Scalable Time Series Forecasting with ARIMA for SystemML

PROJECT REPORT

from the Course of Applied Computer Science

at the Cooperative State University Baden-Württemberg Mannheim

by

Tobias Schmidt

August 28th, 2018

Time of Project	May 7th 2018 to August 1th 2018
Student ID, Course	47478088, TINF16AI-BC
Division/Business Unit	Research
Company	IBM Research - Almaden, San Jose
Supervisor in the Company	Berthold Reinwald

signature Supervisor _____

Contents

Declaration

Hereby I solemnly declare that my project report, titled *Scalable Time Series Forecasting with ARIMA for SystemML* is independently authored and no sources other than those specified have been used.
I also declare that the submitted electronic version matches the printed version.

Mannheim, August 28th, 2018 _____

Place, Date Signature Tobias Schmidt

Abstract - English

SystemML is a Machine Learning Platform built for large scale analytics, that enables flexible and scalable machine learning while also accelerating exploratory algorithm development. It is providing a solution to automatically scale any algorithm to run efficiently and fast on big data. To simplify and speed up the work of data scientists it is already providing a number of machine learning algorithms to solve classifications, clustering and regression problems. But it is not yet providing any built-in solution to analyze time series data.

However, because there are so many applications for time series analysis, the demand for models that are able to fit and predict time series data is high.

The ARIMA model is one of the best known time series models and used to better understand and forecast time series data. In contrast to the ARMA model, it can also handle data that shows evidence of non-stationarity and can therefore be applied to more use-cases.

This report describes what time series data is and explains the different characteristics it can exhibit, such as noise, trend, cyclic and seasonal movements, pulses and steps, and stationarity. It also covers the basics of ARIMA, first explaining what Auto-regression and Moving Average models are and then describing differencing and how this can be used to analyze non-stationarity. After that the differences between additive and multiplicative seasonal ARIMA models are also discussed.

Besides the ARIMA model itself the report also explains the BFGS optimizer and the Jacobi solver which are needed to fit an ARIMA model.

To explain how the ARIMA training algorithm can be applied on big data, SystemML and in particular its optimization methods are briefly described.

Finally the implementation of ARIMA using the Declarative Machine Learning Language is discussed.

By adding the mentioned ARIMA algorithm to the list of algorithms already implemented for SystemML, data scientist can be enabled to easily work with time series data on large scale.

Abstract - German

SystemML ist eine Machine-Learning-Plattform, die für umfangreiche Analysen entwickelt wurde, welche flexibles und skalierbares maschinelles Lernen ermöglicht und gleichzeitig die explorative Entwicklung von Algorithmen beschleunigt. Es bietet eine Lösung, um jeden Algorithmus automatisch so zu skalieren, dass er schnell und effizient mit Big Data läuft. Um die Arbeit von Data-Scientists zu vereinfachen und zu beschleunigen, bietet SystemML bereits eine Reihe von machine learning algorithmen zur Lösung von Klassifizierungs-, Clustering- und Regressionsproblemen an. Es bietet jedoch noch keine fertige Lösung zur Analyse von Time Series Daten.

Da es jedoch viele Anwendungen für Time Series Daten gibt, ist die Nachfrage nach Modellen, die solche vorhersagen können, hoch.

Das ARIMA-Modell ist eines der bekanntesten Time Series Modelle und wird verwendet, um Time Series Daten besser zu verstehen und vorherzusagen. Im Gegensatz zum ARMA-Modell kann es auch mit Daten umgehen, die Anzeichen von Nichtstationarität aufweisen und daher auf mehr Anwendungsfälle angewendet werden können.

Dieser Bericht beschreibt, was Time Series Daten sind und erklärt die verschiedenen Merkmale, die diese aufweisen können, wie bspw. Noise, Trend, zyklische und saisonabhänige Bewegungen, Impulse und Stufen sowie Stationarität. Es behandelt auch die Grundlagen von ARIMA, erläutert zuerst, was Autoregressions- und Moving-Average-Modelle sind und beschreibt dann den Unterschied und wie dies zur Analyse von Nichtstationaren Daten verwendet werden kann. Danach werden die Unterschiede zwischen additiven und multiplikativen saisonalen ARIMA Modellen diskutiert.

Neben dem ARIMA-Modell erläutert der Bericht auch den BFGS-Optimierer und den Jacobi-Solver, die für ein ARIMA-Modell benötigt werden.

Um zu erklären, wie der ARIMA-Trainingsalgorithmus auf Big Data angewendet werden kann, werden SystemML und insbesondere seine Optimierungsmethoden kurz beschrieben.

Abschließend wird die Implementierung von ARIMA unter Verwendung von Declarative Machine Learning Language diskutiert.

1 Introduction

On the 31st of January 2015 NASA has launched the 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 multiple sensors that each take an average of 2000 samples per second which are within a payload of 4 Bytes each. And there are about 15.000 sensors on board. 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. All in all the data produced amounts to roughly 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 value is associated with the time of its measurement and therefore time stamped. 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.[1]

The goal of time series analysis 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]

For this, several models have been developed. The ARIMA (Autoregressive Integrated Moving Average) model is one of them and wildly spread, because of its ability to take into account a reasonable amount of the complexity that time series datasets might embody. However, this doesn't address the complications originating from the sheer amount of data that is supposed to be used to fit the model.[2]

But that is a problem that has already been addressed in multiple ways and a solution that is two-fold: On one hand distributed computing is needed to use the full capacity of a cluster of servers to divide the computing task among individual machines, while on the other hand there is also the need for distributed file system, because all of the machines need to be able to access the same data that the computations might depend on. There are multiple frameworks out there that provide these capabilities. One of these is Apache Hadoop which offers a distributed file system and another one is Apache Spark which can be used for distributed computing.[3]

However implementing the training algorithms needed for Machine Learning to work on large scale required the data scientist to also know how to use Hadoop and Spark, which

is usually not the case. 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 a R-Like programming language, called Declarative Machine Learning Language (DML), SystemML is able to scale any algorithm based on data and cluster characteristics. This makes the need for a separate Hadoop or Spark engineer obsolete and enables the data scientist to use a familiar programming language while still having an algorithm that works for big data.[4]

Furthermore the offering of SystemML is supposed to be extended to simplify time series analysis for data scientists even more. For this the algorithms needed to train an ARIMA model, as well as forecast using that model, are supposed to be implemented in DML.

2 Theory

2.1 Time Series Analysis

Analyzing time series is about understanding and modeling the different characteristics that time series data can exhibit. The most commonly studied features of time series data are:

- Noise
- General trend
- Cyclic movements
- Seasonality
- Pulses and Steps
- Stationarity

The first one that is mostly referred to as *noise* reflects unexpected variations in the time series. It is often used to describe a type of error in a time series model that is due to a lack of information about explanatory variables that can model these variations or due to presence of random noise. [5]

There can be different models used in which there is no trend nor seasonal component. One of the those models describes the independent and identically distributed noise or iid noise. In this model observations are simply independent and identically distributed variables with zero mean. This means that there are no dependencies between observations. [6]

However there is also another model that does show dependencies. And it is known as *Random Walk Model*. The random walk S_t is obtained by cumulatively summing iid random variables $[Z_t]$ with $S_0 = 0$ and can therefore be defined as:[6]

$$S_t = Z_1 + Z_2 + Z_3 + ... + Z_t \tag{2.1}$$

If furthermore the random variables $[Z_t]$ are Bernoulli distributed with $p = 0.5$ then it is called a simple symmetric random walk. An example for this could be a pedestrian who

starts from a position zero at time zero and continuously tosses a fair coin, stepping one unit to the right each time a head appears and one unit to the left for each tail. Both these model are referenced to as *zero-mean models*.[**6**]

An example of the random walk model as well as an iid model can be seen in figure 2.1

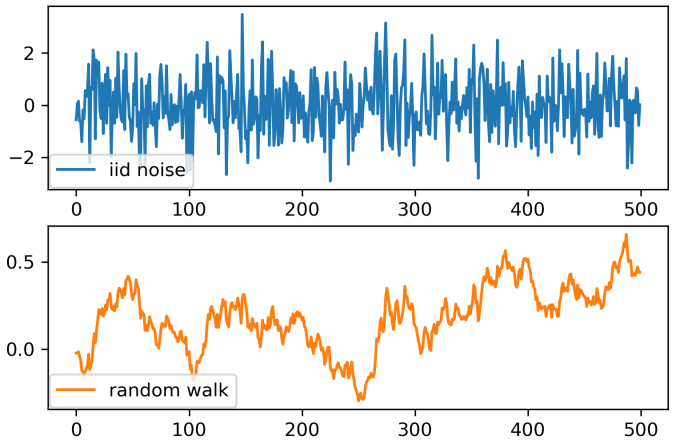

Figure 2.1: Example for iid model (top) and random walk model (bottom)

There are several time series in which a clear trend can be observed in the data. This means it exhibits an upward or downward movement in the long run. In those cases a zero mean model is not sufficient to describe the series. [**5**]

Trend models aim to capture this long run trend in a time series. They can be fitted as linear regressions of the time index. And for this a trend model X_t can be expressed as the sum of the trend component M_t being a slowly changing function and a zero mean model Z_t:[**6**]

$$X_t = M_t + Z_t \tag{2.2}$$

Besides general trend and noise, many time series are also influenced by factors that occur periodically. These repetitive fluctuations are called *cyclic movements*. These can further be divided into nonseasonal cycles and seasonal cycles. Nonseasonal cycles are repetitive, possibly unpredictable patterns in time series values and defined by a periodicity that varies over time. In contrast to that, seasonal cycles have a known constant periodicity. In this case, the time series is said to exhibit seasonality. One example causing seasonality in a series is dependency of the observed system on weather. This in itself is seasonal and when the weather then interferes with the observed system this interference can be seen in periodic fluctuations that occur in a fixed period.[**7**]

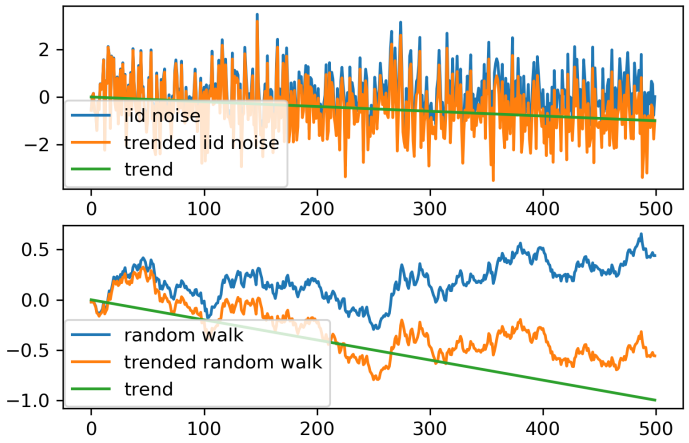

Figure 2.2: Example of trend models using iid and random walk as zero mean models

In order to represent such seasonal effects, allowing noise but assuming no trend, we can use a model that can be expressed as combination of an harmonic regression and the zero mean model [6]

$$X_t = S_t + Z_t \tag{2.3}$$

where S_t is a periodic function of t with period d ($S_{t-d} = S_t$) and is defined as:

$$S_t = a_0 + \sum_{j=1}^{k} (a_j \cos(\lambda_t) + b_j \sin(\lambda_t)) \tag{2.4}$$

here $a_0, a_1, ..., a_k$ and $b_1, ..., b_k$ are unknown parameters and $\lambda_1, ..., \lambda_k$ are fixed frequencies, each being some integer multiple of $2\pi/d$.

To model a time series either a additive or multiplicative model can be used. Both models combine the previously described trend component M_t, seasonal component S_t and the random walk model as the noise or zero-mean component Z_t. The multiplicative decomposition can be expressed as $X_t = M_t \cdot S_t \cdot Z_t$ and is to be used if the variation in S_t appears to be proportional to X_t. Otherwise the additive model $X_t = M_t + S_t + Z_t$ can be used. In the case of fitting a multiplicative model one can instead fit an additive model of the logarithm of the components giving the same result as the multiplicative but as fast as the additive one: [8][9]

$$\begin{aligned} \log(X_t) &= \log(M_t \cdot S_t \cdot Z_t) \\ \log(X_t) &= \log(M_t) + \log(S_t) + \log(Z_t) \end{aligned} \tag{2.5}$$

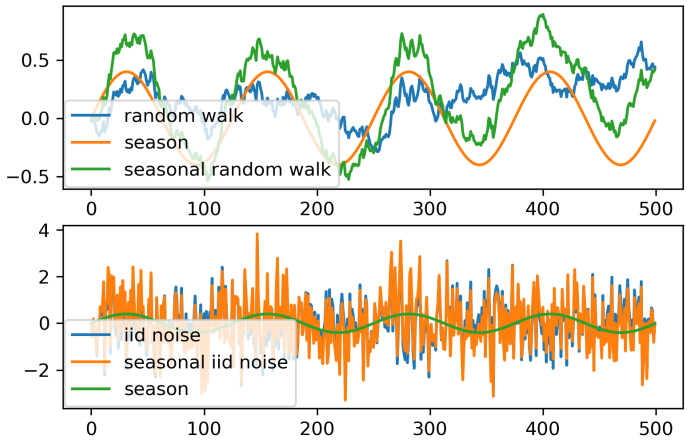

Figure 2.3: Example of seasonal models using iid and random walk as zero mean models

An example of a decomposed and visually represented time series using the additive model can be seen in figure 2.4. However, most real world time series data can usually not be captured completely using only those three components because they also show other more complex characteristics that are harder to model and therefore predict. These more complex characteristics include *pulses* and *steps* as well as different types of *outliers*.[10]

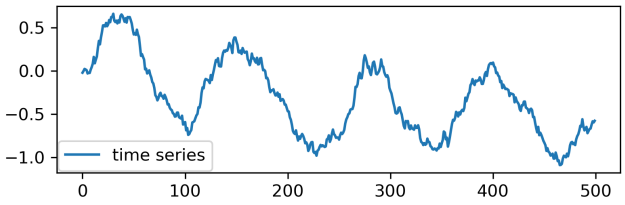

Figure 2.4: Example of time series decomposed into seasonal, trending and zero mean model

Pulses and *steps* are abrupt changes in level that a series might exhibit: A *pulse* is defined as a temporary shift and a *step* is a permanent shift in the series.

When steps or pulses are observed, it is important to find a plausible explanation. Time series models are designed to account for gradual, not sudden, change. As a result, they tend to underestimate pulses and are ruined by steps, which leads to poor model fits and uncertain forecasts. If a disturbance can be explained, it can be modeled using an intervention or event.[10]

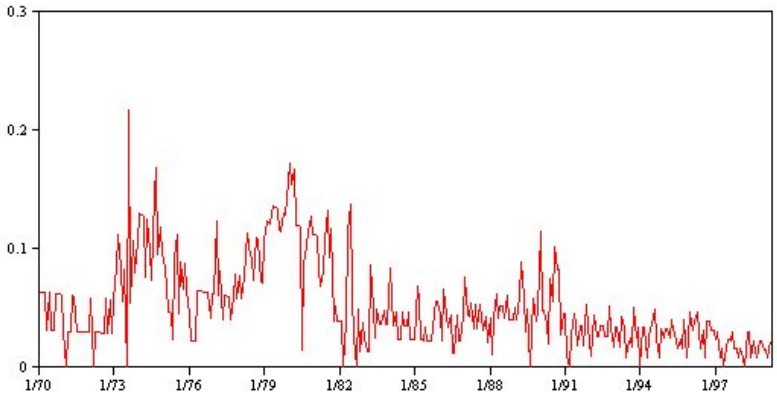

Figure 2.5: Example of a pulse in a time series[10]

A particular importance to many time series forecasting models has a characteristic that is called stationarity. This is because a lot of time series models use the assumption that the series to be forecast is already stationarity or can be made approximately stationary through the means of mathematical transformations. [11] A time series is called stationary if its statistical properties stay constant over time. These restrictions have to apply to those properties that depend on the first- and second-order moments of X_t.[5][6]

Let X_t be a time series with $E(X_t^2) < \infty$. The mean function of X_t is

$$\mu_X(t) = E(X_t) \tag{2.6}$$

The covariance function of X_t is

$$\gamma_X(r,s) = Cov(X_r, X_s)E[(X_r - \mu_X(r))(X_s - \mu_X(s))] \tag{2.7}$$

for all integers r and s. X_t is (weakly) stationary if

i $\mu_X(t)$ is independent of t,

and

ii $\gamma_X(t+h,t)$ is independent of t for each h.

To obtain meaningful statistical properties such as mean, variance, and correlations a timer series has to be stationarized first. Only then such statistics can be used as descriptors for future behavior. For example, if the series is consistently increasing over time, the sample mean and variance will grow with the size of the sample, and they will always underestimate the mean and variance in future periods. Therefore, based on the mean and variance the correlation with other variables would be under- or overestimated, too.[11]

This is why there is a need for mathematical transformations that can make a non-stationary time series approximately stationary.

One of those mathematical transformations is called *de-trending* and can be used when the series has a stable long-run trend and no seasonality. This is achieved by fitting a trend model to the time series and then subtracting it from the original series. The resulting time series can then be further analyzed and modeled and is called *trend-stationary* if the process was able to stationarize the series. An equivalent process is working for time series which do show seasonality. Of course in those cases a seasonal model has to be fitted and then subtracted from the series. There is also the possibility of combining both of these approaches. However, not all time series can be stationarized by these processes. For some series this is insufficient and they might have to be differenced, either from period to period or from season to season. The idea is that if, even after removing a trend and/or seasonal model, the statistical characteristics are still not constant over time, then the statistics of the changes in the series between periods or seasons might be constant. In that case it is said to be *difference-stationary*. Using unit-root tests can help identifying what stationarizing method might be more successful.[11]

For the process of nonseasonal *first order differencing* the *lag-1* operator Δ is defined by:[6]

$$\Delta X_t = X_t - X_{t-1} = (1 - B)X_t \tag{2.8}$$

where B is the backward shift or *backshift* operator

$$BX_t = X_{t-1} \tag{2.9}$$

Powers of operators B and Δ are defined by $B^j X_t = X_{t-j}$ and $\Delta^j X_t = \Delta(\Delta^{j-1} X_t), j >= 1$ with $\Delta^0 X_t = X_t$. Polynomials in B and Δ are manipulated in precisely the same way as polynomial functions of real variables. For example:[6]

$$\Delta^2 X_t = \Delta(\Delta X_t) = (1 - B)(1 - B)X_t = (1 - 2B + B^2)X_t = X_t - 2X_{t-1} + X_{t-2} \tag{2.10}$$

First order differencing is an example for a period to period differencing and is used to remove the trend from a time series. However, as seen in 2.6 the seasonality is still exhibited in the same way and to remove this as well, seasonal differencing has to be applied additionally.

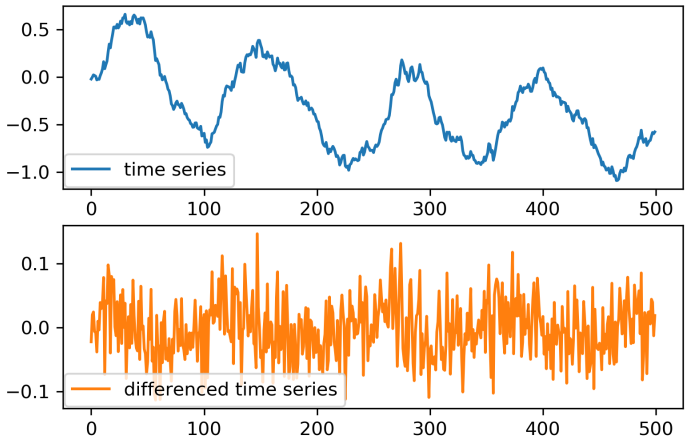

Figure 2.6: Example of a time series with its first order non-seasonal difference

For the process of seasonal *first order differencing* the *lag-1* operator Δ_s with seasonality s is defined by:[6]

$$\Delta_s X_t = X_t - X_{t-s} = (1 - B^s)X_t \tag{2.11}$$

And the *D-th order difference* is defined by:[6]

$$\Delta_s^D X_t = (X_t - X_{t-s})^D = (1 - B^s)^D X_t \tag{2.12}$$

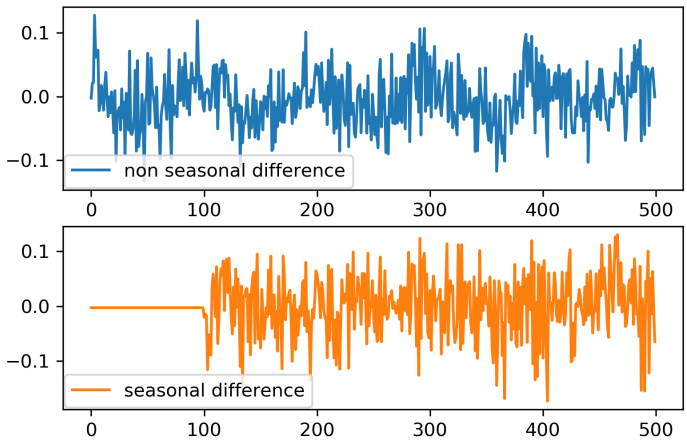

Figure 2.7: Example of a time series with its first order non-seasonal difference

2.2 ARIMA Model

The ARIMA model is used to better understand and forecast time series data. It is based on the Autoregressive Moving Average (ARMA) model but can also handle data that shows evidence of non-stationarity. ARMA is composed out of the Autoregressive (AR) model and the Moving Average (MA) model.

Autoregressive models are used to predict future values of a time series by using linear combinations of previous values of the series and Moving Average models try to forecast future values by combining the prediction error that has been made for previous predictions in the time series.

Given an univariate time series X_t that is stationary and e_t a random variable with an independent and identical distribution representing the error that can occur in any prediction, the AR(p) model is defined by:[6]

$$X_t = e_t + \sum_{i=1}^{p}(\phi_i B^i)X_t = e_t + \phi_1 X_{t-1} + \phi_2 X_{t-2} + ... + \phi_p X_{t-p} \tag{2.13}$$

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: [6]

$$e_t = X_t - \hat{X}_t \tag{2.14}$$

The approximation of X_t using an AR(4) can be written as:

$$\hat{X}_t = \phi_1 X_{t-1} + \phi_2 X_{t-2} + \phi_3 X_{t-3} + \phi_4 X_{t-4} \tag{2.15}$$

The Moving Average model is based on the definition (2.14) of the error term e_t. In contrast to the AR model it combines e_t terms instead of X_t. Hence the definition of MA(q) is:

$$X_t = e_t + \sum_{k=1}^{q}(\theta_k B^k)e_t = e_t + \theta_1 e_{t-1} + \theta_2 e_{t-2} + ... + \theta_p e_{t-q} \tag{2.16}$$

Using e_t makes the calculation of an MA(q) model more complicated, because instead of X_t the e_t term can not be used directly but has to be calculated first. For example the MA(3) model could be expressed as:

$$X_t = e_t + \theta_1 e_{t-1} + \theta_2 e_{t-2} + \theta_3 e_{t-3} \tag{2.17}$$

Replacing e_t with its definition (2.14) and transforming the equation to $\hat{X}_t$ shows why the computation of an MA model is more complicated:

$$\hat{X}_t = \theta_1(X_{t-1} - \hat{X}_{t-1}) + \theta_2(X_{t-2} - \hat{X}_{t-2}) + \theta_3(X_{t-3} - \hat{X}_{t-3}) \tag{2.18}$$

This form shows clearly that $\hat{X}_t$ is dependent on $\hat{X}_{t-1}$, $\hat{X}_{t-2}$, $\hat{X}_{t-3}$ which are also unknown variables that have to be calculated first. Using the same formula for $\hat{X}_{t-1}$ and so on, this leads to more dependencies which lead to more. One eventually ends up with a system of linear equation that, assuming X_t is a series of length 5, looks like this:

$$\begin{aligned}
\hat{X}_5 &= \theta_1(X_4 - \hat{X}_4) + \theta_2(X_3 - \hat{X}_3) + \theta_3(X_2 - \hat{X}_2) \\
\hat{X}_4 &= \theta_1(X_3 - \hat{X}_3) + \theta_2(X_2 - \hat{X}_2) + \theta_3(X_1 - \hat{X}_1) \\
\hat{X}_3 &= \theta_1(X_2 - \hat{X}_2) + \theta_2(X_1 - \hat{X}_1) \\
\hat{X}_2 &= \theta_1(X_1 - \hat{X}_1) \\
\hat{X}_1 &= 0
\end{aligned} \tag{2.19}$$

To be able to calculate this for larger time series, some kind of solving mechanism is needed. To be able to use one of those the system of linear equations needs to be in the form of:

$$\mathbf{A}\vec{\hat{x}} = \vec{b} \tag{2.20}$$

with $\mathbf{A} \in \mathbb{R}^{n \times n}$ being a $n \times n$ 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 values for each equation, $\vec{\hat{x}}$ the unknown variables and $\mathbf{A}$ the coefficients of all the $\vec{\hat{x}}$ for each equation.

To see how $\mathbf{A}$ and $\vec{b}$ have to be constructed, the system of linear equations (2.19) can be transformed to:

$$\begin{array}{rcl}
\hat{X}_1 & = & 0 \\
\theta_1\hat{X}_1 + \hat{X}_2 & = & \theta_1 X_1 \\
\theta_2\hat{X}_1 + \theta_1\hat{X}_2 + \hat{X}_3 & = & \theta_1 X_2 + \theta_2 X_1 \\
\theta_3\hat{X}_1 + \theta_2\hat{X}_2 + \theta_1\hat{X}_3 + \hat{X}_4 & = & \theta_1 X_3 + \theta_2 X_2 + \theta_3 X_1 \\
\theta_3\hat{X}_2 + \theta_2\hat{X}_3 + \theta_1\hat{X}_4 + \hat{X}_5 & = & \theta_1 X_4 + \theta_2 X_3 + \theta_3 X_2
\end{array} \tag{2.21}$$

This leads directly to the form required in (2.20):

$$\begin{pmatrix}
1 & 0 & 0 & 0 & 0 \\
\theta_1 & 1 & 0 & 0 & 0 \\
\theta_2 & \theta_1 & 1 & 0 & 0 \\
\theta_3 & \theta_2 & \theta_1 & 1 & 0 \\
0 & \theta_3 & \theta_2 & \theta_1 & 1
\end{pmatrix} \vec{\hat{x}} = \begin{pmatrix}
0 \\
\theta_1 X_1 \\
\theta_1 X_2 + \theta_2 X_1 \\
\theta_1 X_3 + \theta_2 X_2 + \theta_3 X_1 \\
\theta_1 X_4 + \theta_2 X_3 + \theta_3 X_2
\end{pmatrix} \tag{2.22}$$

Having the system of linear equations in this form, the MA(4) model with given θ_k for $k \in [1, ..., 4]$ can be calculated using a numerical solver for systems of linear equations like the Jacobi (chapter **??**) or Conjugate Gradient solver.

The ARMA(p, q) model can now be composed out of the Autoregressive(p) and Moving Average(q) model defined previously by the equations (2.13) and (2.16) and can be represented as:

$$\begin{array}{rcl}
(1 - \sum_{i=1}^{p} \phi_i B^i) X_t & = & (1 + \sum_{k=1}^{q} \theta_k B^k) e_t \\
\Leftrightarrow X_t - \phi_1 X_{t-1} - \phi_2 X_{t-2} - ... - \phi_p X_{t-p} & = & e_t + \theta_1 e_{t-1} + \theta_2 e_{t-2} + ... + \theta_p e_{t-q}
\end{array} \tag{2.23}$$

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

$$\rightarrow \hat{X}_t = \phi_1 X_{t-1} + \phi_2 X_{t-2} + ... + \phi_p X_{t-p} + \theta_1 e_{t-1} + \theta_2 e_{t-2} + ... + \theta_p e_{t-q} \tag{2.24}$$

Because this equation (2.24) is just a combination of the equations (2.13) and (2.16) it also comes with the same difficulties as previously described and therefore has to be transformed the same way, so it is in the form of equation (2.20).

This is achieved with the same procedures as described for the example of MA(3) eventually leading to an equation in the form of (2.22) with all the AR terms of the equation contained within $\vec{b}$.

However, like already mentioned in the beginning of this chapter: ARIMA is a generalization of ARMA with the additional ability to also work with non-stationary time series. It accomplishes this by differencing the time series d times. Hence, the ARIMA(p, d, q) model also includes the equation for first order differencing (2.8) - modified to equal d-th order differencing - and is therefore defined as:[6]

$$(1 - \sum_{i=1}^{p} \phi_i B^i) \quad (1 - B^d) \quad X_t \quad = \quad (1 + \sum_{k=1}^{q} \theta_k B^k) \quad e_t \tag{2.25}$$

But this model won't be able to fit any seasonal behavior, because it is missing the seasonal parts of all three components AR, MA and d-th order differencing.

The most commonly used ARIMA model for forecasting seasonal time series is the multiplicative Seasonal Autoregressive Integrated Moving Average (SARIMA) model. This model assumes that there is a significant parameter as a result of multiplication between nonseasonal and seasonal parameters. There is also an additive (SARIMA) model that can be used to forecast.[9][12]

The additive models for Seasonal Autoregressive SAR(p, P) are defined as:

$$\begin{aligned} X_t &= e_t + \sum_{i=1}^{p} \phi_i B^i X_t + \sum_{j=1}^{P} \Phi_j B^{i \cdot s} X_t \\ X_t &= e_t + \phi_1 X_{t-1} + ... + \phi_p X_{t-p} + \Phi_1 X_{t-1s} + ... + \Phi_P X_{t-Ps} \end{aligned} \tag{2.26}$$

And analog to this the additive Seasonal Moving Average SMA(q, Q) model is defined by:

$$\begin{aligned} X_t &= e_t + \sum_{k=1}^{q} \theta_k B^k e_t + \sum_{l=1}^{Q} \Theta_l B^{l \cdot s} e_t \\ X_t &= e_t + \theta_1 e_{t-1} + ... + \theta_q e_{t-q} + \Theta_1 e_{t-1s} + ... + \Theta_Q e_{t-Qs} \end{aligned} \tag{2.27}$$

The differencing component is always multiplicative and therefore by also adding the equation (2.12) for the seasonal D-th order difference the additive SARIMA$(p,d,q)(P,D,Q)_s$ is defined as:

$$(1-(\sum_{i=1}^{p}\phi_i B^i+\sum_{j=1}^{P}\Phi_j B^{i\cdot s}))\quad(1-B^d)\quad X_t\quad=\quad(1+(\sum_{k=1}^{q}\theta_k B^k+\sum_{l=1}^{Q}\Theta_l B^{l\cdot s}))\quad e_t \tag{2.28}$$

The multiplicative models are a little bit more complicated for both SAR as well as SMA. With the multiplicative Seasonal Autoregressive SAR(p,P) model being the simpler one defined by:

$$\begin{aligned}
e_t &= (1-\sum_{i=1}^{p}\phi_i B^i)(1-\sum_{j=1}^{P}\Phi_j B^{i\cdot s})X_t \\
e_t &= (1-\phi_1 B-\phi_2 B^2-...-\phi_p B^p)(1-\Phi_1 B^s-\Phi_2 B^{2s}-...-\Phi_p B^{Ps})X_t \\
e_t &= (1-\phi_1 B-\phi_2 B^2-...-\phi_p B^p-\Phi_1 B^s-\Phi_2 B^{2s}-...-\Phi_p B^{Ps} \\
&\quad +\phi_1\Phi_1 B^{1+s}+\phi_2\Phi_1 B^{2+s}+\phi_2\Phi_2 B^{2+2s}+...+\phi_p\Phi_P B^{p+Ps})X_t \\
X_t &= e_t+\sum_{i=1}^{p}\phi_i B^i X_t+\sum_{j=1}^{P}\Phi_j B^{i\cdot s}X_t-\sum_{i=1}^{p}\sum_{j=1}^{P}\phi_i\Phi_j B^{i+js}X_t
\end{aligned} \tag{2.29}$$

And the multiplicative Seasonal Moving Average SMA(q,Q) model defined as:

$$\begin{aligned}
X_t &= (1+\sum_{k=1}^{q}\theta_i B^k)(1+\sum_{l=1}^{Q}\theta_j B^{k\cdot s})e_t \\
X_t &= (1+\theta_1 B-\theta_2 B^2+...+\theta_p B^q+\theta_1 B^s+\theta_2 B^{2s}+...+\theta_p B^{Qs} \\
&\quad +\theta_1\theta_1 B^{1+s}+\theta_2\theta_1 B^{2+s}+\theta_2\theta_2 B^{2+2s}+...+\theta_p\theta_P B^{q+Qs})e_t \\
X_t &= e_t+\sum_{k=1}^{q}\theta_i B^k e_t+\sum_{l=1}^{Q}\theta_j B^{k\cdot s}e_t+\sum_{k=1}^{q}\sum_{l=1}^{Q}\theta_i\theta_j B^{k+js}e_t
\end{aligned} \tag{2.30}$$

This directly leads to the definition of multiplicative SARIMA$(p,d,q)(P,D,Q)_s$ as[9] :

$$(1-\sum_{i=1}^{p}\phi_i B^i)(1-\sum_{j=1}^{P}\Phi_j B^{i\cdot s})(1-B^d)(1-B^s)^D X_t\quad=\quad(1+\sum_{k=1}^{q}\theta_k B^k)(1+\sum_{l=1}^{Q}\theta_j B^{k\cdot s})e_t \tag{2.31}$$

The complication with this model originates from the third sum that is now also combining all the ϕ and Φ in case of AR and all the θ and Θ for MA.

The terms of the third sum increase exponentially if both parameters are increased, which leads to an equivalent increase in compile-time. Additionally this further complicates the transformation of the system of linear equations for Moving Average.

For example, the equation for the approximation $\hat{X}_t$ of multiplicative SARIMA$(2,0,1)(1,0,2)_3$ would look like:

$$\begin{aligned}\hat{X}_t &= \phi_1 X_{t-1} + \phi_2 X_{t-2} + \Phi_1 X_{t-2} + \phi_1\Phi_1 X_{t-3} + \phi_2\Phi_1 X_{t-4} \\ &\quad + \theta_1(X_{t-1} - \hat{X}_{t-1}) + \Theta_1(X_{t-2} - \hat{X}_{t-2}) + \Theta_2(X_{t-4} - \hat{X}_{t-4}) \\ &\quad + \theta_1\Theta_1(X_{t-3} - \hat{X}_{t-3}) + \theta_1\Theta_2(X_{t-5} - \hat{X}_{t-5})\end{aligned} \tag{2.32}$$

Bringing this in the form required in (2.20) assuming that the time series X_t is only of length 8:

.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ \theta_1 & 1 & 0 & 0 & 0 & 0 \\ \Theta_1 & \theta_1 & 1 & 0 & 0 & 0 \\ \theta_1\Theta_1 & \Theta_1 & \theta_1 & 1 & 0 & 0 \\ \Theta_2 & \theta_1\Theta_1 & \Theta_1 & \theta_1 & 1 & 0 \\ \theta_1\Theta_2 & \Theta_2 & \theta_1\Theta_1 & \Theta_1 & \theta_1 & 1 \end{pmatrix} \begin{pmatrix} \hat{X}_1 \\ \hat{X}_2 \\ \hat{X}_3 \\ \hat{X}_4 \\ \hat{X}_5 \\ \hat{X}_6 \end{pmatrix} = \begin{pmatrix} 0 & & & & & \\ \phi_1\theta_1 X_1 & & & & & \\ \phi_1\theta_1 X_2 & +\Phi_1\phi_2\theta_2 X_1 & & & & \\ \phi_1\theta_1 X_3 & +\Phi_1\phi_2\theta_2 X_2 & +\Phi_1\Theta_1 X_1 & & & \\ \phi_1\theta_1 X_4 & +\Phi_1\phi_2\theta_2 X_3 & +\Phi_1\Theta_1 X_2 + & \Theta_2 X_1 & & \\ \phi_1\theta_1 X_5 & +\Phi_1\phi_2\theta_2 X_4 & +\Phi_1\Theta_1 X_3 + & \Theta_2 X_2 & +\theta_1\Theta_2 X_1 & \end{pmatrix} \tag{2.33}$$

For the estimation of the SARIMA coefficients ϕ, Φ, θ and Θ an objective function is needed to be optimized over. And there are different estimators that can be used for this purpose[13][14]:

- Maximum Likelihood (ML) estimation
- Yule-Walker estimation
- Least Squares or Conditional Sum of Squares (CSS) method

With the Yule-Walker and the Maximum Likelihood estimator described in detail in Peter J. Brookwell's 2002 "Introduction to Time Series and Forecasting, Second Edition" and also being the more complicated two.

The Conditional Sum of Squares method in contrast to that can, for X_t with size T, be simple put as the sum of squared residuals[14]:

$$ARIMA_{CSS}(\phi_{1..p}, \Phi_{1..P}, \theta_{1..q}, \Theta_{1..Q}) = \tfrac{1}{2}\sum_{t=1}^{T}(e_t)^2 = \frac{1}{2}\sum_{t=1}^{T}(X_t - \hat{X}_t)^2 \tag{2.34}$$

2.3 Optimization Algorithms

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

$$\begin{gathered}\max_{\phi_{1..p},\Phi_{1..P},\theta_{1..q},\Theta_{1..Q}\in R} g(\phi_{1..p}, \Phi_{1..P}, \theta_{1..q}, \Theta_{1..Q}) \\ or \\ \min_{\phi_{1..p},\Phi_{1..P},\theta_{1..q},\Theta_{1..Q}\in R} g(\phi_{1..p}, \Phi_{1..P}, \theta_{1..q}, \Theta_{1..Q})\end{gathered} \tag{2.35}$$

with $g(\phi_{1..p}, \Phi_{1..P}, \theta_{1..q}, \Theta_{1..Q})$ being the Maximum Likelihood, Yule-Walker, or Conditional Sum of Squares estimator.

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:

- Nelder-Mead method,
- Broyden-Fletcher–Goldfarb-Shanno (BFGS) method,
- Limited-memory Broyden-Fletcher–Goldfarb-Shanno (L-BFGS) method, and
- Brent method

But only the BFGS algorithm has been fully implemented for the DML script. It belongs to the class of *Quasi-Newton* methods and can therefore be used to find roots or local maxima and minima of real-valued functions. *Quasi-Newton* methods do this faster then the "full" *Newton* method by approximating the Hessian or Jacobian instead of calculating it exactly.

If the function $f : R \to R$ its *derivative*, $f'(x)$ and an initial guess $\hat{x}_0$ is given then the *Newton* method's iterative approximation $\hat{x}_{n+1}$ of the root of $f(x)$ is given by[15]:

$$\hat{x}_{n+1} = \hat{x}_n - f(\hat{x}_n)/f'(\hat{x}_n) \tag{2.36}$$

This of course is only for univariate functions. The definition of the iterative approximation $\hat{x}_{n+1}$ of the root for multivariate function $f(x) : R^k -> R$ is:

$$\hat{x}_{n+1} = \hat{x}_n - \frac{f(\hat{x}_n)}{J_f(\hat{x}_n)} = \hat{x}_n - f(\hat{x}_n) \cdot J_f^{-1}(\hat{x}_n) \tag{2.37}$$

With $J_f(\hat{x}_n)$ being the Jacobian matrix, the matrix of all *first-order* partial derivatives.

Using the *Newton* method to optimize the function f is equivalent to finding the root of the *derivative* f' which means that the approximation $\vec{\hat{x}}_{n+1}$ for the extrema of f is given by[15]:

$$\hat{x}_{n+1} = \hat{x}_n - \frac{\Delta f(\hat{x}_n)}{H_f(\hat{x}_n)} = \hat{x}_n - \Delta f(\hat{x}_n) \cdot H_f^{-1}(\hat{x}_n) \tag{2.38}$$

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

The drawback of using the *Newton* method is that it requires inverting the Hessian, which means a computation of $O(n^3)$ using standard techniques. Additionally the exact calculation of the Hessian takes $O(n^2)$ function evaluations (partial derivatives). Consequently using the Newton method is getting expensive for large n.[15]

Quasi-Newton methods try to overcome these limitations by approximating the Hessian instead of calculating it directly. This can be achieved by leveraging the *secant* method.[16]

The *secant* approximation of the second derivative of the univariate function $f(x)$ is:

$$\begin{aligned} f''(x_k) &\approx \tfrac{f'(x_k) - f'(x_{k-1})}{x_k - x_{k-1}} \\ \Leftrightarrow \quad f''(x_k) \cdot (x_k - x_{k-1}) &\approx f'(x_k) - f'(x_{k-1}) \end{aligned} \tag{2.39}$$

The generalization of (2.39) for a multivariate function is:

$$\Delta^2 f(x_k) \cdot (x_k - x_{k-1}) \approx \Delta f(x_k) - \Delta f(x_{k-1}) \tag{2.40}$$

Quasi-Newton methods try to find the Hessian $H_f(x_k) \approx \Delta^2 f(x_k)$ to make (2.40) an equality. For this there is an initial guess needed for $H_0 = I$ which is then incrementally improved by updating H_{k+1}.[16]

Using this approach the Hessian $H_f(x_k)$ still needs to be inverted which is also a time consuming computation, that can be avoided by instead approximating the inverse Hessian $H_f^{-1}(x_k)$ directly.

In that case we define B_k as the inverse Hessian $H_f^{-1}(x_k)$ and the search direction p_k that has to be computed are defined as:

$$p_k = -B_k \cdot \Delta f(x_k) \tag{2.41}$$

To calculate the updated approximation of B_k an acceptable step size α in the direction p_k has to be found by using a line search:

$$\min_{\alpha} \quad h(\alpha) = f(x_k + \alpha p_k) \tag{2.42}$$

An exact line search algorithm would determine a value for α that exactly minimizes $h(\alpha)$. However this is not always necessary or even desirable because of the additional computing costs it would require. Instead using, for example the backtracking line search, the step size α can be approximated reasonably well, which is sufficient for most cases.[16]

The backtracking line search starts the same way an exact one would, by guessing $\alpha_0 > 0$ and then shrinking it in every iteration by multiplying it with a constant $r \in]0, 1[$:

$$\alpha_{k+1} = r \cdot \alpha_k \tag{2.43}$$

This is repeated as long as the Armijo-Goldstein condition is fulfilled, which tests whether the new smaller step size achieves a adequately corresponding decrease in the objective function $h(\alpha)$ defined in (2.42). The *Armijo-Goldstein* condition is fulfilled if:

$$f(x + \alpha p) <= f(x) + \alpha c m \tag{2.44}$$

With $m = p^T \Delta f(x)$ and $c \in]0, 1[$ being a pre defined control parameter.

Given p_k and α_k the update formula of Broyden-Fletcher–Goldfarb-Shanno algorithm for B is defined by:

$$B_{k+1} = B_k + \frac{(s_k^T y_k + y_k^T B_k y_k)(s_k s_k^T)}{(s_k^T y_k)^2} - \frac{B_k(s_k^T y_k + s_k y_k^T)}{s_k^T y_k} \tag{2.45}$$

With $y_k = \Delta f(x_{k+1}) - \Delta f(x_k)$ and $s_k = x_{k+1} - x_k = a_k \cdot p_k$.

The only thing left to be able to optimize using BFGS is the gradient $\Delta f(x)$. This can either be done by approximating using finite differencing or by calculating the exact partial differentials of $f(x)$.

To get more accurate results calculating the partial differentials is preferred. In case of minimizing the objective function $ARIMA_{CSS}$ defined in (2.34) and $\hat{X}_t$

$$\begin{aligned} \hat{X}_t &= \sum_{i=1}^{p} \phi_i B^i X_t + \sum_{j=1}^{P} \Phi_j B^{i \cdot s} X_t - \sum_{i=1}^{p} \sum_{j=1}^{P} \phi_i \Phi_j B^{i+js} X_t \\ &+ \sum_{k=1}^{q} \theta_i B^k e_t + \sum_{l=1}^{Q} \theta_j B^{k \cdot s} e_t + \sum_{k=1}^{q} \sum_{l=1}^{Q} \theta_i \theta_j B^{k+js} e_t \end{aligned} \tag{2.46}$$

the partial differentials are as follows:

$$\begin{aligned} \frac{\delta}{\delta \phi_n} ARIMA_{CSS}(\phi_{1..p}, \Phi_{1..P}, \theta_{1..q}, \Theta_{1..Q}) &= \frac{1}{2} \sum_{t=1}^{T} 2 \cdot (X_t - \hat{X}_t) \cdot (\frac{\delta}{\delta \phi_n}(X_t - \hat{X}_t)) \\ &= \sum_{t=1}^{T} e_t \cdot (-B^n X_t + \sum_{j=1}^{P} \Phi_j B^{n+js} X_t) \end{aligned} \tag{2.47}$$

$$\frac{\delta}{\delta \theta_n} ARIMA_{CSS}(\phi_{1..p}, \Phi_{1..P}, \theta_{1..q}, \Theta_{1..Q}) = \sum_{t=1}^{T} e_t \cdot (-B^n e_t - \sum_{l=1}^{Q} \Theta_j B^{n+ls} e_t) \tag{2.48}$$

$$\frac{\delta}{\delta \Phi_n} ARIMA_{CSS}(\phi_{1..p}, \Phi_{1..P}, \theta_{1..q}, \Theta_{1..Q}) = \sum_{t=1}^{T} e_t \cdot (-B^{n \cdot s} X_t + \sum_{i=1}^{p} \phi_i B^{i+ns} X_t) \tag{2.49}$$

$$\frac{\delta}{\delta \Theta_n} ARIMA_{CSS}(\phi_{1..p}, \Phi_{1..P}, \theta_{1..q}, \Theta_{1..Q}) = \sum_{t=1}^{T} e_t \cdot (-B^{n \cdot s} e_t - \sum_{k=1}^{q} \theta_j B^{k+ns} e_t) \tag{2.50}$$

2.4 Linear Systems Solvers

The hardest part of calculating an approximation $\hat{X}_t$ for the ARIMA model is finding a solution for the system of linear equation depicted in the equations (2.19) and (2.33). And again the solution to this problem can either be calculated exactly or only approximated. One example for an exact solver would be the *Gaussian Elimination* method. However, the complexity of this algorithm is $O(n^3)$ which means that this is not suitable for large systems of equations. And especially for ARIMA this is almost always the case as the system of linear equations will be equal to the size of the time series used to train the model. So instead an iterative approach to numerically approximate a solution is used. The *Jacobi* method is such a solver and is commonly used, because of its simplicity and robustness. Additionally each iteration is quite fast.

Given a system of linear equations in the form of the equation (2.20) that can also be expressed as:[17]

$$\sum_{j=1}^{n} a_{ij} \cdot x_j = b_i \tag{2.51}$$

Using the *fixed-point iteration* method this equation (2.51) can be transformed to be[17]:

$$\begin{aligned}
& \sum_{j=1}^{n} a_{ij} \cdot x_j && = && b_i \\
\Leftrightarrow \quad & a_{ii} \cdot x_i \sum_{\substack{j=1 \\ j \neq i}} a_{ij} \cdot x_j && = && b_i \\
\Leftrightarrow \quad & a_{ii} \cdot x_i && = && b_i \sum_{\substack{j=1 \\ j \neq i}} a_{ij} \cdot x_j \\
\Leftrightarrow \quad & x_i && = && a_{ii} - (b_i \sum_{\substack{j=1 \\ j \neq i}} a_{ij} \cdot x_j)
\end{aligned} \tag{2.52}$$

Leading directly to the update function for the *Jacobi* method[17]:

$$x_i^{(n+1)} = a_{ii} - (b_i \sum_{\substack{j=1 \\ j \neq i}} a_{ij} \cdot x_j^{(n)}) \tag{2.53}$$

2.5 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. The goal of DML is to automatically scale any algorithm by translating all the instructions within the 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. [4]

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 SystemML is invoked. Only at that point, all free variables - or rather parameters the script can be run with - are known and therefore the sizes and characteristics of the matrices (data) needed can be calculated. Using this information combined with the known characteristics of the cluster the script can further be optimized by, for example, calculating the number of nodes needed to run the script.[4]

SystemML does this by translating a DML script through a series of transformations and rewrites into an executable runtime program that can run in parallel on, for example, Spark. 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 abstract syntax tree that describes the original program. This representation is obtained by using a parser generator that creates a parser based on a grammar describing either the R-like DML or Python-Like PyDML language. This parser is used to parse the script 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 that, for example, checks for mandatory parameters of built-in functions. [18]

At the end of the so called language layers, basic statement blocks have been identified. These basic blocks are chunk of straight line code in between branches and for each of those blocks directed acyclic graph (DAGs) of high-level operations (HOPs) are created. These graphs are a data flow representations 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. 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 this specific basic block can be calculated on a single node or needs to be divided 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.[18]

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 runtime 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 on one cluster node. Each low-level operation is runtime-backed-specific and SystemML offers the use of three different run-times. Therefore DML and PyDML scripts can be run in different modes. Either in Spark, Hadoop or Standalone mode. 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 a well established development environment for ML.[18]

3 Methodology

To implement the algorithms for the ARIMA model for SystemML the R-Like DML language has been chosen.

To be able to use the model for forecasting, there are two essential algorithms needed: One for forecasting and one for training the model. To test whether the algorithms for training and forecasting are actually working as they are intended to, there needs to be a method to verifying their correctness.

For this the different outputs produces by the algorithms are compared to the outputs of the implementations of the built-in ARIMA model that R provides. This is chosen to be used as a means of measurement based on the assumption that R's implementation is correct, which is a reasonably good assumption, because of R's good establishment within the Machine Learning and Data Science community. Therefore the algorithms provided within the base framework of R are used broadly and hence well tested and verified by the community of data scientists.

Finally the performance of the algorithms is also of interest. In particular when scaling the algorithm to be used for larger data sets or higher order models. To identify bottlenecks of the algorithms, the time and count of the calculation heavy instructions should be measured too.

3.1 Declarative Machine Learning Language

In SystemML, DML scripts can be run using *spark-submit* in the following command:

```
1  spark-submit SystemML.jar -f hello.dml
```

Listing 3.1: Command running the "hello.dml" script

In general, DML is designed to be as close to the syntax and behavior of R as possible.

Variables in DML can have one of three different data types: frames, matrix, or scalar. Each of those data types can have one out of four value types: double, integer, string, or boolean. The value types that the data types support are restricted in the following way: Frames and matrices are two dimensional with matrices only supporting double values and frames being able to hold any one of the value types in one structure. However, with

all the columns restricted to a single value type. Scalars can be of any value type without further restrictions.[19]

Operators for matrices and scalars in DML follow the same associativity and precedence order as in R. For binary operators of two matrices the dimensions need to match the operator's semantics. When applying a binary operator on a matrix and scalar the operation is performed cell-wise on the matrix and the scalar. The syntax for matrix multiplication is $A\%*\%B$ with A and B being variables holding matrices.[19]

A matrix can be initialized by either using the matrix constructor `matrix()` providing the size and initialization value, by generating a random matrix with `rand()` or by reading it from a file with `read()`.

The built-in indexing functionalities of matrices allows for accessing single cells, rows, columns or sub matrices. The syntax for this is:

```
[Matrix name][lower-row : upper-row],[lower-column : upper-column]
```

Listing 3.2: Matrix range indexing syntax[19]

Providing only one value for the rows or columns is equal to setting both lower and upper to the same value, therefore specifying only or row or column. Leaving both lower and upper out completely will set the default values to select all rows or columns.

```
X[1:2,3:4] # access the sub matrix all the cells form row 1 and 2 that
    are also in column 3 and 4
X[1,4] # access cell in row 1, column 4 of matrix X
X[i,j] # access cell in row i, column j of X.
X[1,]  # access the 1st row of X
X[,2]  # access the 2nd column of X
X[,]   # access all rows and columns of X
```

Listing 3.3: Example for matrix indexing[19]

Additionally to the primitive operators, there are also some predefined functions for manipulating and aggregating matrices:

```
# Constructing a 3 x 4 matrix filled with ones:
A = matrix(1, rows = 3, cols = 4)

# Creating diagonal matrix -> B is 3x3 Identity  Matrix:
B = diag(A[,1:3])

# Concatinating B to A column-wise -> Dimension of C: 3x7
C = cbind(A, B)

# Transposing A -> Dimension of D: 4 x 3:
```

```
D = t(A)

# Casting a matrix cell as scalar
first_cell = as.scalar(A[1,1])

print(nrow(A)) # prints the number of rows of A: 3
print(ncol(A)) # prints the number of columns of A: 4
print(sum(A))  # prints the sum of all cells of A: 12
```

Listing 3.4: Example for matrix manipulation and aggregation[19]

Of course, DML also provides User-Defined Function (UDF). Those functions can store a list of statements containing any statement that is usually allowed. The only restriction is the definition of another UDF. The syntax is as follows:

```
functionName = function([ <DataType>? <ValueType> <var>, ]* )
    return ([ <DataType>? <ValueType> <var>,]*) {
    # function body definition in DML
    statement1
    statement2
    ...
}
```

Listing 3.5: User-Defined Function syntax[19]

3.2 Verifying Correctness

To verify the correctness of our training and prediction algorithms, the script that is used to invoke R's ARIMA needs to be configured to run in the same way that the DML scripts do. This excludes the parameters that are to be set freely when using ARIMA in its different forms like non seasonal and seasonal orders p, d, q, P, D, Q and s as well as the input time series. These parameters only need to be the same for each individual test. Other features however, need to be constantly set to the same values. Given the input variables *data*, *nonseasonalparam*, *seasonalparam* and *seasonality* [19]:

```
arima_model = arima(data, order=nonseasonalparam,  seasonal=list(order=
    seasonalparam, period=seasonality), include.mean=FALSE, transform.
    pars=FALSE, method=c("CSS"), optim.method="BFGS")
```

Listing 3.6: ARIMA model trianing invocation in R

The variable *arima_model* then contains the coefficients that have been found to fit the model to the time series as well as residuals - errors in the prediction when using these coefficients.

Comparing the residuals of R's ARIMA to the residuals calculated by the DML script can be used to verify the correctness of the prediction script. And for the training script the coefficients have to be compared.

A test is defined to be successful if all the values produced by R's implementation match the ones from the DML script up to the 4th digit.

To test the correctness of the calculation of the gradient of ARIMA which is needed when using BFGS, it can be compared to the results of the finite differencing method.

3.3 Benchmarking

To test the performance of any DML script, SystemML already offers two features for benchmarking. First of all, when running any script using *spark-submit* it always measures and prints out the total execution time at the end of every run:

```
1 SystemML Statistics:
2 Total execution time:           0.119 sec.
3 Number of executed Spark inst:  1.
```

Listing 3.7: Example of SystemML's default statistics output

Furthermore, when adding the additional `-stats` parameter to the invocation command, a more detailed version of this statistics is printed out, even showing the execution times on instruction level.

For measuring the performance of the script it should be run with time series of different sizes. These sizes should range from 1000 rows to 10 million rows. Each series should be tested for different values of p. At least going from 1 to 10 and if possible also testing with p equal to 100.

4 Implementation

The implementation of ARIMA needs two separate scripts: One for predictions/forecasting and one for training a suitable model. Naturally the training script is going to use the prediction script in some way to test the current model and improve it.

4.1 Preliminaries

Both scripts need to do three things: First, loading the time series and parameters p, d, q, P, D, Q, and s:

- p being the non seasonal order of AR
- d being the non seasonal differencing order
- q being the non seasonal order of MA
- P being the seasonal order of SAR
- D being the seasonal differencing order
- Q being the seasonal order of SMA
- s being the length of one season, also referenced to as seasonality

Then differencing the time series according to d and D. And finally they both have to construct the auxiliary matrix Z. This matrix is needed to do different calculations within both scripts. The idea is to provide all the X_{t-h} for any $h \in [1, max(p, q, Ps, Qs, p + Ps, q + Qs]$ for each $\hat{X}_t$ to be computed. This means that there is p columns for AR(p), q columns for MA(q), P columns for SAR(P), Q columns for SMA(Q) and when combining non seasonal and seasonal models there is also additional $p \cdot P$ and $q \cdot Q$ columns. Each one of those columns corresponds to one of the X_{t-h} terms found in equation for (2.31). So essentially Z is constructed in a way that the n-th row of Z contains all the time series values needed to predict X_n.

The following function is designed to build such a matrix:

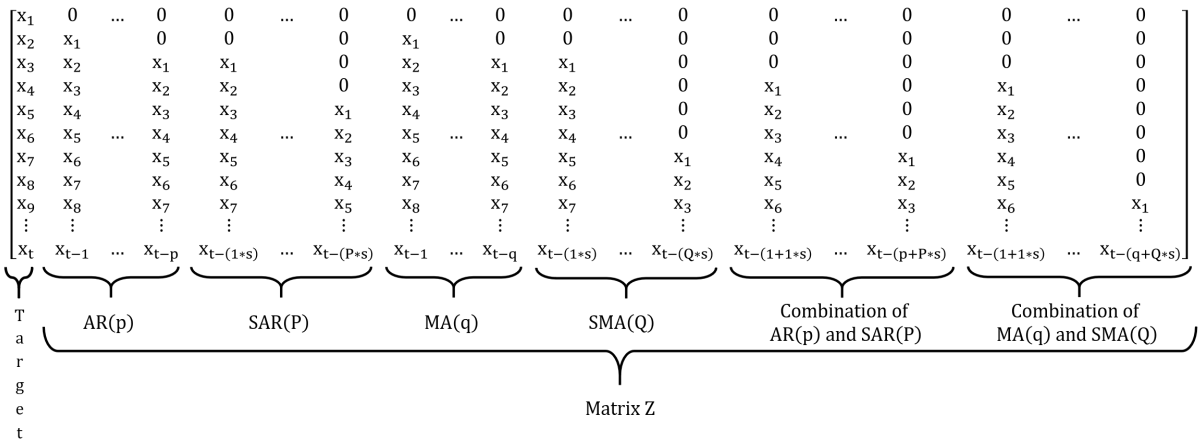

Figure 4.1: Schema of auxiliary matrix Z

```
constructPredictorMatrix = function (Matrix[Double] X, Integer p,
    Integer P, Integer q, Integer Q, Integer s)return( Matrix[Double] Z)
    {
  Z = matrix(0, nrow(X), p+P+Q+q+p*P+q*Q)

  # fills Z with values used for non seasonal AR prediction
  for (i in seq(1, p, 1)){
      Z = addShiftedMatrix(Z, X, i, i)
  }

  #prediction values for seasonal AR
  for(i in seq(1, P, 1)){
      Z = addShiftedMatrix(Z, X, (i * s), p + i)
  }

  #prediction values for combined models of non-seasonal and seasonal AR
  Z = addValuesForCombinedModel (Z, X, p, P, s, p + P)

  #prediction values for non seasonal MA
  for(i in seq(1, q, 1)){
      Z = addShiftedMatrix(Z, X, i, p + P + p*P + i)
  }

  #prediction values for seasonal MA
  for(i in seq(1,Q, 1)){
      Z = addShiftedMatrix(Z, X, (i * s), p + P + p*P + q + i)
  }

  #prediction values for combined models of non-seasonal and seasonal MA
  Z = addValuesForCombinedModel (Z, X, q, Q, s, p + P + p*P + q + Q )
}
```

Listing 4.1: Function for constructing auxiliary matrix Z

The provided function constructs the Z matrix by adding each column one by one. It starts with all the columns for non seasonal then for seasonal AR and proceeds with MA afterwards. Each column that is added is a sub matrix of the time series X. The sub matrix always contains the first elements of X and is only used to cut off the end of the matrix that is not needed. This shorter version of X is then inserted into the new column for Z with an offset of size difference (rows) between X and Z, so the value of the last row of X is in the last row of Z. All cells that are not filled in this way, are set to be zero. The `addShiftedMatrix()` function is therefore defined as:

```
addShiftedMatrix = function (Matrix[Double] targetMatrix, Matrix[Double]
    sourceMatrix, Integer rowOffset, Integer nthColumn) return (Matrix[
    Double] targetMatrix){
  targetMatrix[rowOffset+1:nrow(targetMatrix), nthColumn] = sourceMatrix
      [1:nrow(targetMatrix)-rowOffset, 1]
}
```

Listing 4.2: AddShiftMatrix function

In the case that non seasonal and seasonal models are combined, the ARIMA model is also going to contain terms that are in the form of $\sum_{i=1}^{p}\sum_{j=1}^{P} B^{i+js}X_t$ as shown in the equations for multiplicative SAR (2.29) and SMA (2.30). These terms also need to be added to Z, which is done by using the following function:

```
addValuesForCombinedModel = function (Matrix[Double] targetMatrix,
    Matrix[Double] sourceMatrix, Integer nonSeasonalParam, Integer
    seasonalParam, Integer seasonality, Integer columnOffset)return (
    Matrix[Double] targetMatrix){
  counter = 1
    for (k in seq(1, nonSeasonalParam, 1)){
      for(j in seq(1, seasonalParam, 1)){
        targetMatrix = addShiftedMatrix(targetMatrix, sourceMatrix, k +
            (j*seasonality), columnOffset + counter)
        counter+=1
      }
    }
    #for combination of non seasonal and seasonal model the sign has to
        be inverted:
    if (nonSeasonalParam>0&seasonalParam>0) {
      targetMatrix[,columnOffset+1:columnOffset+nonSeasonalParam*
          seasonalParam] = -targetMatrix[,columnOffset+1:columnOffset+
          nonSeasonalParam*seasonalParam]
    }
```

```
}
```

Listing 4.3: addValuesForCombinedModel function

The calculation of the non seasonal and seasonal difference is done by subtracting the matrix X, representing the time series, with its shifted form. The shift is either by one for non seasonal difference or by s for the seasonal one. And the processes are repeated d and D times:

```
# d-th order differencing:
for(i in seq(1,d,1)){
  X[2:nrow(X),] = X[2:nrow(X),] - X[1:nrow(X)-1,]
}

# D-th order differencing:
for(i in seq(1,D,1)){
  X[s+1:nrow(X),] = X[s+1:nrow(X),] - X[1:nrow(X)-s,]
}
```

Listing 4.4: Differencing

4.2 Prediction Script

The task of the prediction script is simply to calculate the t-dimensional vector $\hat{X}$ containing all the predictions $\hat{X}_t$ for each $t \in [1, T]$ with T being the number of rows of Z. When using the prediction script for training purposes this is also equal to the size of the time series X used for training.

In the simplest case of forecasting AR(p) or SAR(P) the code for the `predict()` function is quite straight forward:

```
predict = function(Matrix[Double] weights, Matrix[Double] Z) return (
    Matrix[Double] approximated_solution){
  approximated_solution = Z%*%weights
}
```

Listing 4.5: Prediction of AR(p) or SAR(P)

For predicting non seasonal and seasonal SAR at once the combination of the purely non seasonal and purely seasonal part have to be taken into consideration. This is done by constructing another auxiliary matrix ω (referred to in the scripts as 'combined_weights') that holds all weights for p and P as well as all the combinations of those weights:

```
combined_weights = weights
if (p>0 & P>0){
  combined_weights = rbind(combined_weights, matrix(weights[1:p,] %*% t(
      weights[p+1:p+P,]), rows=p*P, cols=1))
}
approximated_solution = Z%*%combined_weights
```

Listing 4.6: Prediction of SAR(p)(P)

Like already discussed in chapter 2.2 and described in more detail with the equations (2.16) through (2.21), calculating $\hat{X}$ for MA(p) requires to construct a third matrix R to be used to describe the system of linear equations that needs to be solved with:

$$A = I + R$$

```
b = Z%*%weights
R = matrix(0, nrow(Z), nrow(Z))
for(i in seq(1, q, 1)){
    d_ns = matrix(as.scalar(weights[P+p+i,1]), nrow(R)-i, 1)
    R[1+i:nrow(R),1:ncol(R)-i] = R[1+i:nrow(R),1:ncol(R)-i] + diag(d_ns)
}
approximated_solution = numerical_solvers::jacobi (A=A, b=b)
```

Listing 4.7: Prediction of MA(q)

The equivalent for-loop can be implemented for SMA as well. With the only difference being the use of the term i instead of $i \cdot s$.

To solve the system of linear equations, the *Jacobi* solver had to be implemented as described in chapter **??**:

```
jacobi = function (Matrix[Double] R, Matrix[Double] b, Double tolerance
    =0.01, Integer max_iterations=1000)  return (Matrix[Double] x){
  x = matrix(0, nrow(R), 1)
    k = 0
    diff = tolerance+1
    while(k < max_iterations & diff > tolerance){
        x_k = b - R%*%x
        diff = sum((x_k-x)^2)
        x = x_k
        k += 1
    }
}
```

Listing 4.8: Implementation of *Jacobi* solver

4.3 Training Script

The training script provides the coefficients describing an ARIMA model by solving one of the optimization problems described at the end of chapter 2.2. The first one to be implemented, because of its simplicity in comparison to the other, is the Conditional Sum of Squares defined in equation (2.34) as the sum of the squared residuals.

```
arima_residuals = function(...) return (Matrix[Double] errs, Matrix[
    Double] combined_weights){
  [approximated_solution, combined_weights] = arima_prediction::predict(
      weights, Z, p, P, q, Q, s, solver)
  errs = X - approximated_solution
}
arima_css = function(...) return (Double obj, Matrix[Double] errs){
  [errs, combined_weights] = arima_residuals(weights, X, Z, p, P, q, Q,
      s, solver)
  obj = 0.5 * sum(errs^2)
}
```

Listing 4.9: Functions for calculating residual and Conditional Sum of Squares

When using the BFGS method to minimize the Conditional Sum of Squares then the gradient of that function is also needed. The partial differentials have been derived at the end of chapter **??** and can be described with the equations (2.47), (2.48), (2.49) and (2.50).

For calculating the gradient of an AR(p) or SAR$(P)_s$ model, which again is the simplest case, the transposed Z matrix can just be multiplied with the residuals. This is equal to $\sum_{t=1}^{T} e_t \cdot (-X_{t-n})$ for all partial differentials $\frac{\delta}{\delta\phi_n}$ and $\frac{\delta}{\delta\Phi_n}$:

```
gradArimaCSS = function(Matrix[Double] weights, Matrix[Double] X, Matrix
    [Double] Z, Integer p, Integer P, Integer q, Integer Q, Integer s,
    String solver) return(Matrix[double] grad){
  [residuals, combined_weights] = arima_residuals(weights, X, Z, p, P, q
      , Q, s, solver)

  res_ar = t(-Z) %*% residuals
  grad = matrix(0, rows=p+P+q+Q, cols=1)
  if (p > 0) grad[1:p,] = res_ar[1:p,]
  if (P > 0) grad[p+1:p+P,] = res_ar[p+1:p+P,]
}
```

Listing 4.10: Gradient of Conditional Sum of Squares for AR(p) or SAR$(P)_s$

When the models are combined to SAR$(p)(P)_s$ and both parameters are greater than zero then there is another term to be added to the already calculated gradients:

$$\frac{\delta}{\delta\phi_n} ARIMA_{CSS}(\phi_{1..p}, \Phi_{1..P}, \theta_{1..q}, \Theta_{1..Q}) = \sum_{t=1}^{T} e_t \cdot (-B^n X_t) + \sum_{t=1}^{T} e_t \cdot (\sum_{j=1}^{P} \Phi_j B^{n+js} X_t) \tag{4.1}$$

The first of the two terms that are added here can already be calculated with the function shown in listing 4.10. So it only needs to be altered to calculate and add the second term to each partial differential:

```
if (p>0 & P>0){
    for(i in seq(1, p, 1)){
    permut = matrix(0, rows=p, cols=P)
    permut[i,] = t(combined_weights[p+1:p+P,])
    grad[i,1] = grad[i,1] + sum(res_ar[p+P+1:p+P+p*P,] * matrix(permut,
        rows=p*P, cols=1))
}

for(i in seq(1, P, 1)){
    permut = matrix(0, rows=p, cols=P)
    permut[,i] = combined_weights[1:p,]
    grad[p+i,1] = grad[p+i,1] + sum(res_ar[p+P+1:p+P+p*P,] * matrix(
        permut, rows=p*P, cols=1))
    }
}
```

Listing 4.11: Gradient of Conditional Sum of Squares for SAR$(p)(P)_s$

With both the `ARIMA_CSS()` and the `gradArimaCSS()` functions in place the [!BFGS] optimizer can be implemented. To start the iterative method a first guess of the coefficients (weights) has to be made. And afterwards the Conditional Sum of Squares and its gradient can be calculated. Before the first iteration the approximation of the inverse Hessian is initialized as the identity. The iterative process is stopped if one of the following criteria has been met:

- The difference of `ARIMA_CSS()` of the previous and current iteration is smaller than a predefined tolerance
- The L2 norm of the gradient is smaller than the tolerance
- The number of iteration has reach a predefined maximum

The code for initializing the algorithm should therefore be:

```
bfgs_optimizer = function (Matrix[Double] X, Matrix[Double] Z, Integer
    max_iterations, Integer p, Integer P, Integer q, Integer Q, Integer
    s, String solver)return(Matrix[Double] weights){
```

```
  nParam = p+P+q+Q

    weights = matrix(0.5, nParam, 1)
    fx = arima_css(weights, X, Z, p, P, q, Q, s, solver)
    gx = gradArimaCSS(weights, X, Z, p, P, q, Q, s, solver)
    norm_gx = L2norm(gx)

    #The approximated inverse hessian
    B = diag(matrix(1.0, nParam,1))

    iter = 0
    tol = 1.5 * 10^(-8)
    diff = tol +1
    while(norm_gx > tol & iter < max_iterations & diff > tol){
      #BFGS iterative update
    }
}
```

Listing 4.12: Initilization of BFGS method

The calculations of the BFGS algorithm for each iteration as described in chapter ?? can be divided into 5 steps: First the calculation of a search direction. Then a linear search in that direction. And finally the calculation of the updated formula. This updated is further divided in three parts. The calculation of s_k and y_k which are required to calculate the equation (2.45) and then the calculation of the equation itself:

```
#1. + 2. step
[new_weights, fxnew, stepsize] = linesearch ((-B%*%gx), fx, weights, X,
    Z, p, P, q, Q, s, solver)
gxnew = gradArimaCSS(new_weights, X, Z, p, P, q, Q, s, solver)

#3. step
sk = new_weights - weights # = stepsize * searchDirection
weights = new_weights

#4. step
yk = gxnew - gx

#5. step
FT = as.scalar(t(sk)%*%yk + t(yk)%*%B%*%yk) * sk%*%t(sk)/as.scalar(t(sk)
    %*%yk)^2
ST = (B%*%(yk%*%t(sk)) + (sk%*%t(yk))%*%B)/as.scalar(t(sk)%*%yk)
B = B + FT - ST

#preperation of next iteration
fx = fxnew
gx = gxnew
```

```
20 norm_gx = L2norm(gx)
21 iter =+ 1
```

Listing 4.13: Iterative update of BFGS method

The line search that is used to find the next best coefficients is implemented exactly as described in the chapter **??**. First the control parameters and α_0 are initialized with appropriate values. And then as long as the *Armijo-Goldstein* condition is fulfilled, the update $a_{k+1} = a_k \cdot r$ can be calculated:

```
1  linesearch = function (Matrix[Double] searchDirection, Double fx,
       Matrix[Double] weights, Matrix[Double] X, Matrix[Double] Z, Integer
       p, Integer P, Integer q, Integer Q, Integer s, String solver)return(
       Matrix[Double] new_weights, Double fxnew, Double ak){
2      #constrol parameters
3      c = 0.0001
4      r = 0.9
5
6      #stepsize
7      ak = 1.0
8
9      new_weights = weights + ak*searchDirection
10     fxnew = arima_css(new_weights, X, Z, p, P, q, Q, s, solver)
11
12     while(fxnew > fx +  c*ak*as.scalar(t(searchDirection)%*%
           searchDirection)){
13         ak = r*ak
14         new_weights = weights + ak*searchDirection
15         fxnew = arima_css(new_weights, X, Z, p, P, q, Q, s, solver)
16     }
17 }
```

Listing 4.14: Backtracking Line Search

5 Results

5.1 Correctness Test

The correctness of implementation of the ARIMA model has been tested for each sub-model AR, SAR, MA and SMA separately and also for the combinations of the sub-models.

For AR(p) the correctness tests have been carried out for all $p <= 14$ as well as $p = 100$ and have all been successful.

For the same data, the tests have also been run for SAR$(P)_s$. But in contrast to the previous test the implementation of R has restricted the maximum values for P and s, because the maximum lag that R allows is 350. Therefore $P \cdot s$ has to be smaller or equal to 350. For $s = 10$ the results were positive for $P <= 14$. However, when setting s to be equal to 33, the results showed that the implementation of the Seasonal Autoregressive model does not always provide correct values. Only for $P <= 4$ the test yielded positive results. But for $P > 5$ the test failed. And as P was increased further, the difference between the results of R and the DML script increased too. This might be a result of an improper implementation of the multiplicative seasonal AR or could be indicating that R is actually not using an multiplicative model but instead an additive one.

Previous test have shown that AR(p) and SAR$(P)_s$ for $s = 10$ should be working, the test for the combination of the non seasonal and seasonal Autoregressive model yielded some negative results nonetheless. When testing SAR$(p)(P)_s$ with $s = 10$, $p = 1$ and $P <= 14$ as well as for $p = 14$ and $P <= 1$ the tests were successful. But for all combinations of p and P where both parameters had values greater than two the tests failed. And again with the same pattern of increasing divergence in the results as could be seen before when setting s to 33.

In the case of both the MA and SMA model all test have failed starting from $q = 1$ or $Q = 1$. And again not completely, but only with a divergence that was increasing faster and starting earlier. Therefore the test for non seasonal as well as seasonal ARIMA have also yielded negative results in the same manner.

5.2 Performance Test

The performance tests have been run for time series with 1, 10 and 100 thousand rows as well as for series with 1 and 10 million rows. All the datasets have been used to fit AR(p) models with $p \in \{1, 2, 5, 10\}$ and the 1K dataset has also been fitted for AR(100) and AR(500). The other models have not been tested further, because the previous tests in chapter 5.1 have shown, that they weren't providing correct results. The results of the performance tests for the AR model can be seen in the figures 5.1 through 5.4:

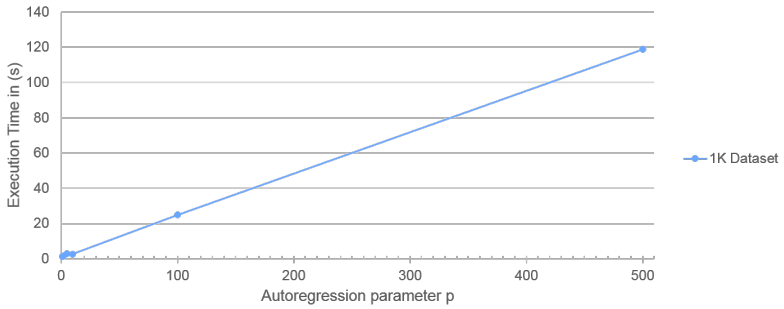

Figure 5.1: Execution time of AR(p) depending on p for 1K Dataset

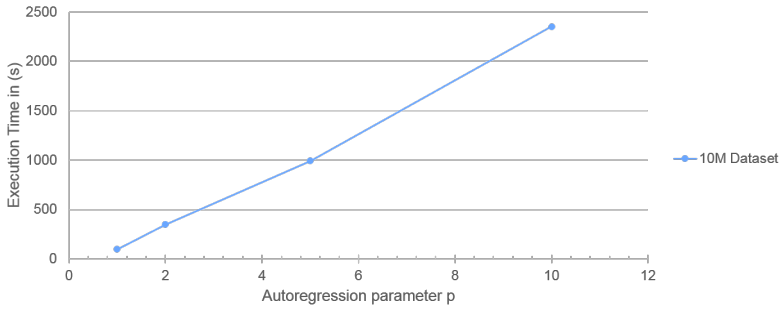

Figure 5.2: Execution time of AR(p) depending on p for 10M Dataset

The figures 5.1 and 5.2 both show a linear trend when increasing the parameter p. This is what was expected, because for the same time series increasing the parameter p means a linear increase in the complexity of the prediction function which is called at every iteration of the algorithm.

The same holds true for figures 5.3 and 5.4 which show how the execution time increases when increasing the size of the time series. And as can be seen the trend is also a linear one.

Figure 5.5 shows what parts of the script need the most execution time on the level of single instructions. Worth pointing out is the unexpected high spike for *Jacobi*, considering that the fitting of an AR model does not need a solver for systems of linear equations at all.

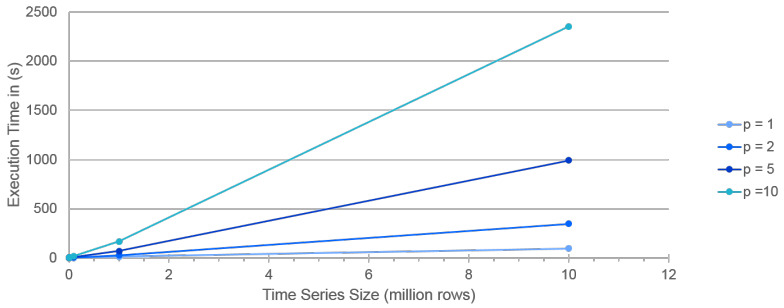

Figure 5.3: Execution time of AR(p) depending on size of time series for $p \in \{1, 2, 5, 10\}$

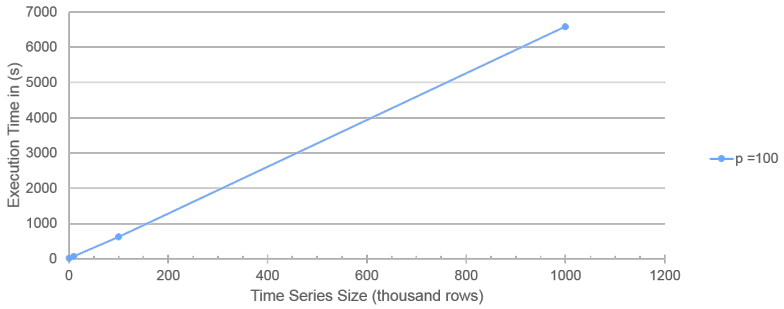

Figure 5.4: Execution time of AR(p) depending on size of time series for $p = 100$

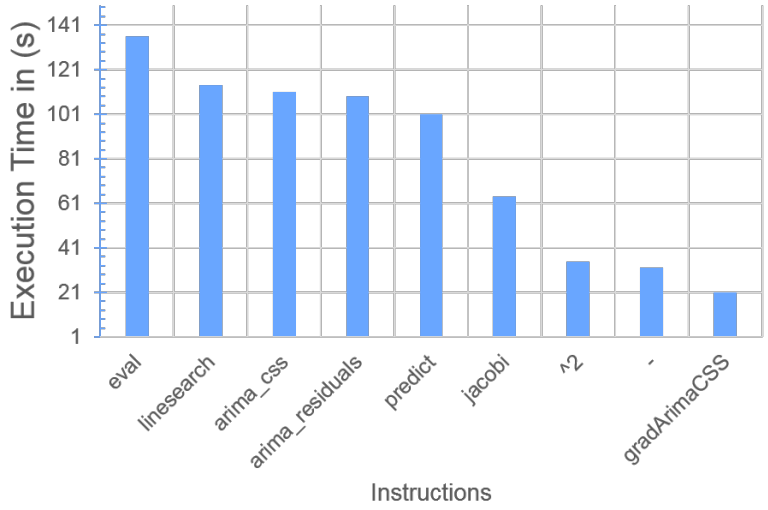

Figure 5.5: Heavy hitter instructions of AR(100)$for100K Dataset$

6 Conclusion

The goal of the project was to extend the offering of SystemML by providing an implementation of ARIMA. This was supposed to simplify time series analysis for data scientists that wished to use this model on larger datasets. Unfortunately this goal could not be fully achieved, because only the AR model has been implemented correctly.

This was mainly due to a lack of time that had been available to complete the project. Of course this wasn't the only reason as the methodologies which were decided upon at the begin of the project also factored in.

Implementing the ARIMA model for SystemML using DML brought all the advantages described in chapter 2.5 and 3.1. The most significant being the ability to use the same easy to write and expressive code to not only develop and test algorithms on a small scale but also having it scaled up the algorithm to run the same code on distributed machines, enabling large scale machine learning. But using DML also had its disadvantages: Debugging was made quite complicated compared to other languages, because of all the code optimizations that were going on under the hood. The debugger available could only be used from the terminal and had a rather small feature set. Of course one could also try to debug the code using print statements, but even though the error was not occurring before these statements were supposed to be executed, they sometimes weren't. This was a result out of the said code optimizations, that divided the code into segments which were run separately. And a 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. And even though the core syntax of DML is documented, some features are not or are lacking in detailed documentation. One example would be the *eval* function, that can be used to call a UDF by name which can be provided in a string. Another one would be the *source* function that can be used to load UDFs from other files. And on top of the lacking documentation the fact that the language itself is also still in development could be experienced on multiple occasions. Over the course of the project at least 5 bugs within the language itself stopped the further development of the algorithms for some time,

because they had to be first identified as bugs of the language and not the algorithm and then further singled out and reported so they could be fixed. Fortunately each bug that was reported was also fixed very fast. Nonetheless they still prolonged the development process.

Using R to verify the correctness of the implementation of the non seasonal ARIMA algorithm was made easy enough by the well documented package and language. All the parameters were described sufficiently and sometimes even explained in more details to better understand how they interacted with other parameters. However, this can not be said about the documentation of the parameters regarding the seasonal ARIMA model, because the very important distinction between the additive or multiplicative model has just been left out. So it was just assumed to be a multiplicative model, before noticing that this might have been a mistake. In spite of that R's implementation made finding bugs within the DML script easier because it not only provided the coefficients of the finished model but also the residuals when applying those coefficients. Furthermore a set of fixed coefficients could also be provided to R's ARIMA which would then calculate the residuals based on that, which made it possible to verify the *predict* function separately without having the optimizer on top making the difference in the results harder to interpret. However, there was one particular disadvantage when using R's implementation, because it didn't support any seasonal models with a lag that was bigger than 350. Meaning that neither $P \cdot s$ nor $Qcdots$ for additive models or $p + P\dot{s}$ nor $q + Q\dot{s}$ for multiplicative could be bigger than 350.

The performance tests could also run comfortably, because it could be done in the exact same way as before when developing the algorithm. The additional *-stats* argument also proved to be quite useful for identifying bottlenecks easily. Using this feature the performance of the algorithm could be improved by a factor of 4 because a particular instruction could be identified that needed a lot of computation time, but wasn't actually needed and could be removed completely form the script. For running a lot of performance tests with different configurations a automated testing script should be used instead of manually starting the single tests, which has been done in this case.

6.1 State of ARIMA in SystemML

The current version of the ARIMA training script is able to produce a valid AR(p) as well as a valid SAR$(P)_s$ model with the same accuracy and results of R's implementation. However the SAR$(p)(P)_s$ model is only working partially and any other form of the ARIMA model including Moving Average does not provide accurate results at all. In the current state the models are obtained by using the BFGS optimizer for minimizing the Conditional

Sum of Squares and the Jacobi solver for solving systems of linear equations. And the gradient of $ARIMA_CSS$ is calculated exactly instead of using the finite differencing method to approximate it. And finally, the model is also only working for univariate time series.

6.2 Next Steps for ARIMA in SystemML

Obviously the implementation of ARIMA is not yet finished. The first next step would be to debug the seasonal Autoregressive model which is still providing results that diverge from R's results when increasing both p and P. This might be achieved by implementing the an additive seasonal model instead of a multiplicative one. Beforehand the implementation of R should be checked in more detail to figure out what seasonal model it is implementing. Then of course there is also the Moving Average models that are still not working which might be caused by an improper implementation of *Jacobi* or simply because using an inexact solver is just not sufficient in this particular case.

Additionally there also need to be proper unit tests to be implemented to automatically compare the results of the DML script to the ones of R to make sure, that further changes to the script don't break what is already working.

And as soon as the ARIMA model is implemented fully there need to be further and more detailed performance and scalability tests both for the non seasonal as well as the seasonal model. And they should also be done for even bigger datasets.

After the implementation of the basics, the script can also be further improved by adding more features. The first one should be the calculation of the Root-Mean-Square Error (RMSE) and then it should also be considered to add the Maximum Likelihood estimator to be used as alternative for the Conditional Sum of Squares. Additionally there should also be more optimizers provided like Nelder-Mead, CG, etc. And finally there is most likely going to be room for performance improvements. One that could already be done now is the elimination of the call to the *Jacobi* solver when only fitting an Autoregressive model that doesn't even need any systems of linear equations to be solved.

Acronyms

NASA National Aeronautics and Space Administration

SMAP Soil Moisture Active Passive

ARIMA Autoregressive Integrated Moving Average

SARIMA Seasonal Autoregressive Integrated Moving Average

ARMA Autoregressive Moving Average

AR Autoregressive

SAR Seasonal Autoregressive

MA Moving Average

SMA Seasonal 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

CSS Conditional Sum of Squares

DML Declarative Machine Learning Language

iid independent and identically distributed

API Application Programming Interface

UDF User-Defined Function

RMSE Root-Mean-Square Error

Bibliography

[1] NASA, *SMAP Specifications*
Retrieved from https://smap.jpl.nasa.gov/observatory/specifications/
Retrieved on 08/02/2018

[2] Saranya Anandh, 23 June 2016, *Everything About Time Series Analysis And The Components of Time Series Data*
Retrieved from https://www.linkedin.com/pulse/everything-time-series-analysis-components-data-saranya-anandh/
Retrieved on 07/18/2018

[3] Amir Kalron, 12 February 2018, *How do Hadoop and Spark Stack Up?*
Retrieved from https://logz.io/blog/hadoop-vs-spark/
Retrieved on 07/18/2018

[4] Apache Software Foundation, *What is SystemML*
Retrieved from https://systemml.apache.org//specifications/
Retrieved on 05/21/2018

[5] Dr. Avishek Pal, Dr. PKS Prakash, September 2017
Practical Time Series Analysis
Retrieved from https://www.safaribooksonline.com/library/view/practical-time-series/9781788290227/3526bc2a-96a5-4c11-97da-a0b5c74c4caf.xhtml

[6] Brockwell, P. J. and Davis, R. A. (2002)
Introduction to Time Series and Forecasting, 2nd Edition
Springer, New York
ISBN 0-387-95351-5

[7] D.S.G. POLLOCK
Time Series and Forecasting, LECTURE 2, Seasons and Cycles in Time Series
Retrieved from
https://www.le.ac.uk/users/dsgp1/COURSES/TSERIES/2CYCLES.PDF

[8] *Additive and Multiplicative Models*
Retrieved from http://www-ist.massey.ac.nz/dstirlin/CAST/CAST/Hmultiplicative/multiplicative1.html

[9] Rob J Hyndman and George Athanasopoulos (2018)
Forecasting: Principles and Practice, 2nd Edition
OTexts
ISBN-10: 0987507109
Retrieved from https://otexts.org/fpp2/

[10] IBM
Characteristics of Time Series - Pulses and Steps Retrieved from
https://www.ibm.com/support/knowledgecenter/SS3RA7_17.0.0/
components/dt/timeseries_pulses.html

[11] Robert Nau
Stationarity and differencing Retrieved from
https://people.duke.edu/~rnau/411diff.htm

[12] Suhartono (2011)
Time Series Forecasting by using Seasonal Autoregressive Integrated Moving Average: Subset, Multiplicative or Additive Model, Volume 7, Pages 20-27
Retrieved from
http://thescipub.com/abstract/10.3844/jmssp.2011.20.27

[13] Florian Pelgrin (2011)
Lecture 4: Estimation of ARIMA models Retrieved from
https://math.unice.fr/~frapetti/CorsoP/Chapitre_4_IMEA_1.pdf

[14] Neil Shephard (September 1997)
The relationship between the conditional sum of squares and the exact likelihood for autoregressive moving average models Retrieved from
https://www.nuffield.ox.ac.uk/economics/papers/1997/w6/ma.pdf

[15] Kris Hauser (January 2012)
Lecture 6: Multivariate Newton's Method and Quasi-Newton methods Retrieved from http://people.duke.edu/~kh269/teaching/b553/newtons_method.pdf

[16] Kurt Bryan
Quasi-Newton Methods Retrieved from
https://www.rose-hulman.edu/~bryan/lottamath/quasinewton.pdf

[17] Prof. Dr. Karl Stroetmann, (August 2017)
Analysis, Chapter 6.5 Retrieved from https://github.com/karlstroetmann/Analysis/blob/master/Skript/analysis.pdf

[18] Boehm et al (2011)
SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs Retrieved from
https://www.researchgate.net/profile/Yuanyuan_Tian/publication/266911194_SystemML's_Optimizer_Plan_Generation_for_Large-Scale_Machine_Learning_Programs/links/543ec4290cf2e76f02243522.pdf

[19] Apache Software Foundation,
SystemML Documentation - DML Language Reference Retrieved from https://systemml.apache.org/docs/0.12.0/dml-language-reference.html

[20] *RDocumentation - ARIMA* Retrieved from https://www.rdocumentation.org/packages/stats/versions/3.1.1/topics/arima