Fachhochschule Südwestfalen
University of Applied Sciences

South Westphalia University
Department of Engineering and Economics

Datenanalyse in Big Data
Supervisor: Prof. Dr. Buchwitz

Distributed OLS 1a

Casimir Giesler, Hendrik Metzner, Sinan Eker,
Patrick Adrian Ulbrich

Meschede
9th October 2023

Wir geben Impulse

Declaration of Authorship

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references. This paper was not previously presented to another examination board and has not been published.

I am aware that the thesis in digital form can be examined for the use of unauthorized aid and in order to determine whether the thesis as a whole or parts incorporated in it may be deemed as plagiarism. For the comparison of my work with existing sources I agree that it shall be entered in a database where it shall also remain after examination, to enable comparison with future theses submitted.

Meschede, 9th October 2023.

Casimir Giesler
MatNr: 123454678
Email: giesler.casimir@fh-swf.de

Hendrik Metzner
MatNr: 123454678
Email: metzner.hendrik@fh-swf.de

Sinan Eker
MatNr: 123454678
Email: eker.sinan@fh-swf.de

Patrick Adrian Ulbrich
MatNr: 123454678
Email: ulbrich.patrick@fh-swf.de

This Declaration must be signed by all listed Authors to be valid.

Contents

1 Simulation of a Dataset

In order to generate a large dataset which fulfills the requirements ($n \gg 10^9, k \gg 10^5$), the generation of the values needs to be done in a distributed fashion. PySpark does not have a pre-defined function to generate an entire dataset suited for OLS, therefore this function is implemented manually. At first, the following values need to be initialized:

- `n` - number of rows/samples
- `k` - number of columns/features
- $\vec{\beta}$ - beta, the coefficients of the function
- `cov` - a covariance vector that determines the covariance to the first column for each column

In this implementation, `n` and `k` need to be set by the user while $\vec{\beta}$ and `cov` are generated randomly by numpy. For generating the actual dataset, `pyspark.mllib.random.RandomRDDs.normalVectorRDD(sparkContext, n, k)` is used. This function creates an `RDD` containing `n` vectors, each containing `k` entries, where each entry is generated from a standard-normal distribution.

After generating this random noise matrix, the user-defined-function `createRow(noise)`, as implemented in *DAiDB.ipynb* in the appendix, is applied to the `RDD`, which returns two values, $\vec{x}$ (1) and y (2).

With `noise` as ϵ and `cov` as c:

$$\vec{x} = (\epsilon_0, \epsilon_0 c_1 + \epsilon_1, \ldots, \epsilon_0 c_i + \epsilon_i) \tag{1}$$

$$y = \vec{x} \cdot \vec{\beta} \tag{2}$$

Applying this function will produce an RDD where the first element is the x-vector while the second element is the target variable.

Therefore, the final outcome is a feature matrix (consisting of n $\vec{x}$ vectors) that consists out of k columns, where each column is linearly dependent on the first column, with additional noise. An example of a distribution is shown in the figure 1.

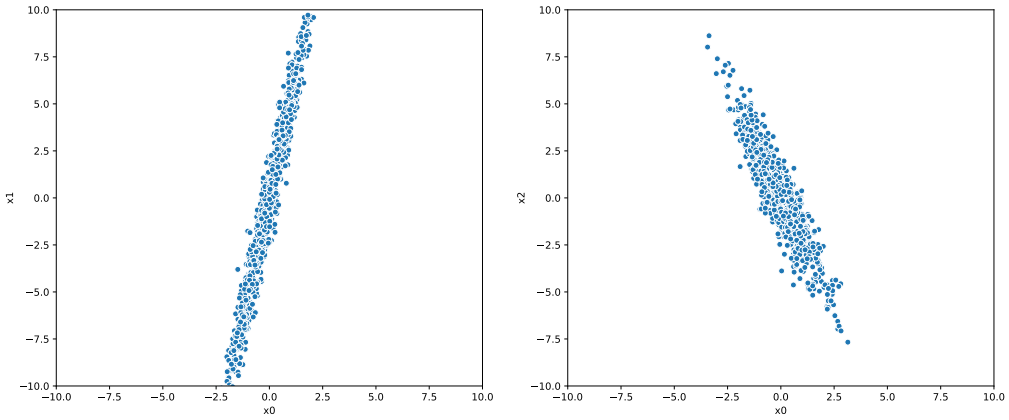

Figure 1: *exemplary generated dataset*

2 Singular Value Decomposition (SVD)

In the following two subchapters the singular value decomposition (SVD) will be briefly explained. In the first subchapter the mathematical background will be layed out. In the second subchapter references to the implementation of SVD in PySpark will be made. The focus is set on the main things that are important for understanding the general concept of SVD and the implementation in PySpark. References to additional mathematical proofs are also provided.

2.1 Mathematical Background

A singular value decomposition (SVD) is mainly used to determine the pseudo-inverse of a matrix to solve the linear system of equations that is represented by the matrix. A pseudo-inverse is a generalized inverse matrix. According to Burg et al. (2012, p. 354, definition 3.37), a matrix G must satisfy the following conditions (3) and (4) to be referred to as a pseudo-inverse:

$$AGA = A \tag{3}$$

$$GAG = G \tag{4}$$

To be called a *Moore-Penrose-Inverse* the following condition (5) also has to be met.

$$AG \text{ and } GA \text{ are symmetrical} \tag{5}$$

The Moore-Penrose inverse is denoted by $A^{\dagger}$.

Furthermore, the general form of a SVD can be written as shown in equation (6) (Burg et al. 2012, p. 354, equation 3.425).

$$A = U \begin{bmatrix} S & 0 \\ 0 & 0 \end{bmatrix} V^T \tag{6}$$

An alternative way of writing this equation is shown in equation (7) (Duvvuri & Singhal 2016, p. 251 - 252).

$$A = U\Sigma V^T \tag{7}$$

The matrices have the following properties:

- A is the original matrix with *m-rows* and *n-columns*
- U is a column-orthonormal matrix with *m-rows* and *r columns*
- V^T is the transpose of a column-orthonormal matrix with *n-rows* and *r columns*
- Σ is an $r \times r$ diagonal matrix containing non-negative real numbers

The vectors in U are also called the left-singular vectors of A. Respectively, the vectors in V are called the right-singular vectors of A (Apache Spark 2017). The elements of $\Sigma \in \text{Mat}(r; R)$ are non-negative and arranged in descending order. These diagonal values are called the singular values of Matrix A, which is why the equation (6) is called the singular value decomposition of A.

It is further assumed that for each matrix $A \in \text{Mat}(m, n; R)$ there is exactly one Moore-Penrose inverse. The following equation (8) is from Burg et al. (2012, p.355 equation 3.427). The complete

mathematical proof of this assumption is not part of this study and can be found in Burg et al. (2012, p. 355 - p. 357).

$$A^{\dagger} = V \begin{bmatrix} S^{-1} & 0 \\ 0 & 0 \end{bmatrix} U^T \in \text{Mat}(n, m; \mathbb{R}) \tag{8}$$

The final step in solving the system of linear equations is to find the optimal solution by utilizing the Moore-Penrose-Inverse. According to Burg et al. (2012, p. 357 Satz 3.86), with the Moore-Penrose-Inverse $A^{\dagger} \in \text{Mat}(n, m; \mathbb{R})$, an original matrix $A \in \text{Mat}(n, m; \mathbb{R})$ and a given $b \in \mathbb{R}^m$, the following equation (9) is the solution set of the linear optimization problem.

$$x = A^{\dagger}b + y - A^{\dagger}Ay \quad \text{mit} \quad y \in \mathbb{R}^n \tag{9}$$

Derived from that the optimal solution is shown in equation (10).

$$\hat{x} = A^{\dagger}b \tag{10}$$

As shown, the SVD is mainly a way to calculate the Moore-Penrose-Inverse, which then is used to find the optimal solution for the given matrix. There are multiple methods to calculate the SVD to determine the corresponding matrices shown in equation (6). Typical methods are:

1. Jacobi Method
2. Golub-Kahan-Reinsch algorithm
3. Divide-and-Conquer method

One way of thinking about the singular value decomposition is that the matrix Σ in equation (7) contains the strength of the corresponding components in the two other matrices (Duvvuri & Singhal 2016, p. 252). So one additional way of approximately solving numerical problems (or doing lossy image or data compression in general) is to set the values in the matrix Σ below a lower magnitude threshold to zero to reduce the number of relevant rows in the remaining two matrices.

2.2 implementation in PySpark

Apache Spark uses two ways to perform the SVD, depending on the absolute size of the number of rows n or the size of n compared to the number of columns k (Apache Spark 2023c). In the case that n is small (n < 100) or n is small compared to k (n/2 < k) "the Gramian matrix (is computed) first and then the top eigenvalues and eigenvectors are locally computed on the driver" (Apache Spark 2023c). In all other cases $(A^T A)v$ is calculated "in a distributive way and send (...) to ARPACK to compute (ATA)'s top eigenvalues and eigenvectors on the driver node" (Apache Spark 2023c).

It is possible to use an additional optimization step to decrease the calculation time by only taking the top k singular values into consideration as described in Duvvuri & Singhal (2016, p. 252) by setting the parameter k to a specific value (Apache Spark 2017). In our implementation we chose to not use this optimization to arrive at the most accurate solution.

The values for S, V and U in relation to equation (6) are printed in the Jupyter Notebook *DAiBD.ipynb* that can be found in the appendix. It is important to note that the matrix U is a distributed RowMatrix. Since a distributed RowMatrix stores data in a distributed manner, it lacks the concept of row order, which is necessary for transposing a matrix. As we require the

transpose U^T of matrix U for equation (8), the RowMatrix U is transformed into a CoordinateMatrix and transposed. After transposing the matrix it is transformed back into a RowMatrix and saved as U_T.

In the final *Results* block in the notebook the results of the SVD are calculated. In the first line *step1*, the product of the transpose of the matrix U, obtained from the SVD, and the target variable y is computed. y is a DenseMatrix and was created in the QR part of the notebook. In *step2*, the result from step1 is transformed into a NumPy array. This array is flattened using the ravel() method and then divided by the corresponding singular values previously computed and stored in *svd.s*. Next the right singular matrix V is transformed into a NumPy matrix. Subsequently the linear regression coefficients *SVD_coeeffs* are calculated by performing matrix multiplication between the right singular matrix V and the transformed vector from step2. The coefficients are also flattened using ravel(). The SVD section concludes by printing the elapsed time for the SVD-based linear regression, by displaying the calculated coefficients based on the SVD, and by showing the real values of the coefficients, allowing for a comparison between the predicted and actual coefficients.

3 QR Dekomposition:

This section describes the theoretical basics of QR decomposition. Following on from that, the second part deals with the mathematical basics. In the third chapter, instructions for the implementation of QR decomposition in PySpark are given. The focus here is on the central aspects that are important for a basic understanding of the QR concept as well as the implementation in PySpark.

3.1 Theoretical basics

Note that the Gram-Schmidt method is used to transform a linearly independent set of vectors into an orthonormal vectorset. In other words, a vector set that has the standard of unity and is orthogonal to each other.

Given a K x L matrix A, its columns are labeled

$$A_1, ..., A_L.$$

When these columns are linearly independent, they can be transformed into a set of orthonormal column vectors

$$Q_1, ..., Q_L$$

using the Gram-Schmidt method, in which normalization and projection steps alternate. These steps will be presented in the next chapter about mathematical basics (Taboga n.d.).

3.2 Mathematical basics

As already mentioned in the theoretical basics, the QR decomposition is used to describe a matrix with linear independent columns as a product of a matrix Q with orthonormal columns and an upper triangular matrix. According to Burg et al. (2012, p. 310, definition 3.69), a QR decomposition can be performed under the following conditions:

Any regular matrix A can be decomposed into a product $A = QR$, where Q is an orthogonal matrix and R is a regular triangular matrix. Q is a product of at most $(n-1)$ reflections.

Proof:

Assuming that (11) and (12) are given, if (13) is true, then set S(1) := E. If however (13) is true, then we form with (14) the reflection (15).

$$A = [a_1, ..., a_n] \tag{11}$$

$$E = [e_1, ..., e_n] \tag{12}$$

$$a_1 = |a_1|e_1 \tag{13}$$

$$u = \frac{a_1 - |a_1|e_1}{|a_1 - |a_1|e_1|} \tag{14}$$

$$S^{(1)} := S_u \tag{15}$$

For this we calculate (16) and thereof (17) with (18).

$$S^{(1)}a_1 = |a_1|e_1 \tag{16}$$

$$A^{(2)} := S^{(1)}A = \begin{bmatrix} r_{11} & * \\ 0 & A \end{bmatrix} \tag{17}$$

$$r_{11} = |a_1| \tag{18}$$

The same step is now performed for **A2**, which means that a mirror **S2** is formed in (19) (or **S2 = unit matrix**), so that in **S2 A2** the first column is filled only with an **r22 > 0**. All the other elements of this column are zero. With (20) follows (21).

Proceeding in this way, in the end we obtain (22), where R is a right triangular matrix. It is regular because the left side is regular. With (23) follows **A = QR** and therefore the proof of the theorem.

$$\mathbb{R}^{n-1} \tag{19}$$

$$S^{(2)} = \begin{bmatrix} 1 & 0 \\ 0 & S_2 \end{bmatrix} \tag{20}$$

$$A^{(3)} := S^{(2)}A^{(2)} = \begin{bmatrix} r_{11} & * & ...* \\ 0 & r_{22} & *...* \\ & & A_3 \end{bmatrix} \tag{21}$$

$$S^{(n-1)}S^{(n-2)}...S^{(2)}S^{(1)}A = R \tag{22}$$

$$\mathbb{Q} = S^{(1)}S^{(2)}...S^{(n-1)} \tag{23}$$

3.3 Implementation in PySpark

In the implementation in PySpark, according to the documentation from Apache Spark 2023, a RowMatrix is created from a vector instance. With this RowMatrix, it is possible to perform various statistical summaries of the columns as well as decompositions. An important decomposition in this scope is the QR decomposition, which takes the form A = QR. Here Q stands for an orthogonal matrix and R for an upper triangular matrix. This type of decomposition enables efficient calculations and analysis of large datasets in Spark environments.

In Spark, there are several functions for calculating QR decomposition, depending on the property of the matrix at hand. In the present use case of a RowMatrix, the tallSkinnyQR() function is best suited because it is optimized specifically for RawMatrices. The computeQR() method is suitable as a generalist for any matrix, but is not optimized for any particular shape and is therefore misfit. The tallSkinnyQR() function has the boolean parameter computeQ as input parameter. With computeQ = True, both R-matrix and Q-matrix are computed. With computeQ = False only the R-matrix is calculated. For the calculation of the betas both matrices are needed, therefore the tallSkinnyQR method is passed the boolean TRUE as input parameter. The Apache Spark documentation was used as a literature source, in particular the page on the

RowMatrix class (Apache Spark 2023c). Since the dataMatrix has dimensions n x k, the QR.Q matrix has dimensions n x n and the upper triangular matrix has dimensions n x k.

In the next step the inverse of the R matrix is calculated. Since PySpark is specialized for the calculation of distributed data sets, there is no direct method to calculate the inverse. For the local calculation of the inverse the method np.linalg.inv() of the numpy library is suitable, because numpy is optimized for the numerical calculations of matrices, vectors and arrays. To use the np.linalg.inv() method correctly, the R matrix is converted to a numpy matrix using the np.asmatrix() function. The np.asmatrix() again expects a numpy array as input parameter to perform the conversion. For this reason the result is passed to QR.R.toArray() which converts the R matrix into a numpy array. The dimension for the inverse of the R matrix is k x k. The inverse is a smaller dimension compared to "n".

In the next step, the transpose of the Q matrix is formed. Since the RowMatrix in PySpark does not have a transpose method, other distributed approaches are needed. For this, a CoordinateMatrix is created using so-called "MatrixEntry" objects. With QR.Q.rows.zipWithIndex() an index is passed to each vector in the RDD. This is necessary to correctly assign the rows and columns of the transposed matrix later. With flatMap() a function is applied to all elements of the RDD. Since as described "MatrixEntry" objects are necessary for the creation of the CoordinateMatrix, the transformation of the elements in the RDD into a list of "MatrixEntry" objects is done with the help of flatMap() (Apache Spark 2023b). This approach allows efficient computation of the transposed Q-matrix in a distributed Spark environment, especially to ensure scalability and performance. For the transposed Q-matrix the dimension k x n follows.

Subsequently, the values of the dependent variable "y" are represented as a single-column matrix. In order to multiply "y" with a RawMatrix, a compatible data structure is required. PySpark offers the DenseMatrix as a suitable multiplier. To create the DenseMatrix accordingly, the data array is required in addition to the input parameters numRows and numCols (Apache Spark 2023a). To achieve this, dataDF.select("y").toPandas().to_numpy().ravel() converts the column "y" from the DataFrame "dataDF" into a pandas dataframe and finally into a Numpy array. With ravel() an exclusively one-dimensional vector is stored.

Finally, the matrix multiplications are performed. For this, in the first step "Q_T" and "y" are multiplied with multiply, a function from the PySpark framework. With rows.collect() the calculations of the Spark driver nodes are returned to a local data structure, in this case a Python list is suitable. With np.matmul(), the matrices are multiplied together locally after appropriate conversion. Thus the matrix multiplication k x k * k x 1 is available. In the end, the results of the OLS estimation, the true betas and the total execution time are output. The implementation enables efficient and distributed processing of matrix operations by using Apache Spark, and local matrix multiplication by applying the Numpy library.

4 LU Decomposition

The first paragraph explains the mathematical approach, with particular emphasis on its use in linear systems. The second paragraph explains the divide and conquer approach to LU decomposition of large matrices and how the PySpark and Scipy libraries are used.

4.1 Mathematical Background

In LU decomposition, a matrix A is transformed into the product of matrices L and U. The mathematical formula is:

$$A = LU$$

If problems arise during the application of the transformations, such as a division by 0, a permutation matrix can be used. This permutation matrix also increases the robustness with limited accuracy as well as the numerical stability (Lu 2022, p. 23). The corresponding form is:

$$A = PLU$$

The matrices A, P, L and U are defined as follows:

- A is the origin matrix
- L is a lower triangular matrix with 1 at the diagonal, and 0 above the diagonal
- U is an upper triangular matrix
- P is a permutation matrix

The LU decomposition is often used to calculate the inverse of nonsingular matrices or to calculate the determinant of a matrix. It is also used for solving linear systems (Lu 2022, p. 31-33).

To solve a linear system like $Ax = b$ using LU decomposition, the following steps must be performed as in (Furlan 1997, p. 4):

- 1. Calculate the LU decomposition of A : $A = PLU$
- 2. Solve $P\vec{z} = \vec{b}$ with $\vec{z} = \mathbf{P}^\top \vec{b}$
- 3. Solve $L\vec{y} = \vec{z}$ recursive, start with y_1
- 4. Solve $U\vec{x} = \vec{y}$ recursive, start with x_n

Then the coefficients can be taken from the solution vector.

4.2 Implementation in PySpark

Since there is no direct function for LU decomposition like for QR or SVD included in PySpark, the mapreduce approach like (Pozdnoukhov & Kaiser 2011) is taken form this. The dataset is divided into equal parts and then the LU decomposit is performed separately for each of these parts.

In the program, the first step is to create the function which calculates the coefficients using the LU decomposition. As an input the function gets a pandas dataframe with the matrix A (features) and the corresponding values b (y). To use the *lu_factor* function (SciPy 2023a) and the *lu_solve* function (SciPy 2023b) from the Scipy library, first a Numpy array is created from A. Following the principle from (4.1), the function *lu_factor* first calculates the LU decomposition from A and stores the LU matrix and the P matrix. Then the *lu_solve* function performs steps 2,3 and 4 from (4.1). The calculated coefficient values x (betas) are returned with the number of rows of the partial data set A (sampleCounts) as Pandas DataFrame.

To split the dataset, the function *.groupBy(spark_partition_id)* is used. This function splits the dataset into n equal sized partitions. n corresponds to the number of different partitions of the RDD. For using the function *.applyInPandas()* each of these data partitions is passed as a Pandas data frame to the function described above. The computation now takes place paralelly in the individual Spark instances.

The return of the function *.applyinpands()* is a DataFrame which contains the coefficients (betas) for each of the parts of the data set, calculated by *lu_solve*, and the number of records (sampleCounts) of the data part. The number of records is required to calculate a weighted average of the resulting coefficients of the return DataFrame. This weighted average is calculated at the end of the program and contains the result vector of the linear equation system.

5 Performance measurement

To evaluate the scalability of the different implementations, both in rows and columns as well as in cluster size, the DUS Airports Hadoop cluster is used. The Spark native implementation `pyspark.ml.regression.LinearRegression` is used as baseline. Each permutation of number of rows, number of columns, number of nodes and applied algorithm is measured at least five times.

5.1 Data scalability

The test shows, that the custom implementations perform significantly worse than the PySpark implementation, except the map-reduce LU implementation which is the only method which performs better than the PySpark implementation. QR consistently performs worst, as visualized in figures 2 and 3. Besides the worse performance, the QR and SVD implementations still show a linear algorithmic complexity and therefore are scaleable to some extend.

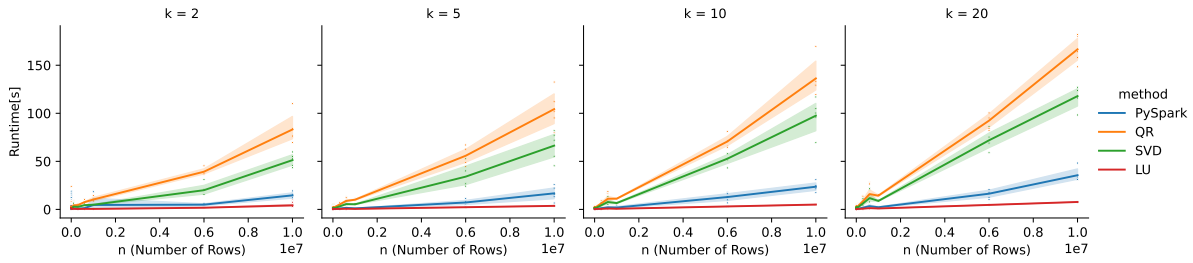

Figure 2: *Runtime comparision for the different implementations with linear scale*

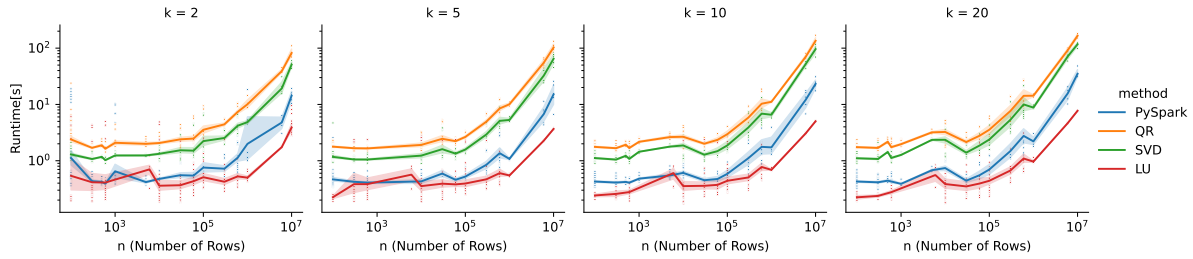

Figure 3: *Runtime comparision for the different implementations with logarithmic scale*

5.2 Node scalability

To evaluate the horizontal compute scaling capabilities, we analized the runtime for the same amount of data with different cluster sizes. As expected, the results show that increasing the cluster size reduces the runtime, with the effectiveness increasing as the amount of data increases. Figure 4 shows, that performing the calculations on a cluster for `n=100` increases the runtime, which is obvious since a lot of overhead work has to be done in order to distribute the tiny workload, which itself takes a lot longer than the actual computation. The map-reduce LU implementation obviously fails on more than eight nodes, since the rows are evenly distributed in the cluster and each node does not have enough datapoints to compute the coefficients. For $n = 10^4$ eight node seems to be an efficient sizing, for $n = 3 \cdot 10^5$ 32 nodes bring a considerable

performance improvement, while the improvement from 32 to 64 nodes for $n = 6 \cdot 10^5$ does not improve computation time significantly.

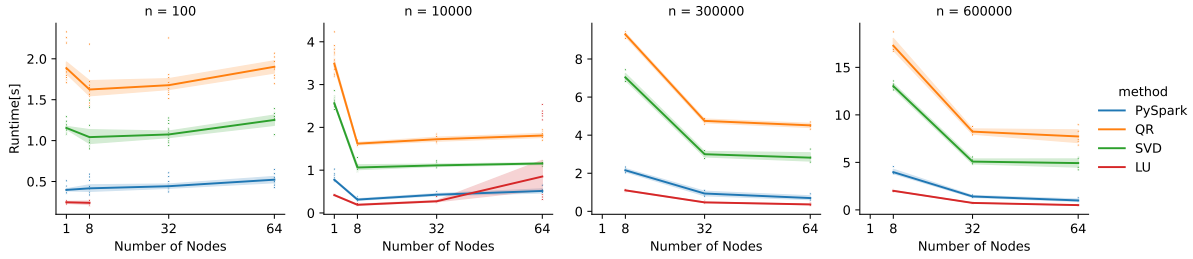

Figure 4: *Comparision of runtimes for the same amounts of data on different cluster sizes, with k=10*

5.3 Conclusion

The results show, that the implemented methods are scaleable, with ~$O(N)$ for SVD and QR and ~$O(log(N))$ for the map-reduce LU implementation (Fig. 2). Each implementation scales well with increasing cluster sizes.

6 Appendix

The following 3 files contain all the relevant code and can be found in the folder named “code”.

1. DAiBD.ipynb

This file contains the code for generating the dataset. Additionally it contains the implementation of the PySpark linear regression as a baseline as well as the implementation of the QR, SVD and LU regressions.

2. DaiBD_yarn_test.ipynb

This file contains the code for testing all implementations from the previous file. It saves the results in a csv for analyzing.

3. analize_test_results.ipynb

This file contains the code for analyzing and visualizing the previous results.

7 Appendix - Personal Contributions

Casimir Giesler

- code PySpark linear regression
- code QR
- code SVD
- code LU
- code analize_test_results.ipynb
- code DaiBD_yarn_test.ipynb
- Simulation of a Dataset
- performance measurement data scalability
- performance measurement node scalability
- performance measurement conclusion

Hendrik Metzner

- code LU
- LU decomposition mathematical background
- LU decomposition implementation in PySpark

Sinan Eker

- QR decomposition theoretical basics
- QR decomposition mathematical basics
- QR decomposition implementation in PySpark

Patrick Adrian Ulbrich

- SVD mathematical background
- SVD implementation in PySpark
- consolidation of all parts, formating and proof-reading paper

Technical Appendix

```
Sys.time()
```

```
## [1] "2023-10-09 16:30:50 CEST"
```

```
sessionInfo()
```

```
## R version 4.2.2 (2022-10-31 ucrt)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 10 x64 (build 22621)
##
## Matrix products: default
##
## locale:
## [1] LC_COLLATE=German_Germany.utf8  LC_CTYPE=German_Germany.utf8
## [3] LC_MONETARY=German_Germany.utf8 LC_NUMERIC=C
## [5] LC_TIME=German_Germany.utf8
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] fhswf_0.0.3
##
## loaded via a namespace (and not attached):
##  [1] bookdown_0.31   digest_0.6.31   lifecycle_1.0.3 magrittr_2.0.3
##  [5] evaluate_0.19   rlang_1.0.6     stringi_1.7.12  cli_3.6.0
##  [9] rstudioapi_0.14 vctrs_0.5.1     rmarkdown_2.19  tools_4.2.2
## [13] stringr_1.5.0   glue_1.6.2      xfun_0.36       yaml_2.3.6
## [17] fastmap_1.1.0   compiler_4.2.2  htmltools_0.5.4 knitr_1.41
```

References

Apache Spark (2017). Apache Spark pyspark.mllib.linalg.distributed.IndexedRowMatrix documentation. https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.mllib.linalg.distributed.RowMatrix.html#pyspark.mllib.linalg.distributed.RowMatrix.computeSVD.

Apache Spark (2023a). Apache Spark pyspark.ml.linalg.DenseMatrix. https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.linalg.DenseMatrix.html.

Apache Spark (2023b). Apache Spark pyspark.mllib.linalg.distributed.CoordinateMatrix. https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.mllib.linalg.distributed.CoordinateMatrix.html.

Apache Spark (2023c). Dimensionality Reduction - RDD-based API. https://spark.apache.org/docs/latest/mllib-dimensionality-reduction.html#performance.

Burg, K, H Haf, F Wille & A Meister (2012). *Höhere Mathematik für Ingenieure Band II - Lineare Algebra*. Berlin Heidelberg New York: Springer-Verlag.

Duvvuri, S & B Singhal (2016). *Spark for Data Science* -. Birmingham: Packt Publishing Ltd.

Furlan, P (1997). Zusätze zum gelben Rechnenbuch LU Zerlegung. *Verlag Martina Furlan Dortmund*.

Lu, J (2022). *Matrix Decomposition and Applications*.

Pozdnoukhov, A & C Kaiser (Nov. 2011). Scalable local regression for spatial analytics. In: *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM.

SciPy (2023a). *Scipy API referenc lu factor*. Version 1.11.2. https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.lu_factor.html.

SciPy (2023b). *Scipy API referenc lu solve*. Version 1.11.2. https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.lu_solve.html#scipy.linalg.lu_solve.

Taboga, M (n.d.). QR decomposition (). https://www.statlect.com/matrix-algebra/QR-decomposition.