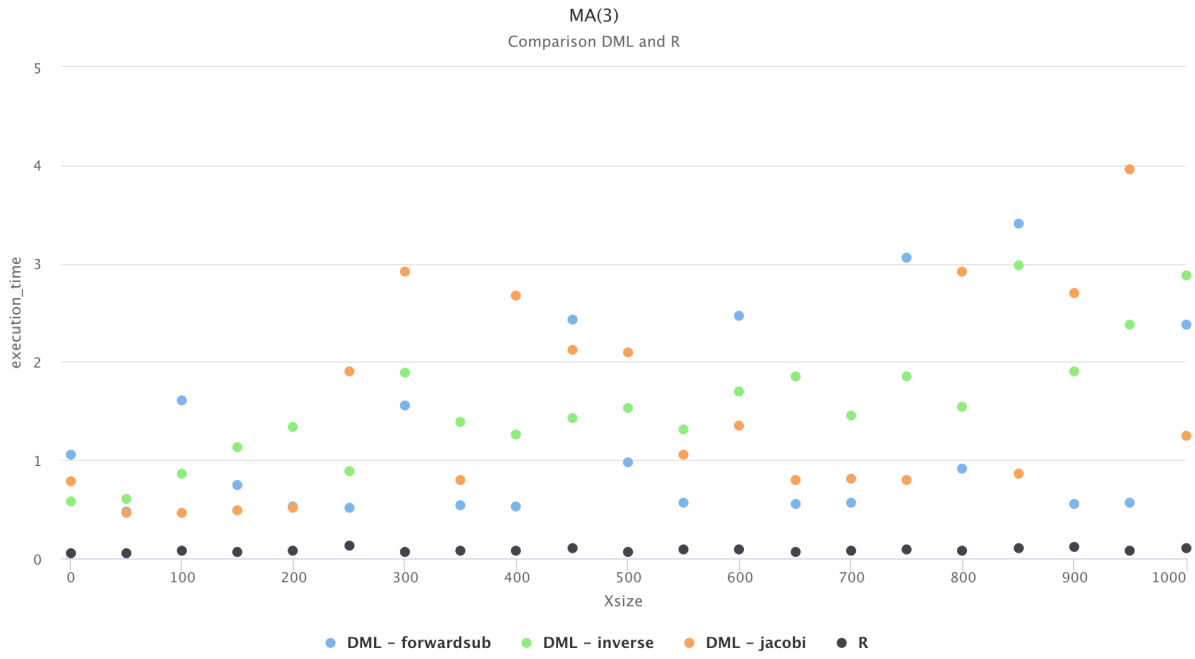

Figure A.2.9.: **Execution time** of MA(3) for DML and R for time series with sizes **from** 50 **to** 1,000

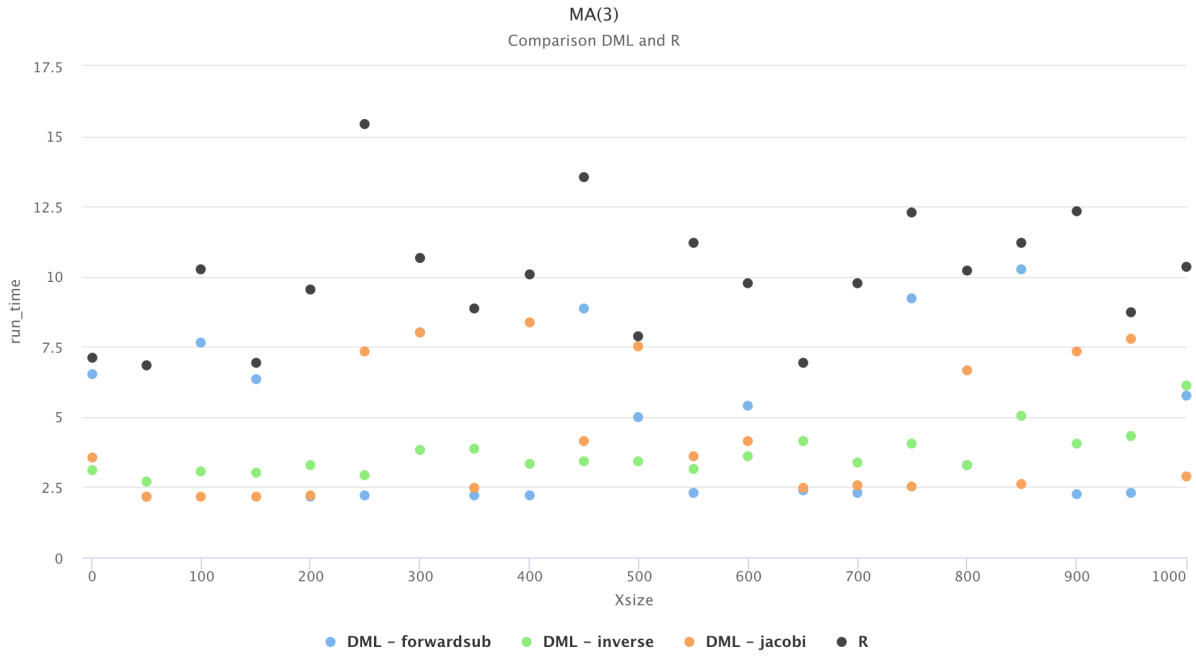

Figure A.2.10.: **Run time** of MA(3) for DML and R for time series with sizes **from** 50 **to** 1,000

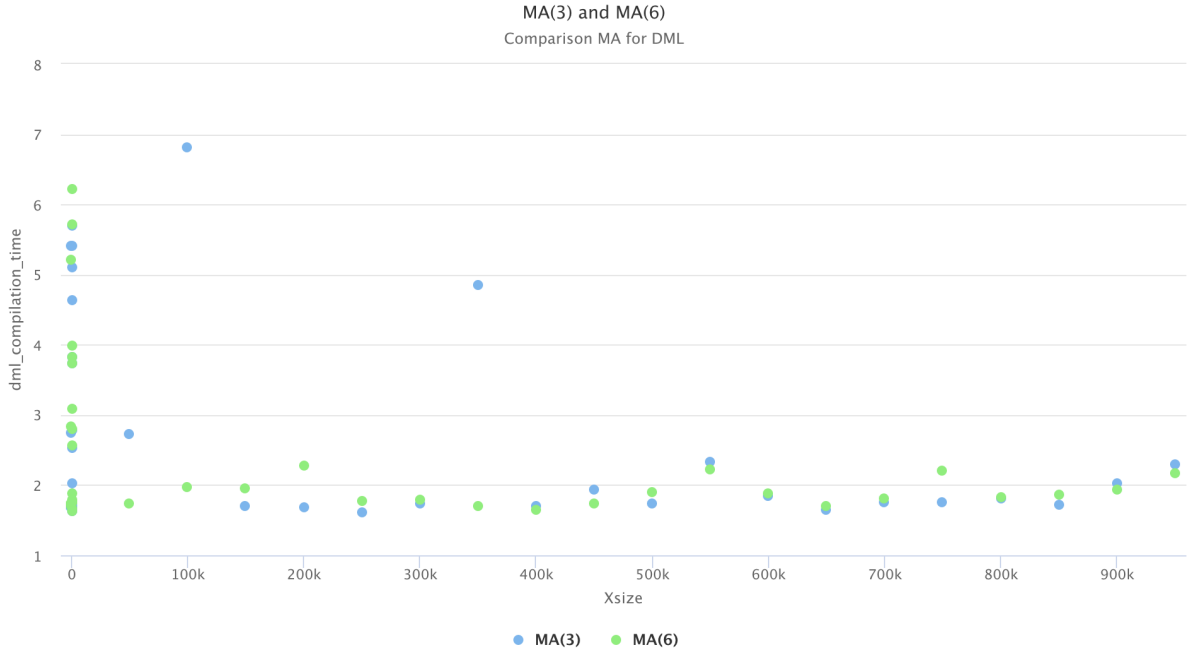

Figure A.2.11.: Comparison of DML **execution time** of MA(3) and MA(6) - *average of all solvers*

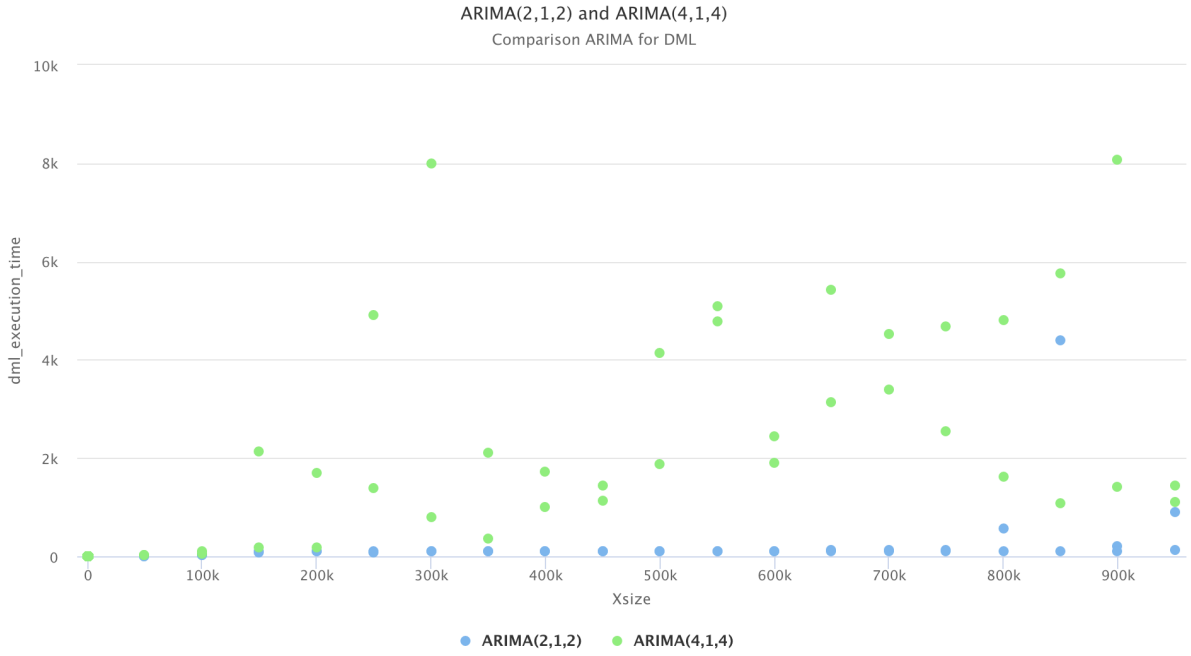

Figure A.2.12.: Comparison of DML **execution time** of ARIMA(2,1,2) and ARIMA(4,1,4) - *average of all solvers*

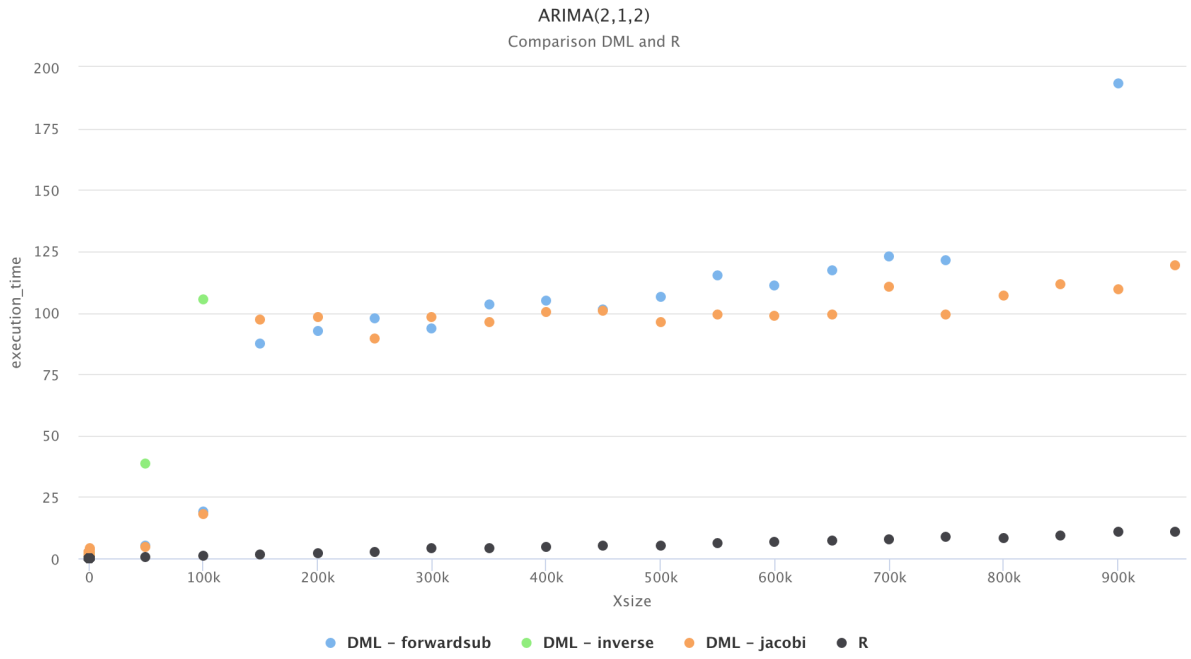

Figure A.2.13.: **Execution time** of ARIMA(2,1,2) for DML and R for time series with sizes **up to** 950,000

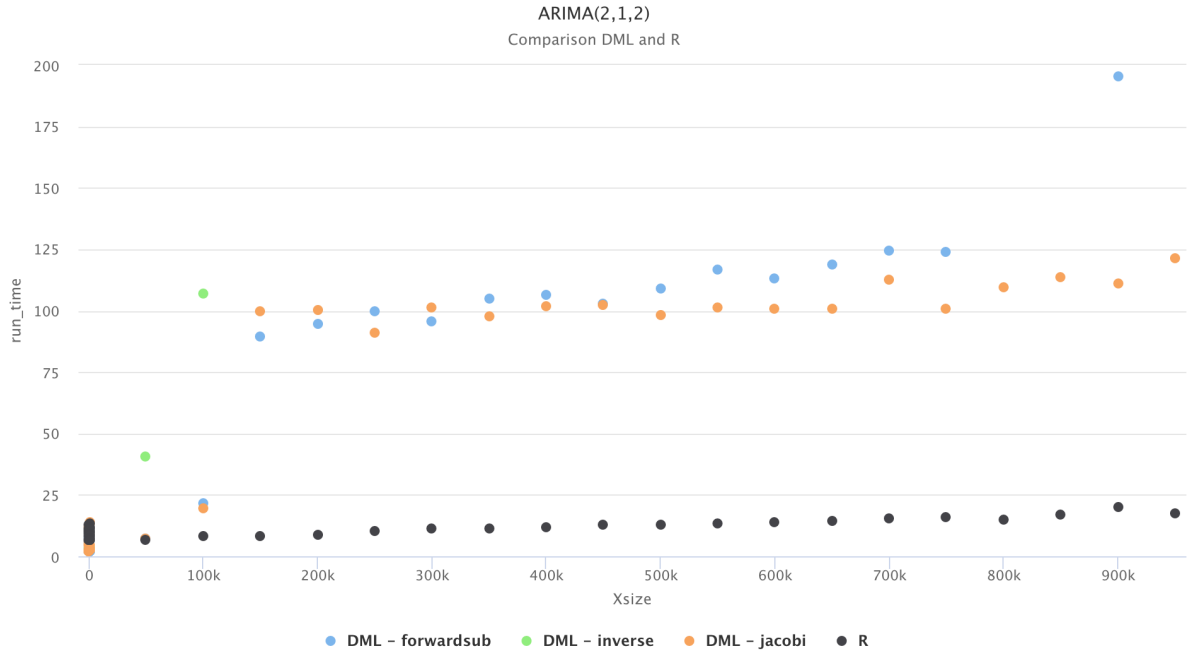

Figure A.2.14.: **Run time of** ARIMA(2,1,2) for DML and R for time series with sizes **up to** 950,000

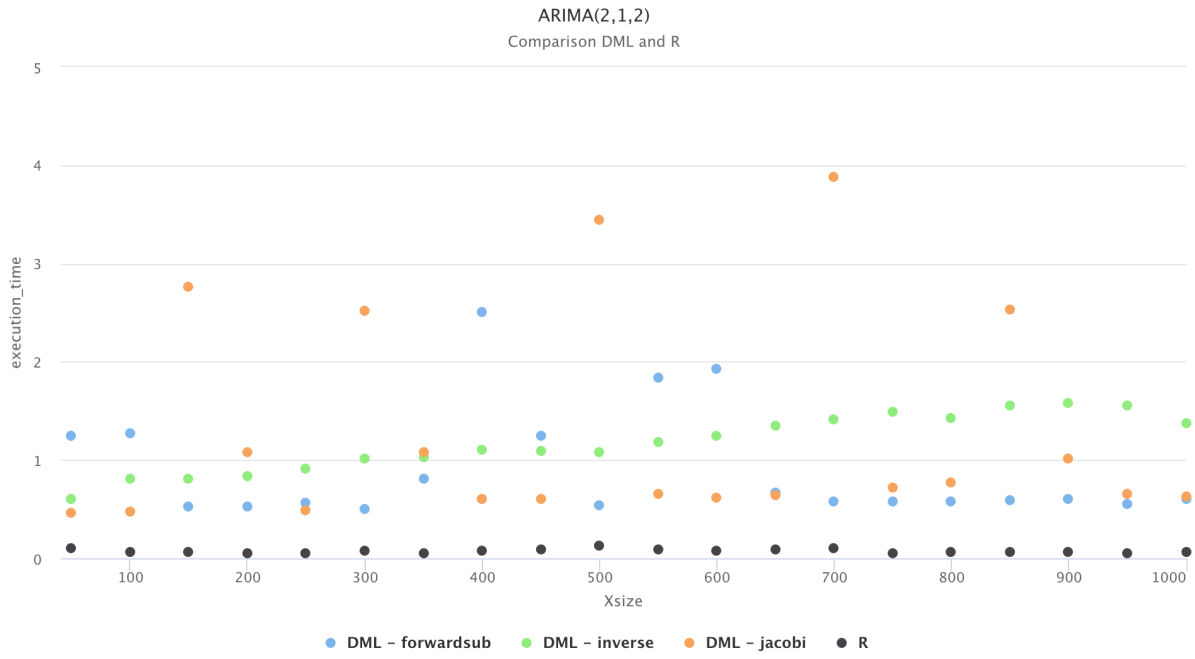

Figure A.2.15.: **Execution time** of ARIMA(2,1,2) for DML and R for time series with sizes **from** 50 **to** 1,000

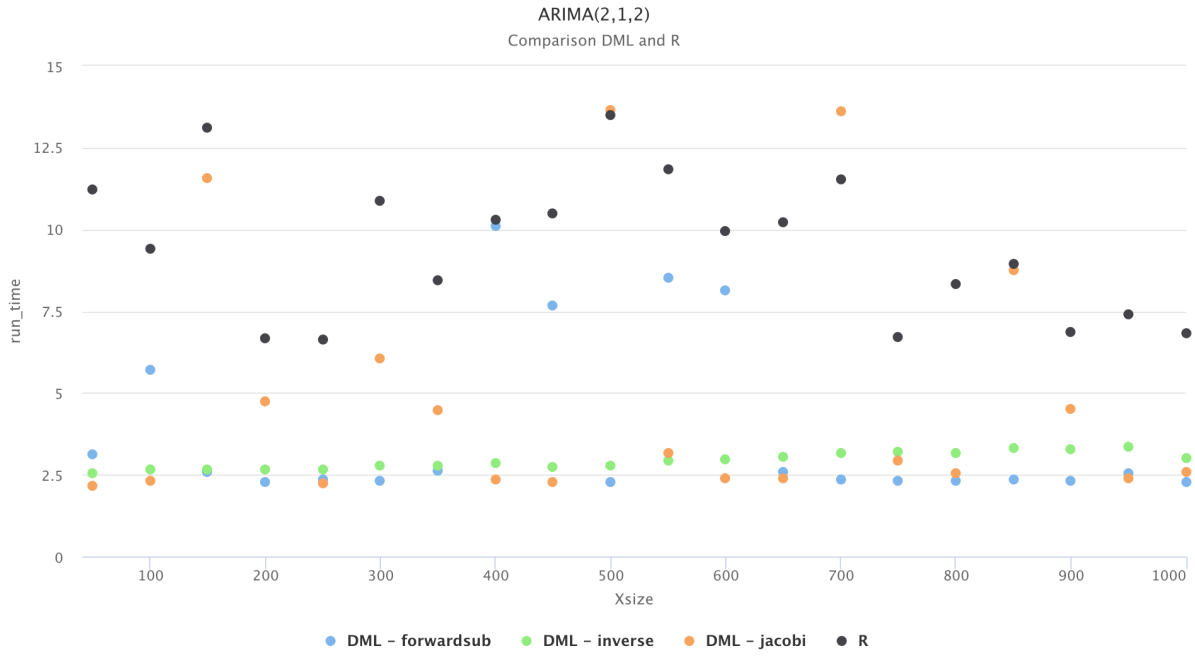

Figure A.2.16.: **Run time** of ARIMA(2,1,2) for DML and R for time series with sizes **from** 50 **to** 1,000

Acronyms

NASA National Aeronautics and Space Administration

SMAP Soil Moisture Active Passive

ARIMA Autoregressive Integrated Moving Average

ARMA Autoregressive Moving Average

ARMAX Autoregressive Moving Average with exogenous inputs

AR Autoregression

MA Moving Average

ML Maximum Likelihood

DAG directed acyclic graph

HOP high-level operation

LOP low-level operation

BFGS Broyden-Fletcher–Goldfarb-Shanno

L-BFGS Limited-memory Broyden-Fletcher–Goldfarb-Shanno

CG Conjugate Gradient

Bi-CG Biconjugate Gradient

CGS Conjugate Gradient Squared

Bi-CGSTAB Biconjugate Gradient Stabilized

GS Gauss-Seidel

SOR Successive Over-Relaxation

CSS Conditional Sum of Squares

DML Declarative Machine Learning Language

iid independent and identically distributed

API Application Programming Interface

JVM Java Virtual Machine

HDFS Hadoop Distributed File System

CSV Comma Seperated Values

Bibliography

[1] ALGLib. *Unconstrained optimization: L-BFGS and CG*. URL: http://www.alglib.net/optimization/lbfgsandcg.php.

[2] Saranya Anandh. *Everything About Time Series Analysis And The Components of Time Series Data*. 2016. URL: https://www.linkedin.com/pulse/everything-time-series-analysis-components-data-saranya-anandh/.

[3] Claude J P Bélisle. "Convergence theorems for a class of simulated annealing algorithms on d". In: *Journal of Applied Probability* 29.4 (1992), pp. 885–895. ISSN: 0021-9002. DOI: DOI:10.2307/3214721. URL: https://www.cambridge.org/core/article/convergence-theorems-for-a-class-of-simulated-annealing-algorithms-on-d/FCC646C64A013B306AEB5D822BDBE9D1.

[4] Noel Black and Shirley Moore. *Gauss-Seidel Method*. URL: http://mathworld.wolfram.com/Gauss-SeidelMethod.html.

[5] Noel Black and Shirley Moore. *Stationary Iterative Method*. URL: http://mathworld.wolfram.com/StationaryIterativeMethod.html.

[6] Noel Black and Shirley Moore. *Successive Overrelaxation Method*. URL: http://mathworld.wolfram.com/SuccessiveOverrelaxationMethod.html.

[7] Matthias Boehm et al. "SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs". In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* (2014).

[8] Peter Brockwell and Richard Davis. *Introduction to Time Series and Forecasting 2nd*. 2002. ISBN: 0387953515. DOI: 10.2307/1271510.

[9] Voker John. "Chapter 3 - Classical Iterative Schemes". In: *Lecture Notes on Iterative Methods for Linear Systems of Equations*. URL: https://www.wias-berlin.de/people/john/LEHRE/NUMERIK_II_18_19/linsys_3.pdf.

[10] NASA. *SMAP - Soil Miosture Active Passive: Data Products*. URL: https://smap.jpl.nasa.gov/data/.

[11] NASA. *SMAP - Soil Moisture Active Passive: Specifications*. URL: https://smap.jpl.nasa.gov/observatory/specifications/.

[12] Mike Rambo. "The Conjugate Gradient Method for Solving Linear Systems of Equations". 2016.

[13] Karl Stroetmann. “Iterative Lösung linearer Gleichungs-Systeme”. In: *Skript: Analysis*. 2017. Chap. Anwendunge. URL: https://github.com/karlstroetmann/Analysis/blob/master/Skript/analysis.pdf.

[14] *SystemML Documentation*. URL: http://apache.github.io/systemml/.

[15] R-Core Team. *R Documentation: General-purpose Optimization*. URL: https://stat.ethz.ch/R-manual/R-devel/library/stats/html/optim.html.

[16] Yuvashankar Vinay, Sayari Nejad Mojdeh, and Liu Ash (Chang). “Understanding the Bi Conjugate Gradient Stabilized Method (Bi-CGSTAB)”. 2016.