Benjamin Bluhm | Jannic Cutura

Econometrics at Scale: Spark Up Big Data in Economics

SAFE Working Paper No. 266

Econometrics at Scale: Spark Up Big Data in Economics[*]

Benjamin Bluhm[†] Jannic Cutura[‡]

February 6, 2020

Abstract

This paper provides an overview of how to use "big data" for economic research. We investigate the performance and ease of use of different Spark applications running on a distributed file system to enable the handling and analysis of data sets which were previously not usable due to their size. More specifically, we explain how to use Spark to (i) explore big data sets which exceed retail grade computers memory size and (ii) run typical econometric tasks including microeconometric, panel data and time series regression models which are prohibitively expensive to evaluate on stand-alone machines. By bridging the gap between the abstract concept of Spark and ready-to-use examples which can easily be altered to suite the researchers need, we provide economists and social scientists more generally with the theory and practice to handle the ever growing datasets available. The ease of reproducing the examples in this paper makes this guide a useful reference for researchers with a limited background in data handling and distributed computing.

Keywords: Econometrics, Distributed Computing, Apache Spark

[*]An earlier version of this paper circulated as "Time Series Econometrics at Scale: A Practical Guide to Parallel Computing in (Py)Spark". We would like to thank Sanjiv Das, Frauke Kreuter and Satachit Sagade as well as participants from the *GRADE Workshop on Big Data in the Social Sciences*, the *Bank of England Conference on Modelling with Big Data and Machine Learning: Interpretability and Model Uncertainty* (2019), participants of the *Columbia University Data Science Institute Financial and Business Analytics Center Poster Session November 2019* and members of the *International Monetary Funds' BigData@Fund Community of Pratice* for helpful comments and suggestions. We gratefully acknowledge a travel grant sponsored by the Bank of England. We gratefully acknowledge research support from the Leibniz Institute for Financial Research SAFE. All remaining errors are our own.

[†]*Data Science Freelancer*, Email: `benjaminbluhm@gmail.com`

[‡]*Goethe University Frankfurt*, Email: `janniccutura@gmail.com`

Contents

1 Introduction

Research in economics and finance moved towards ever larger datasets and computationally more advanced methods, including statistical methods borrowed from the *machine learning* (ML) literature (Hamermesh 2013; Einav and Levin 2014). A growing number of papers discusses the potential and limits of modern data analysis frameworks such as ML algorithms (Varian 2014; Mullainathan and Spiess 2017), text as data (Gentzkow et al. 2019; Grimmer and Stewart 2013) and the progress on inference methods in high-dimensional settings (Athey and Imbens 2017; Kleinberg et al. 2015). Finally, Fernández-Villaverde and Valencia (2018) provide an introduction to parallel computing for economics. While most of these works describe the underlying algorithms at great length, there is almost no guidance on how to handle the typically very large[1] datasets used for these tools. As data availability is very likely to grow in the foreseeable future, the inability to handle big data sets poses a severe challenge to empirical economic research.[2]

This paper aims to fill this gap by providing accessible guidance on how to use distributed computing solutions for economic research. With datasets in the billions of observations (e.g. Cavallo and Rigobon (2016) and Gao et al. (2019)) and peta-bytes of data (e.g. Ng (2017)) and growing, economists need to be able to handle and analyse those in a practical and efficient manner. We provide such a framework by showing how to run your existing data handling pipeline on a distributed computing solution. More specifically we illustrate the parallelization of key tasks frequently encountered in economic research and policy anaylsis: (i) computing summary statistics, (ii) estimating micro econometric models, (iii) panel data models and (iv) time series models using the *Apache Spark* framework. The paper specifically targets a non-expert audience and is therefore useful for researchers in economics and social sciences more general who struggle with datasets larger than their in house resources allow them to handle.

[1]For this paper we consider a data set as *large* whenever analysing it on retail-grade computers is a challenge. This is typically the case whenever datasets exceeds the 64 GB memory limit of most machines. Ng (2017) analyse over 4 million gigabytes (=4 zetabytes) of price data in their work, describe the memory problem that researchers face, when dealing with this kind of data.

[2]Other field in economics, notably macroeconomics, provide guidance on how to numerically solve models. For example Druedahl (2019) describes how to solve non-convex consumption-saving models. Caraiani (2018) shows how to solve heterogeneous agents models in Julia. Guidance on handling large datasets for empirical research is limited. A notable exception is the textbook by Foster et al. (2016) who provide an excellent overview of multiple topics related to big data in social sciences, including web crawling, machine learning tools for classification and ethical consideration. In our paper, we focus exclusively on Spark as a tool to handle and analyse datasets available to researchers that were too large to handle on retail grade computers.

The basic idea behind Spark is that instead of *bringing the data to the computation* (i.e. read the data from your hard drive into your computers memory) you should *bring the computation to the data* (i.e. run several computations in parallel on the machines where the different parts of the dataset are stored). This allows to handle datasets which are much larger than your computers memory usually allows to handle. There are several providers for cloud computing solutions providing a rich ecosystem of tools for distributed data storage and processing including Spark. In this paper we use *Amazon Web Services* (AWS), but the logic described seamlessly extends to other platforms.

To illustrate the use of Spark for economists, we demonstrate four typical use cases. First, we handle and pre-process a real-world dataset of US home mortgage applications with nearly 140 million observations using R's `sparklyr` library, which is built on the popular `dplyr` library providing an efficient and intuitive approach for data pre-processing. We then use this dataset to illustrate the estimation of various micro econometric models where we provide standard regression output tables and information on runtime performance conditional on cluster resource constraints. Next, we use Python's `pyspark` to fit a static fixed effects model via within-group data transformation using a simulated panel dataset with one billion observations. In this respect we show how to compute panel robust standard errors using a simple customized distribution scheme. Finally, we provide an example that entails forecasting a large number of time series in parallel.

Fernández-Villaverde and Valencia (2018) conclude that Python and R are inferior to higher level programming languages like C++ and Julia, when it comes to run-time performance based on their comparison of value function iteration. Our results indicate, that for empirical economic research Python and R by using Spark are well equipped for data handling and analysis of very large datasets. As Python and R are considerably easier to learn than e.g. C++ (and in fact today's working standard in data science), we view our introduction to Spark (based on Python and R) as a useful guide for economists who want to analyse datasets larger than their computer's memory allows. Our results in terms of runtime and ease of handling suggest Spark is a suitable tool for economic research. Using an *Elastic Map Reduce* (EMR) setup we are able to pre-process a 150 GB dataset in just under five minutes, whereas the standalone approach on our local machine crashes. Similarly, estimating micro econometric and panel regression models on our local machine would require computing crossproducts of very large arrays, which is not feasible on retail grade computers. Moreover, for the time series analysis case, the distribution scheme reduces total runtime performance by about 95% relative to a single-machine setting.

The contribution of the paper is two-fold. First, we demonstrate the usage of Spark for economic research and its superiority in fact for many applications involving large datasets. Secondly, the intuitive explanation of the framework along side the uses cases should enable economists without previous experience in parallel computing to work with Spark. This will allow them to (i) easily migrate their *existing* data handling and analysis to gain significant run time performance and (ii) allow them to handle datasets which were previously not manageable. To ease the process, we provide the codes[3] used in this paper and (in the appendix) carefully explain how to connect to a cloud service to run the codes. Moreover, we show how you can easily develop, test and debug your Spark programs (written in Python and R) on your local machine (avoiding paying fees for cloud services).

The remainder of the paper is organized as follows. The next section will briefly motivate the use of distributed computing solutions for empirical economic research. Section 3 will elaborate on the distributed computing architecture and provide a few numerical examples to illustrate the key idea of the Spark framework. Section 4 starts with a description on how to handle and preprocess a large real-world dataset followed by subsection 4.2 which uses this dataset to walk through the estimation of micro econometric models in Spark. Subsection 4.3 shows how to implement a fixed effects regression in Spark and how to obtain panel robust standard errors. The last subsection 4.4 shows a distributed set up for estimating time series models. The last section concludes.

2 Why Distributed Computing?

Parallel Computing has gained a lot attention and is today used across various fields in economic research, in particular for solving highly complex quantitative models. In this paper, we argue that *Distributed Computing* is the next step in the evolution of computational methods for economic research. The major advantage of parallel computing is that it can allow to solve quantitative problems, that were previously prohibitive expensive to evaluate. Fernández-Villaverde and Valencia (2018) provide an excellent overview of parallel computing performance of various programming languages and illustrate the runtime gains for solving a standard value

[3]Completely reproducible examples provided as RMarkdown and Jupyter notebook files are posted on our github repository: `https://github.com/benjaminbluhm/econometrics_at_scale`

function iteration problem.[4] They point out that (depending on your problem) you can speed up the analysis by a factor equal to the number of your computer's CPU cores. A standard retail grade computer at the time of writing typically comes with 4-8 cores, which means you can speed up the performance of your computations by up to 8 times, if you use parallel computing appropriately. As pointed out in Fernández-Villaverde and Valencia (2018), that very same logic holds true for the case of distributed computing with that exception that instead of being able to only use all your computer's CPU cores, you can use hundreds and even thousands of CPU cores of a cloud computing framework. Therefore, while parallel computing on your own machine certainly speeds up the process of many applications, it pales in comparison with the performance of a Spark cluster.[5]

Secondly, while your own computer comes at a (potentially high) fixed cost and is only used for a certain number of hours a day, the access to a cloud computing instance can be turned on and off at the flick of a switch. From a societal point of view, this will save resources: Instead of individuals owning powerful machines they only use so many hours a day, one can buy runtime on centralized high performance cloud computers.[6]

Thirdly, and most importantly for the scope of this paper, distributed computing allows to tackle data handling and analysis which were previously prohibitively expensive to run. For empiricists, this is usually the case when the dataset under consideration is considerably larger than your computers memory, making any analysis on it painfully slow or when the number of models which need to be evaluated becomes so large that even parallelization on a powerful retail-grade computer does not alleviate overall runtime concerns. Moreover, the researchers training with data handling and analysis shapes and limits the kind of research questions she conceives of in the first place. By lowering the threshold to use distributed computing solutions for economic research, we hope to achieve two goals. Firstly, we enable social scientists to approach their existing (big) data handling more efficiently and tackle *existing* questions that were previously prohibitively expensive to run. Secondly, by demonstrating the ease of use of cloud computing technologies, we hope to inspire *new* questions which leverage the possibilities of ever growing and ever more accessible datasets in the future.

[4]See S. Boragan Aruoba and Fernández-Villaverde (2015) for a comparison of different programming languages with regards to non-parallel computing performance.

[5]Sagade et al. (2019) fit vector error correction models for a large number of stocks and days. Even though the model was implemented in Python and C, the authors communicated to us that run-time was a major issue.

[6]Gray (2008) discusses pricing implications for cloud computing.

3 Distributed Computing Architecture

3.1 General overview and cluster architecture

In this section, we describe the core architecture of a distributed computing system based on Spark. The system is not limited to solving large-scale data handling and econometric tasks as described in this guide, but can be applied to many other expensive computing workloads that can be broken up into subsets of independent tasks. To only mention a few use cases from an economist's perspective, the distributed system in this guide could be adopted to distribute tasks such as for example value function iteration (S. Boragan Aruoba et al. 2003), extreme bounds analysis (Leamer 1985; Sala-I-Martin 1997), forecast combination (Clemen 1989; Timmermann 2006) or hyperparameter search.[7]

The choice of the distributed computing architecture presented in this section is guided by the following goals:

- Facilitate distributed computations on datasets which do not fit into a single machine's memory
- Highly scalable to large clusters of machines
- Minimal effort for setting up a cluster and pre-installation of user-defined libraries
- Ease of use to handle and analyse data in a distributed fashion using your existing Python and R data analysis pipelines[8]

A simple diagram of the distributed computing architecture is illustrated Figure 1. There are four layers, providing different capabilities and functionalities to the cluster. The distributed storage layer is based on the Hadoop API and uses Amazon S3 as a distributed, scalable file system, where input and output files from the application are stored on multiple machines, each storing a subset of all files. Hadoop scales to hundreds or even thousands of machines and therefore supports applications that run on very large datasets. A key idea of Hadoop (Ghemawat et al.

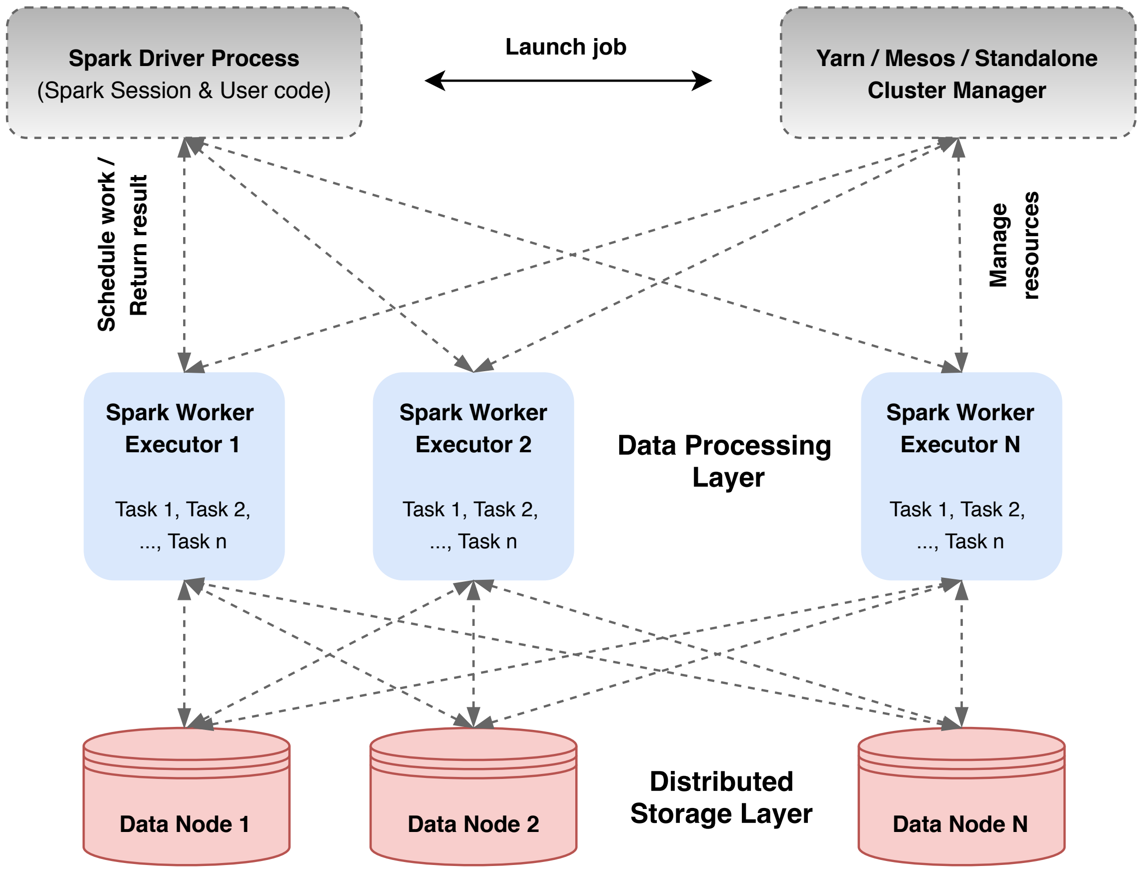
[7]Further details on using Spark for distributed hyperparameter search can be found on the Databricks website: https://docs.databricks.com/applications/machine-learning/automl/hyperopt/hyperopt-spark-mlflow-integration.html

[8]Other statistical packages like Stata unfortunately do not offer any Spark interfaces. Matlab, another popular computing language, recently started to offer Spark/Hadoop support (see https://www.mathworks.com/products/compiler/hadoop-and-spark.html). Java and Scala also offer interfaces, but are less known among social scientists

2003; Dean and Ghemawat 2004) is to move the computation to the data (and not vice versa) in order to minimize network congestion which yields large benefits in terms of computational efficiency for huge datasets of gigabytes to terabytes in size.

Figure 1: Spark's distributed computing architecture

This schema illustrates a distributed computing architecture. When the user submits a Spark application it launches the *Spark Driver* which is the process that takes care of breaking down the user program into individual tasks and coordinating each of these tasks on the *Spark Executors*. The *Spark Driver* submits a resource request to the *Cluster Manager* which launches the *Spark Executors* according to the requested resources. The *Spark Executors* perform the tasks received from the *Spark Driver*. The *Distributed Storage Layer* is based on the Hadoop API and holds the distributed dataset which is partitioned across harddrives of the Spark worker nodes. Distributed datasets can be used on any Hadoop supported storage system including for example Hadoop Distributed File System (HDFS), S3, Cassandra, Hive and HBase. Authors graph based on Karau, Konwinski, et al. (2015), Karau and Warren (2017) and Samadi et al. (2018).



The resource management layer uses YARN (Yet Another Resource Negotiator) and is in charge of managing cluster resources and scheduling data-processing jobs.[9]

[9]The other two existing Spark cluster managers are standalone mode and Mesos. While the latter is just another cluster manager for running Spark in a distributed environment, the former is typically used for running Spark on your local machine where the cluster size is determined by the machine's memory and number of cores.

Moreover, the cluster resource manager is responsible for administering YARN components and keeping the cluster in good health.

The data processing layer uses Spark, which was first introduced by Matei Zaharia et al. (2010) at UC Berkeley for large-scale machine learning use cases. In the meantime, Spark has turned into an open-source, distributed data processing platform for big data workloads relating to machine learning, stream processing and graph analytics.[10] The *Resilient Distributed Dataset* (RDD) defines the core component of Spark's distributed data processing engine. RDDs are collections of lazily evaluated, distributed data objects - also called partitions - which are stored in the data nodes connected to the Spark worker nodes and can be manipulated in a parallel fashion on the different executors of the system. Spark is based on a master/worker architecture where the driver communicates with the cluster manager as a single coordinator which is responsible for managing the workers in which executors run. The Spark driver is a process that hosts a Spark application and executors are processes that run computations and store data defined by your application code (for example, a `sparklyr` program for micro econometric analysis). A more elaborate description of the Spark architecture can be found, for example, in Chambers and M. Zaharia (2018).

3.2 The map–reduce framework

In this section we illustrate how Spark manipulates data in a parallel fashion using two simple examples. The first example takes as an input an RDD with key/value pairs to highlight a map-reduce algorithm that returns the mean value for each key. In the second example we briefly sketch how Spark is applied to approximate the maximum likelihood estimate of a generalized linear model.

Example 1 – A simple case for distributed computing: Compute the mean of a large dataset

Perhaps the simplest example to demonstrate the map-reduce framework is to compute the average value for each group for a large dataset.[11] Consider a dataset D = [('A', 5), ('B', 7), ('C', 3), ('D', 4), ('A', 8), ('B', 6), ('C', 2), ('D', 1), ('A', 9), ('B', 3)]. Figure 2 illustrates the map–reduce logic used in Spark. As an input it takes an RDD with 10 key/value tuples where the capital letters define the keys. The map function takes as an input the RDD with key/value pairs which is distributed across

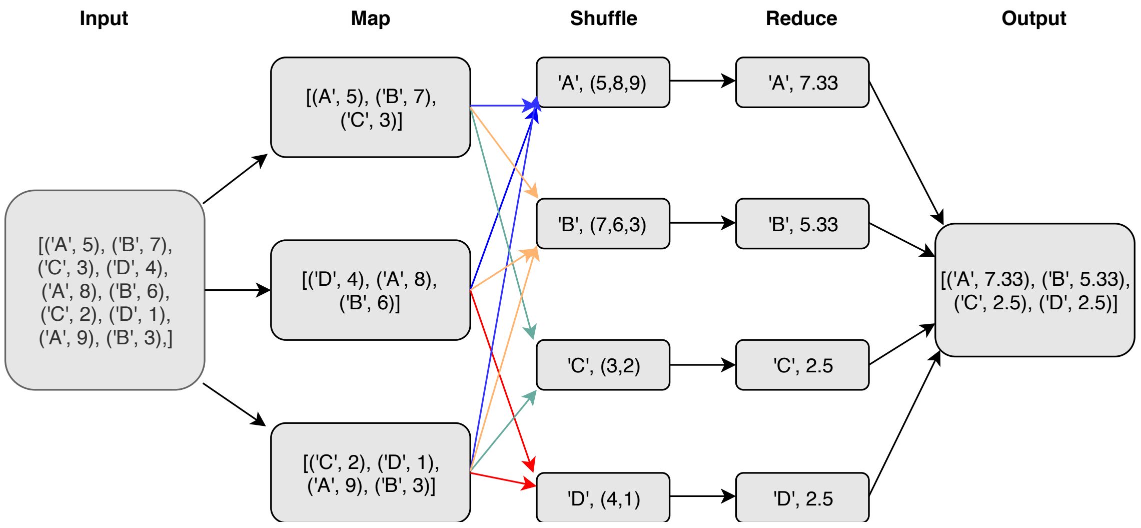
[10]For further details see: https://spark.apache.org/

[11]A similar example is provided in Fernández-Villaverde and Valencia (2018)

three partitions in this example. The reduce function is called once for each key, takes the input values to compute the average value and returns a key/value tupple. Note that the data is shuffled between the map and reduce stage to ensure that all values of a given key share the same node.[12] After the reduce step is completed, the data is transferred back to the masternode, where we can find the output [('A', 7.33), ('B', 5.33), ('C', 2.5), ('D', 2.5)], i.e. the averages for each group A, B, C and D. We can simultaneously count the number of observations per group such that in a second step one can compute a weighted average of the group averages, weighted with the number of observations in each group to obtain the overall average. While the computational overhead of mapping the data across nodes does not justify the efficiency gains for such a small dataset, it becomes increasingly powerful when the size of the input data grows, and in particular if the input data exceeds memory.

Figure 2: Distributed Computation of the mean

This illustrates a simple map–reduce logic for computing the average of a list of numbers. The data is first (randomly) *mapped* across workers. In a second step it is *shuffled* such that all values with a given key are allocated to the same worker. In the *reduce* step, the average is computed and finally returned back to the master node. On the master node, one can then compute the weighted average of the groups' averages.


Input
Map
Shuffle
Reduce
Output
[('A', 5), ('B', 7), ('C', 3), ('D', 4), ('A', 8), ('B', 6), ('C', 2), ('D', 1), ('A', 9), ('B', 3),]
[(A', 5), ('B', 7), ('C', 3)]
[('D', 4), ('A', 8), ('B', 6)]
[('C', 2), ('D', 1), ('A', 9), ('B', 3)]
'A', (5,8,9)
'B', (7,6,3)
'C', (3,2)
'D', (4,1)
'A', 7.33
'B', 5.33
'C', 2.5
'D', 2.5
[('A', 7.33), ('B', 5.33), ('C', 2.5), ('D', 2.5)]

[12]Since this shuffle process can be computationally expensive it may be more efficient to implement a hash partitioner to ensure that all values associated with a particular key are grouped together in the same partition. An example of this is provided in subsection 4.3.

Example 2 – A not so simple case for distributed computing: Compute median of a large dataset

The previous subsection outlined how to compute the average value for each group of a dataset, which is one of the most common examples to illustrate the map–reduce framework. Does the framework easily extend to all frequently used characteristics of data? Consider a dataset $D = [3, 4, 2, 6, 7, 1, 2, 4, 5, 6, 4, 4, 5, 6, 4]$. If you sorted that dataset by size it would look like $D_{sorted} = [1, 2, 2, 3, 4, 4, 4, 4, 4, 5, 5, 6, 6, 6, 7]$ and its median would be equal to 4. If D is too large to be fit into memory, (how) can we use Spark to compute the median value? A simple application of the map–reduce framework is not possible for this case, since there is no way to allocate the data across workers in a helpful way. There is however a large literature on dealing with these kind of computational problems.[13] For practical purposes Greenwald et al. (2001) propose an algorithm implemented in Spark that strikes a good balance between accuracy and runtime to compute percentiles on large datasets. The main take away however remains: When computation cannot be broken down across nodes, the trivial map–reduce framework fails to deliver a simple solution and more elaborate algorithms are required.

Example 3 – Back to econometrics: Distributed Ordinary Least Squares

Linear regression is arguably one of the most popular statistical model used in economics. Spark uses a distributed version of *stochastic gradient descent* (SGD), an optimization technique which is widely used in existing machine learning libraries (Apache 2020). While a thorough treatment of the method is beyond the scope of this paper[14], we shall outline the basic ideas to provide an intuitive understanding to the reader. Stochastic gradient descent is essentially a stochastic approximation of the well-known gradient descent optimization. To run SGD in a distributed fashion, Spark computes SGD for sub-samples on each worker and averages the estimated parameters, a procedure similar in spirit to Zinkevich et al. (2010). A graphical representation of the procedure (with three workers) is provided in Figure 3. A similar procedure is feasiable for generalized least squares models (allowing probit and logit specifications). At the time of writing, Spark's MLlib contains a useful set of estimation commands for our purposes[15], which is likely to grow in the future.

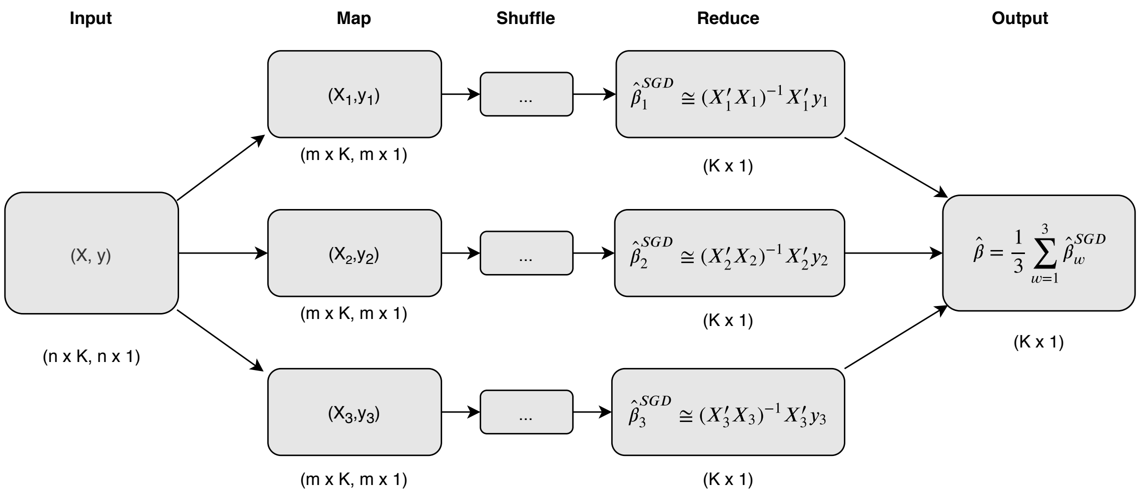
[13]The interested reader is referred to Munro and Paterson (1980) as a starting point on selection and sorting problems under limited storage.

[14]The interested reader is referred to Bonaccorso (2018).

[15]Bottou (2010) provides a comparison of run-time performance of different algorithms for a big data setting

Figure 3: Distributed Linear Regression Algorithm

In this example, we illustrate distributed OLS. This example uses three Spark executors $w = 1, 2, 3$. The input data (X, y) has n observations of K variables and a dependent variable y. In the *map* step, random sub-samples of the data (m observations each) are distributed across the executors. Each executor w computes a stochastic-gradient-descent solution $\hat{\beta}_w^{SGD}$ which stochastically approximates the (true) OLS solution $(X_w' X_w)^{-1} X_w' y_w'$ on the sub-sample of data allocated to the executor w. In a final step, the estimates are averaged to obtain $\hat{\beta}$. For details see Zinkevich et al. (2010).



Input
Map
Shuffle
Reduce
Output
(X, y)
(n x K, n x 1)
(X_1, y_1)
(X_2, y_2)
(X_3, y_3)
(m x K, m x 1)
...
$\hat{\beta}_1^{SGD} \cong (X_1' X_1)^{-1} X_1' y_1$
$\hat{\beta}_2^{SGD} \cong (X_2' X_2)^{-1} X_2' y_2$
$\hat{\beta}_3^{SGD} \cong (X_3' X_3)^{-1} X_3' y_3$
(K x 1)
$\hat{\beta} = \frac{1}{3} \sum_{w=1}^{3} \hat{\beta}_w^{SGD}$

4 Distributed Econometrics

In this section we discuss how to use well-known econometric techniques in a distributed setting. The first subsection shows how to obtain summary statistics of a big data set. Subsection 4.2 and 4.3 show how to run micro-econometric and panel-regression models in spark on data which would be too expensive to evaluate in a non-distributed fashion. Finally subsection 4.4 demonstrates how to train time series models at scale. In all subsections, we provide information about computing time and cluster resources which may serve as a reference for other researchers confronted with similar big data applications.

4.1 Summarising a large dataset

In this subsection, we discuss how to use spark to explore and understand datasets that are too large to fit in memory (referred to as *big data* from here on).[16] With ever more data being both collected and connected, the ability to handle such data

[16]The code to reproduce the results in this section are stored on our github repository: https://github.com/benjaminbluhm/econometrics_at_scale/tree/master/chapter_4.1

amounts is of crucial importance. Even at the time of writing, many existing datasets used in economic research qualify as big data. For example, Edwards et al. (2007), Dick-Nielsen et al. (2012), and Jankowitsch et al. (2014) use the TRACE dataset on corporate bond trades which as of today consists of more than 230 million observations and 32.9 GB. Many non-public datasets are even larger. For example, the European Central Banks collects daily snapshots of derivative exposures of financial intermediaries in the Euro area, which resulted in the need of a specific IT infrastructure (Boneva et al. 2019). With the trend of administrative data for research (Einav and Levin 2014), being able to handle such amounts of data will be crucial both for academic research and policy work (Irving-Fisher-Committee 2020).

Dataset

For this paper we focus on the well-known HMDA dataset, which contains loan application data from the US, frequently used in economic research (see for example Munnell et al. (1996), Duchin and Sosyura (2014), and Gilje et al. (2016)), which is also introduced in Foster et al. (2016). The data can be downloaded in yearly files from the *Federal Financial Institutions Examination Council* (FFIEC) website.[17] The entire data set spans from 2007–2017 and contains 150GB+ of data, which one can reduce to 29GB by replacing character labels with numerical identifiers (for example using the FIPS numeric code "01" instead of spelling out "Alabama"). The dataset contains applications for loan mortgages along several characteristics of borrowers (income, county, etc. . .) and lenders (bank name, balance sheet info, etc. . .). We provide a copy of the dataset[18] and the subset we are using[19] on our dropbox. Please note that we do not own or maintain this dataset. Check the FFIEC's website for the most recent version.

Spark Setup

To analyse the dataset, we follow a two-part strategy. We locally develop a Spark application on a randomly selected sub-sample of 200,000 observations. We test and debug our spark program on a retail grade computer and after finding satisfactory performance run the same code in a distributed fashion on AWS. While we restrict ourselves to the main steps of the distributed computing logic here, the complete `sparklyr` code is available including a fully reproducible example on our github

[17] `https://www.consumerfinance.gov/data-research/hmda/explore`
[18] `https://www.dropbox.com/sh/y5vrc3fnhwvw14o/AAAkgKja5YVpTT2vSUM0dW6-a?dl=0`
[19] `https://www.dropbox.com/s/z690uga5a0qrezv/HMDA_subsample.csv?dl=0`

repository. To use `sparklyr` for handling and analysing large datasets, we found the following tasks useful:

- Upload the data to S3
- Do all heavy computations in `sparklyr` using `dplyr` syntax for dataframe manipulation and `spark_apply()` to use base R functions
- Use `collect()` to get the results back to base R and continue to plot or print tables

Results

With 29.4 GB in size, we were not able to load the entire data set into R or Python on our local machine (which featured 16 GB of memory). Therefore, running the entire analysis in base R was not feasible.[20] Instead we imported a subset of 200,000 randomly selected observations and developed a spark application which conveniently summarizes the dataset. We find a combination of `spark_apply()` and `dplyr` functions very helpful to handle the dataset. For example, we need to combine the two-digit state fips code with the three-digit county fips code to create a unique county identifier. To do so we need to pad all single-digit state codes with a leading zero (i.e. changing “1” to “01”). There is no `dplyr` function to do so, so we use `spark_apply` to pass a base R function to create the “state_code_fips” variable:

```
hmda_spark = hmda_spark %>%
  spark_apply(function(e)
    data.frame(sprintf("%02d",as.numeric(e$state_code)), e),
    names = c('state_code_fips', colnames(hmda_spark)))
```

We can use `dplyr` backend to analyse large datasets, which provides very readable code. Consider for example the following six lines of code which generate the raw data used for Figure 4. It first groups the data by year and county and computes the average loan to income ratio for country-year. Subsequently we use `collect()` to read the Spark data back onto driver and further manipulate it:

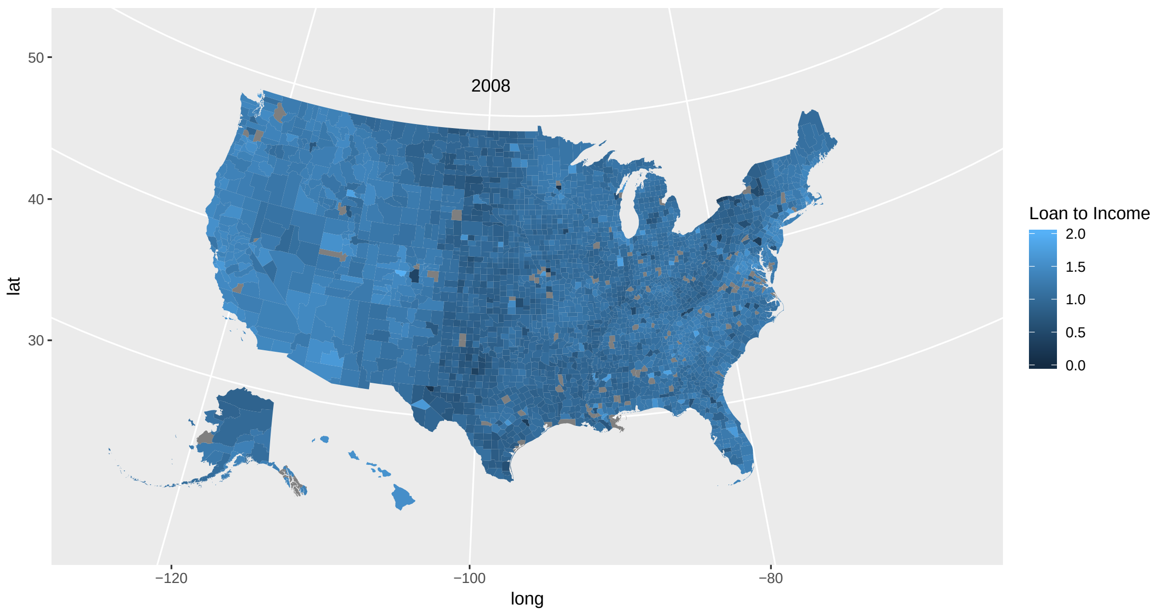
```
hmda_group = hmda_spark %>%
  group_by(as_of_year,county_fips_code) %>%
  summarise(avg_prc_to_inc = mean(loan_to_inc , na.rm=TRUE)) %>%
  collect() %>%   # read back to memory after heavy lifting is done by Spark
  mutate(log_loan_to_inc  = log(1+loan_to_inc) %>%
  select(county_fips, log_loan_to_inc, as_of_year)
```

A visual illustration is provided in Figure 4.

[20] We could have broken the computation down manually into several steps, however we found the Spark application much easier to implement, especially for researchers with a limited background in data handling, since the conceptually difficult steps are handled by spark autonomously.

Figure 4: Heat map US loan-to-income ratios

This graph plots the log of the loan-to-income of home mortgage application ratio across US counties for 2008.



4.2 Micro Econometrics on Spark

In this section, we explore how to run some popular micro econometric models using Spark. We use the same dataset as in the previous section to estimate linear regression, probit and logit models. We estimate those on a subset of the dataset locally, using base R functions and Spark functions to demonstrate their equivalence. Finally, we run Spark functions on AWS on the entire dataset. The code to reproduce the results in this section are stored on our github repository.[21]

Spark Setup

Using `sparklyr`, we access the Spark's machine learning library MLlib, which contains many statistical models used for economic research as well. As outlined in the previous section, we can clean the data using `dplyr` syntax in `sparklyr` to create a spark dataframe against which we can run regressions. `sparklyr` allows us to use base R formulas to be evaluted in their `ml_*` function family. For example, using the 137,819,151 observations strong `hmda_spark` dataframe created in the previous section, we can run a probit regression using the following code:

[21] https://github.com/benjaminbluhm/econometrics_at_scale/tree/master/chapter_4.2

```
glm_model = hmda_spark %>%
  ml_generalized_linear_regression(application_accepted ~  applicant_income_000s ,
                                   family = "binomial",
                                   link = "probit")
```

`sparklyr` automatically distributes the computation across the cluster, increasing the speed of computation or making it feasible in the first place. We run different versions of the following baseline regression

$$\begin{aligned} LoanGranted_i = \beta_0 \cdot Income_i + \beta_1 \cdot Male_i + \beta_2 \cdot White_i \\ + \beta_3 \cdot Black_i + \mathbb{1}_{Year(i)} + \mathbb{1}_{LoanPurpose(i)} + \varepsilon_i, \quad (1) \end{aligned}$$

where Income is the applicants income in \$1000, Male is a dummy variable which is equal to one if the applicant is male (and zero otherwise), White, Black are dummy variables which are equal to one if the applicant is white or black respectively (with Hispanics being the omitted category), $\mathbb{1}_{LoanPurpose(i)}$ and $\mathbb{1}_{Year(i)}$ are loan purpose and year fixed effects respectively. LoanGranted is a dummy variable which is equal to one if the loan application was successful. We estimate equation (1) using OLS, probit and logit regressions.

Results

To evaluate the performance of Spark, we run 9 models and report the results in Table 1. We run a linear regression, a probit and a logit model, each one locally on a sub-sample using base R functions and the Spark algorithm as well as the Spark algorithm executed on AWS on the entire dataset. While the empirical estimation confirm some well-known facts (such as white and male privilege in the loan market), the interesting result with regards to this paper's research question is the performance of the `sparklyr` regression commands. For example consider the linear regression estimated in column (1) – (3). Column (1) and (2) estimate a linear regression on the same dataset. Column (1) was estimated using native R's linear regression model, while column (2) used Spark's OLS via MLlib. Both report identical results, as expected. Runtime is considerably larger for the Spark solution. On a small dataset, the computational overhead used for the distribution scheme in Spark outweighs the speed benefits gained by distributed computing. Column (3) provides the same regression on the entire dataset, ran on AWS. Columns (4) – (6) and (7) – (9) repeat this exercise for *probit* and *logit regression* and yield similar conclusions.

A concern with big data as input for regression models is that with enough

Table 1: Microeconometric regression in Spark

To estimate a linear regression, specification (1) uses native R's `lm()` on the subsample, specification (2) uses `sparklyr`'s `ml_linear_regression()` on the subsample while specification (3) uses `sparklyr`'s `ml_linear_regression()` on the entire dataset. To estimate a probit regression, specification (4) uses native R's `lm()` on the subsample, specification (5) uses `sparklyr`'s `ml_generlized_linear_regression()` on the subsample while specification (6) uses `sparklyr`'s `ml_generlized_linear_regression()` on the entire dataset. To estimate a logit regression, specification (7) uses native R's `lm()` on the subsample, specification (8) uses `sparklyr`'s `ml_generlized_linear_regression()` on the subsample while specification (9) uses `sparklyr`'s `ml_generlized_linear_regression()` on the entire dataset. Runtime in local spark depends on the machine it is ran on. The results are based on an AWS EC2 instance type *m5.xlarge* (master + 4 nodes). Runtime is measured in minutes. ***, **, *, indicate statistical significance at the 1%, 5%, and 10% respectively.

	OLS			**Probit**			**Logit**		
	Base R (local)	Spark (local)	Spark (AWS)	Base R (local)	Spark (local)	Spark (AWS)	Base R (local)	Spark (local)	Spark (AWS)
Loan granted	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
Income (000$)	0.0001***	0.0001***	0.0000061944***	0.0003***	0.0003***	0.0002***	0.0006***	0.0006***	0.0002***
	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)
Male	0.0250***	0.0250***	0.0298***	0.0639***	0.0639***	0.070***	0.0982***	0.0982***	0.07006***
	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)
Race									
White	0.01383***	0.01383***	0.00829***	0.0354***	0.0354***	0.0264***	0.0613***	0.0613***	0.02645***
	(0.0113)	(0.0113)	(0.0000)	(0.0109)	(0.0109)	(0.0000)	(0.0061)	(0.0061)	(0.0000)
Black	-0.1486***	-0.1486***	-0.1450***	-0.3823***	-0.3823***	-0.3629***	-0.6060***	-0.6060***	-0.3629***
	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)
# Observations	147,329	147,329	137,819,151	147,329	147,329	137,819,151	147,329	147,329	137,819,151
Year FE	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Loan Purpose FE	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Runtime (min)	**0.71**	**0.007**	**9.90**	**0.146**	**1.556**	**33.69**	**0.026**	**0.995**	**19.833**

observations, any correlation will be statistically significant at conventional levels (usually 1%). This is true, since the variance of the estimator shrinks towards zero, as the number of observations grows to infinity. However, this does *not* imply that regressions on large datasets will incorrectly find effects, it merely means that the effect size can be arbitrary small and still statistically significant. When discussing these issues with other researchers, we often encountered suggestions like "With big data, the size of the effect is more important". We are sceptical of that interpretation: The size of an effect matters for small and big data, but the larger your dataset, the more likely it is that you are able to pick up even very small effect sizes (Flom 2013). In fact, you pick up effect sizes which one would simply discard as statistically "insignificant" on small datasets. For a more thorough treatment of large sample theory, the reader is referred to Ferguson (2017).

4.3 Panel Econometrics on Spark

This section illustrates how to estimate a panel data regression in a distributed fashion using Spark.[22] We simulate a big data scenario by generating an artificial panel dataset that is about 90GB in size and contains one billion observations. The key contributions of this section are as follows: First, we illustrate a simple Spark SQL logic to implement within-group data transformation in order to fit a one-way fixed effects estimator. While we restrict the example in this paper to a single fixed-effect, the approach can be easily generalized to a larger number of fixed effects which are encountered in many real-world datasets. Second, we provide empirical results on the validity of the estimated model coefficients and we give an indication of runtime performance and required computing resources. Third, we show how to compute panel-robust standard errors allowing for heteroscedasticity and serial correlation. These standard errors are currently not available in Spark MLlib, however, we show that robust standard errors can be computed in a distributed fashion using our customized distribution scheme.

As for the other subsections, we provide the relevant source code in the format of a Jupyter notebook available on our github repository.[23] Additionally, the data used for this section is provided on our Dropbox.[24]

[22]For a comprehensive treatment of panel data models we refer the interested reader to Baltagi (2008).

[23]https://github.com/benjaminbluhm/econometrics_at_scale/tree/master/chapter_4.3

[24]https://www.dropbox.com/sh/vk2ra1ufupi0yky/AABHUX6FZxIOWdk9LMnNTy5ea?dl=0

Dataset

To generate our large artificial dataset we simulate data according to the following linear panel regression model (for a general introduction to panel data and fixed effects see e.g. Wooldridge (2010)):

$$y_{it} = X'_{it}\beta + \epsilon_{it} \quad \forall \quad i = 1, ..., N \quad t = 1, ..., T \tag{2}$$

where subscript i defines the panel index and refers for example to an individual, subscript t denotes the time period, y_{it} is the dependent variable, X_{it} is the matrix of regressors, β denotes the vector of parameters and ϵ_{it} the error term. Furthermore, we assume that the data contains an unobserved individual-specific fixed effect that is constant over time. In particular, the error term ϵ_{it} is decomposed into an idiosyncratic component u_{it} and an individual-specific effect α_i that is constant over time:

$$\epsilon_{it} = \alpha_i + u_{it} \tag{3}$$

In addition, the individual-specific fixed effect is assumed to enter one of the regressors:

$$X_{1,it} = \alpha_i + \gamma_{1,it} \tag{4}$$

For simplicity all random variables above are drawn from a standard normal distribution. We simulate a dataset with $T = 10$, $N = 100,000,000$ and $K = 8$ (the number of regressors including the intercept). Furthermore, we impose a coefficient of 0.5 on all regressors without loss of generalization.

Note that the data generating process described by equations (2), (3), (4) introduces correlation between the regressors and the error implying that estimation of β via OLS yields inconsistent results. In the next section, we illustrate how to address this problem in Spark by implementing the well known within-group data transformation scheme which allows for consistent estimation of β via OLS for panels with short T and large N.[25] While the within-group estimator is implemented in various standard econometric software packages (see for example `reghdfe` command in Stata (Correia 2016), `plm` and `lfe` packages in R (Millo 2017; Gaure 2019) or `linearmodels` in Python (Sheppard 2019)), Spark does not yet provide any out-of-the box functionalities for estimating panel data models.

[25]Note that Hansen (2007) showed that tests based on robust standard errors are consistent even for large T as long as $N \to \infty$.

Spark Setup

To apply the within-group data transformation scheme we rewrite (2) by following the definition in Cameron and Trivedi (2005):

$$y_{it} - \bar{y}_i = (X_{it} - \bar{X}_i)'\beta + (\epsilon_{it} - \bar{\epsilon}_i) \tag{5}$$

where $\bar{y}_i = 1/T_i \sum_{t=1}^{T_i} y_{it}$. By subtracting the mean across time for each individual we remove the unobserved fixed-effect such that β can be consistently estimated via OLS. Below, we provide a brief sketch of the Spark SQL logic that implements this simple data transformation scheme for a dataset with one right-hand side variable:[26]

```
# Load panel data into Spark dataframe
df = spark.read.parquet('./panel_data')

# Create dataframe with mean across time for each individual i
df.createOrReplaceTempView("df")
df_mean = spark.sql("SELECT i, AVG(y) AS y_bar, AVG(x1) AS x1_bar FROM df GROUP BY i")

# Apply within-group data transformation scheme via left join
df_mean.createOrReplaceTempView("df_mean")
df_within_group = spark.sql("
SELECT a.i, t, (a.y - b.y_bar) AS y_tilde, (a.x - b.x_bar) AS x_tilde
FROM df AS a LEFT JOIN df_mean AS b ON a.i = b.i")
```

Following the transformation defined by (5), we can implement the within estimator using Spark MLlib generalized linear regression class. We should notice though that the standard errors provided by MLlib are the default OLS standard errors which tend to be too low as they do not account for the loss in degrees of freedom arising from demeaning the data. To get a consistent and unbiased estimate for the standard errors we must inflate them by factor $([N(T-1)-K]^{-1}[NT-K])^{1/2}$ (see Cameron and Trivedi (2005) for further details).

Yet, researchers often prefer to compute a panel-robust estimate of the variance-covariance matrix which permits serial correlation in the error term ϵ_{it} and heteroskedasticity of arbitrary form. In practice, model errors are often correlated over time for a given individual which violates the assumption of independence in the model errors. This erroneous assumption leads to a downward bias in conventional standard errors as the benefit of additional time periods is overestimated. Moreover, the failure to control for heteroskedasticity induces additional bias in the standard errors. While at the time of writing this paper, panel-robust standard errors are not available in Spark MLlib or any other Spark package we are aware of, we can define our own simple distribution scheme to compute these standard errors. The distri-

[26]Note that the same logic could be easily implemented using Spark's Dataframe API rather than Spark SQL.

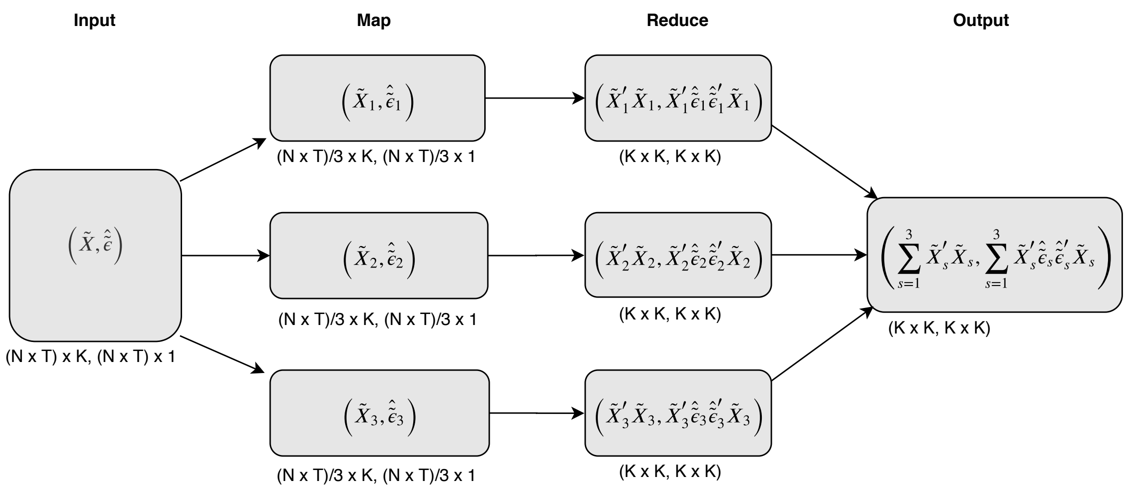
bution scheme can be easily derived from the standard definition of the estimator for the panel-robust asymptotic variance matrix (Arellano 1987):

$$\hat{V}[\hat{\beta}] = \left[\sum_{i=1}^{N} \tilde{X}_i'\tilde{X}_i\right]^{-1} \sum_{i=1}^{N} \tilde{X}_i'\hat{\tilde{\epsilon}}_i\hat{\tilde{\epsilon}}_i'\tilde{X}_i \left[\sum_{i=1}^{N} \tilde{X}_i'\tilde{X}_i\right]^{-1} \tag{6}$$

where $\tilde{X}_i = X_i - \bar{X}_i$ is a $T \times K$ matrix of the transformed regressors and $\hat{\tilde{\epsilon}}_i = \tilde{y}_i - \tilde{X}_i\hat{\beta}$ is a $T \times 1$ vector of residuals for panel index i from estimating (5) via OLS. Note that in our example $N = 100,000,000$ making the size of this dataset much too large to compute $\hat{V}[\hat{\beta}]$ on a standard computer. However, we exploit (6) to break the computation into small independent chunks, apply the computation on each of them separately and finally combine the resulting output to construct $\hat{V}[\hat{\beta}]$. Figure 5 provides a graphical representation of the distribution scheme.

Figure 5: Distribution Scheme for Panel Robust Variance Estimate

In this example, we illustrate a distribution scheme for panel robust variance estimates. This example uses three Spark executors $w = 1, 2, 3$. The input data $(\tilde{X}, \hat{\tilde{\varepsilon}})$ has $N \times T$ observations of K variables and the $N \times T \times 1$ vector of residuals of the regression $\hat{\tilde{\varepsilon}}$. In the *map* step, the data $(N \times T/3$ observations each) is distributed across exexutors. A hash partitioner is used to ensure that all data for a given panel index is sent to the same executor (for details refer to the code on our github repository). Each executor w computes $(\tilde{X}_w'\tilde{X}_w, \tilde{X}_w'\hat{\tilde{\varepsilon}}_w'\hat{\tilde{\varepsilon}}_w\tilde{X}_w')$ on the sub-sample of data allocated to the executor w. In a final step, the results are returned to the master node where the partial sums are summed up to serve as an input for (6).


Input
Map
Reduce
Output
$(\tilde{X}, \hat{\tilde{\epsilon}})$
(N x T) x K, (N x T) x 1
$(\tilde{X}_1, \hat{\tilde{\epsilon}}_1)$
(N x T)/3 x K, (N x T)/3 x 1
$(\tilde{X}_2, \hat{\tilde{\epsilon}}_2)$
(N x T)/3 x K, (N x T)/3 x 1
$(\tilde{X}_3, \hat{\tilde{\epsilon}}_3)$
(N x T)/3 x K, (N x T)/3 x 1
$(\tilde{X}_1'\tilde{X}_1, \tilde{X}_1'\hat{\tilde{\epsilon}}_1\hat{\tilde{\epsilon}}_1'\tilde{X}_1)$
(K x K, K x K)
$(\tilde{X}_2'\tilde{X}_2, \tilde{X}_2'\hat{\tilde{\epsilon}}_2\hat{\tilde{\epsilon}}_2'\tilde{X}_2)$
(K x K, K x K)
$(\tilde{X}_3'\tilde{X}_3, \tilde{X}_3'\hat{\tilde{\epsilon}}_3\hat{\tilde{\epsilon}}_3'\tilde{X}_3)$
(K x K, K x K)
$\left(\sum_{s=1}^{3} \tilde{X}_s'\tilde{X}_s, \sum_{s=1}^{3} \tilde{X}_s'\hat{\tilde{\epsilon}}_s\hat{\tilde{\epsilon}}_s'\tilde{X}_s\right)$
(K x K, K x K)

A paired RDD with equally sized data partitions is generated where the RDD's key is defined by the panel index and the value holds the data $(\tilde{X}_i, \hat{\tilde{\varepsilon}}_i)$ for that particular index. After mapping the RDD onto the executors a reducer function is called to perform the crossproduct computation of submatrices. The output is then collected back to the master node where $\hat{V}[\hat{\beta}]$ can be computed with little computational effort. Given a size of one billion observations and 8 regressors (incuding the intercept), we assign one million observations to a single partition which is reduced to two small two-dimensional arrays, each of dimension 8×8. As a result, 2,000 of these arrays are collected to the master node which are then used as an input to form (6).

The box below shows that it only requires a few lines of code to calculate the distributed version of the robust variance estimator. Note that in order to obtain correct results when distributing the computation across executors each RDD partition must hold all the data for a given panel index which is ensured by the hash partitioner function.

```python
# Select relevant columns for computing sandwich VCE
df = df.select(["id", "time", "u", "intercept", "x1", "x2", "x3", "x4", "x5", "x6", "x7"])

# Create hash partitioner assuring that data for each id is in one partition
def key_partitioner(id):
    return hash(id)

# Create key-value RDD with 1,000 partitions
key_value_rdd = df.rdd.map(lambda x: (x[0], x[1:10])).partitionBy(1000, key_partitioner)

# Compute array cross-products for sandwich VCE and collect results to master node
arr_bread_meat = key_value_rdd.mapPartitions(compute_bread_meat).collect()

# Construct bread and meat arrays and sandwich VCE
bread = np.linalg.inv(sum([item[0] for item in arr_bread_meat]))
meat = sum([item[1] for item in arr_bread_meat])
vcov = bread.dot(meat).dot(bread)
```

Results

In this section we show results for 3 different model specifications reported in Table 2 as OLS, Fixed Effects and Fixed Effects (Robust VCE). For each specification Table 2 shows estimation and runtime results across three separate estimation runs. The first two columns in each specification provide a comparison of results between Spark in local mode and R's plm package using a small subsample of data (100,000 rows). Essentially, this comparison serves to confirm the validity of estimated coefficients and standard errors obtained in Spark taking as a benchmark a popular panel data package from the R community. The estimation and runtime results for the complete dataset can be found in the last column of each specification.

Columns (1) - (3) show coefficient and standard error estimates for the OLS specification corresponding to a linear regression without prior data transformation to account for fixed effects. As expected the coefficient estimates on regressor X_1 show in all three regressions a significant deviation from its true value of 0.5. This bias is corrected for in the fixed effects case when estimation is performed on the transformed dataset as shown in columns (4) - (6) where standard errors have been adjusted for the loss in degrees of freedom. Columns (7) - (9) contain the estimation results with panel robust standard errors using our custom distribution scheme.

A comparison of runtimes across columns shows that the performance advantage of Spark comes into play for large volumes of data. For the cases that consider only a subsample of data the parallelization through Spark does not provide any performance improvement over local model fitting. Columns (3), (6) and (9) show the results when the entire dataset is used. Note that in order to provide a realistic indication of required cluster life time the reported runtime includes not only the time for the actual model fitting stage but also for the time it takes to perform the relevant data transformation steps. A comparison of columns (3) and (6) indicates that this data transformation can be computationally expensive as runtime is substantially higher for the fixed effects specification. The highest runtime is reported for column 9 which is not surprising given that on top of data transformation this specification involves both the computation of residuals and the variance covariance matrix in a distributed fashion. Finally, note that the standard errors in column (7) are the same as in column (8) which confirms that our custom distribution scheme yields valid results for the panel robust variance estimator.

Table 2: Panel Regression in Spark

To estimate a linear regression, specification (1) uses base R's `lm()` on the subsample, specification (2) uses `PySpark`'s `MLlib LinearRegressionModel()` on the subsample while specification (3) uses `PySpark`'s `MLlib LinearRegressionModel()` on the entire dataset. To estimate a static panel regression, specification (4) uses R's `plm()` package on the subsample, specification (5) uses `PySpark`'s `MLlib LinearRegressionModel()` on the within-group transformed subsample while specification (6) uses `PySpark`'s `MLlib LinearRegressionModel()` on the within-group transformed entire dataset. Standard errors under specification (4)-(6) have been adjusted for the loss in degrees of freedom induced by the within transformation. To estimate a panel regression with robust standard errors, specification (7) uses R's `plm()` package on the subsample, specification (8) uses our distributed variance covariance estimator on the subsample while specification (9) uses our distributed variance covariance estimator on the entire dataset. Runtime in local spark depends on the machine it is ran on. The results are based on an AWS EC2 instance type *m4.xlarge* (master + 10 nodes). Runtime is measured in minutes. ***, **, *, indicate statistical significance at the 1%, 5%, and 10% respectively.

	OLS			**Fixed Effects**			**Fixed Effects (Robust VCE)**		
	base R (local) (1)	Spark (local) (2)	Spark (AWS) (3)	R `plm()` (local) (4)	Spark (local) (5)	Spark (AWS) (6)	R `plm()` (local) (7)	Spark (local) (8)	Spark (AWS) (9)
X_1	0.9975***	0.9975***	0.9999***	0.4997***	0.4997***	0.5000***	0.4997***	0.4997***	0.5000***
	(0.0027)	(0.0027)	(0.0000)	(0.003339)	(0.003339)	(0.000)	(0.003342)	(0.003342)	(0.000)
X_2	0.4967***	0.4967***	0.5000***	0.4987***	0.4987***	0.5000***	0.4987***	0.4987***	0.5000***
	(0.0039)	(0.0039)	(0.0000)	(0.003337)	(0.003337)	(0.000)	(0.003361)	(0.003361)	(0.000)
X_3	0.4895***	0.4895***	0.5000***	0.4916***	0.4916***	0.5000***	0.4916***	0.4916***	0.5000***
	(0.0039)	(0.0039)	(0.0000)	(0.003338)	(0.003338)	(0.000)	(0.003306)	(0.003306)	(0.000)
X_4	0.5028***	0.5028***	0.5000***	0.5039***	0.5039***	0.5000***	0.5039***	0.5039***	0.5000***
	(0.0039)	(0.0039)	(0.0000)	(0.003339)	(0.003339)	(0.000)	(0.003350)	(0.003350)	(0.000)
X_5	0.4988***	0.4988***	0.5000***	0.5017***	0.5017***	0.5000***	0.5017***	0.5017***	0.5000***
	(0.0039)	(0.0039)	(0.0000)	(0.003334)	(0.003334)	(0.000)	(0.003316)	(0.003316)	(0.000)
X_6	0.4994***	0.4994***	0.5000***	0.5002***	0.5002***	0.5000***	0.5002***	0.5002***	0.5000***
	(0.0039)	(0.0039)	(0.0000)	(0.003352)	(0.003352)	(0.000)	(0.003354)	(0.003354)	(0.000)
X_7	0.5101***	0.5101***	0.5000***	0.5036***	0.5036***	0.5000***	0.5036***	0.5036***	0.5000***
	(0.0039)	(0.0039)	(0.0000)	(0.003334)	(0.003334)	(0.000)	(0.003316)	(0.003316)	(0.000)
# Observations	100,000	100,000	1,000,000,000	100,000	100,000	1,000,000,000	100,000	100,000	1,000,000,000
Runtime (min)	**0.019**	**0.017**	**8.33**	**0.183**	**0.083**	**36.09**	**0.386**	**0.367**	**83.52**

4.4 Time Series Econometrics on Spark

In this section we illustrate how to leverage Spark for large-scale time series analysis which plays a crucial role in the decision making process of many public and private institutions. Real-world forecasting systems in industries including manufacturing, retail, finance and energy nowadays have to process large forecasting workloads scaling to millions of time series.[27] Moreover, research in economics often requires fitting many time series models.[28] With each individual model typically containing only a limited number of data points, the setup is ideal for distributed computing since existing estimation methods can be executed in parallel across the worker nodes of the cluster framework.

An end-to-end machine learning system for probabilistic demand forecasting at *Amazon* built on Spark is described in Böse et al. (2017). The platform scales to large datasets containing millions of time series. The authors propose a simple distribution scheme for what they call a *local learning* approach, using Spark's *map()* operator to distribute model fitting and forecasting tasks across the cluster. Note that the distribution logic described in this section follows a very similar approach. A brief review of other distributed machine learning frameworks is given by Chun et al. (2016).[29] To replicate the results in this subsection we provide the input data[30] and a Juypter notebook which is available on our github repository[31].

Dataset

The dataset consists of 1,000 simulated time series with each draw of length 1,000. While many real-world time series datasets are considerably larger, the dataset is

[27]Consider for example a large retailer with several thousand stores and several thousand items per store. In order to forecast sales demand on store/item level granularity, the number of forecasts to be produced on a regular basis is in the order of several millions while the historical data to train the underlying models is even bigger. Similar examples could be made for other areas where time series analysis plays a key role. Ultimately, the computational challenge of such large-scale forecasting systems requires a high degree of parallelism to be able to produce models and forecasts in a reasonable amount of time.

[28]Many models which originated from the macro-econometric literature found their way in other fields. Sagade et al. (2019) use vector-error-correction models to study the market micro structure of stock exchanges. For this purpose, they have to evaluate a VECM for each stock and each day using tick-by-tick data, which is computationally expensive.

[29]Different distributed systems for time series forecasting have been proposed in the literature. For example, Stokely et al. (2011) introduce a computational infrastructure for large-scale statistical computing at *Google* using the MapReduce paradigm for R. Their technique is able to generate hundreds of thousands of forecasts in a matter of hours, using the `googleparallelism` package.

[30]`https://www.dropbox.com/home/Time_Series_SampleData`

[31]`https://github.com/benjaminbluhm/econometrics_at_scale/tree/master/chapter_4.4`

sufficiently large to demonstrate the performance gains from distributing the model fitting and forecasting process. Moreover, the limited size of the dataset facilitates easy reproducibility of the steps in this guide.

Time series are simulated from an *Autoregressive Moving Average Process (ARMA)* process, defined as follows (see, for example, Hamilton (1994)):

$$\left(1 - \sum_{i=1}^{2} \alpha_i L^i\right) X_t = \left(1 + \sum_{i=1}^{2} \theta_i L^i\right) \epsilon_t, \quad \epsilon_t \sim \mathcal{N}(\mu, \sigma^2) \tag{7}$$

where X is a real valued vector ordered by time index t, L is a lag operator, α_i and θ_i define the parameters on the autoregressive (AR) and moving average (MA) component, and ϵ_t is an independent, identically distributed disturbance term sampled from a normal distribution.[32]

The simulated time series data is written to a csv file with three columns. One column holds the time series data, a second column a unique identifier for each series and a third column a sequence of numbers specifying the order of the data for each series.[33] The last column is required because Spark may not preserve the temporal order of records when distributing the data across the cluster. To be able to fit a time series model after processing the data in Spark we therefore need to add this column in order to recover the temporal ordering of the data.

[32]Time series draws are generated with the *arima_sim()* method in `R`'s `stats` package (see Team (2016)). Following the example in the official package documentation, the orders of the AR and MA components are restricted to two and the AR and MA coefficients α_1, α_2, θ_1, θ_2 are set to 0.89, -0.49, -0.23, 0.25 respectively. The variance σ^2 of the disturbance term is set to 0.18. For further details see: `https://stat.ethz.ch/R-manual/R-devel/library/stats/html/arima.sim.html`

[33]In analogy to a timestamp or date in a real-world time series dataset.

Spark Setup

This section illustrates the key concept of our distributed forecasting system in Spark.[34] In short, the distribution of model fitting and forecasting is broken up into the following subtasks:

- Create custom Python module (we call it `fit_model_and_forecast.py`) with method that
  - reads time series data based on time series identifier contained in RDD partition
  - fits time series models, generates forecasts and saves fitted model
- Read dataset with time series data into Spark dataframe on master node
- Create *RDD* with distinct time series identifiers from Spark dataframe and partition *RDD* into collections of distinct identifiers
- Map *RDD* partitions of identifiers onto Spark executors
- On each Spark executor, import custom Python module and call method from module

To illustrate the simplicity of this approach we provide below the Python code that implements this parallelization logic in less than 10 lines of code:

```python
# Load time series data into Spark dataframe
df = spark.read.parquet('/path/to/time_series_data')

# Create RDD with dictinct identifiers and repartition dataframe into 100 chunks
time_series_ids = df.select('ID').distinct().repartition(100).rdd

# Add Python module to Spark context for parallel execution
spark.sparkContext.addPyFile('/path/to/python_module/fit_model_and_forecast.py')

# Function to import Python module on Spark executor for parallel forecasting
def import_module_on_spark_executor(time_series_ids):
        from fit_model_and_forecast import fit_model_and_forecast
        return fit_model_and_forecast(time_series_ids)
```

[34]Noteworthy, there are two libraries for distributed time series analysis in Spark. The `spark-ts` package provides functionalities for fitting time series models and manipulating large time series datasets. For further details see: https://github.com/sryza/spark-timeseries The package contains some frequently used univariate time series models, however, it is not under active development anymore and does not allow for parallel execution of algorithms not covered by the package (for example, multivariate time series models). The set of supported models is found at: https://github.com/sryza/spark-timeseries/tree/master/python/sparkts/models Another initiative is `Flint`, a library for highly optimized time series operations in Spark, which provides functionalities to efficiently compute across large panel and high frequency data. The github repository can be found at https://github.com/twosigma/flint. To the best of our knowledge, at the time of writing this guide Flint does not provide methods to fully parallelize all stages of the model fitting and forecasting process.

```
# Parallel model fitting and forecasting
time_series_ids.foreach(lambda x: import_module_on_spark_executor(x))
```

We first load the entire time series dataset into a Spark dataframe and create a partitioned RDD with distinct time series identifiers. In the context of the exercise in this paper, we set the number of RDD partitions to 100 using the *repartition()* function, cutting the collection of distinct identifiers into 100 subsets. Given that we have 1,000 time series in our dataset, the average number of identifiers in each partition is 10.[35] Since Spark will run one task for each partition, the number of RDD partitions in combination with the number of executors allocated for the application is an important parameter that determines the degree of parallelism.

In order to make our custom Python module available to all Spark executors, we need to add the module to the Spark context by calling the *addPyFile* method that takes as an argument the file path to the module. Next we define a function that we call below to (i) import the Python module on each Spark executor and (ii) to execute the Python module's method that takes as an input an RDD element and performs model fitting and forecasting tasks for the time series in question. Finally, we call this function for each RDD partition and its elements in a distributed fashion by calling Spark's *foreach* method.[36]

Results

Given a total sample size of 1,000 for each time series, we reserve the last 50 observations of the sample for forecast evaluation while the first 950 observations are used to fit an initial *ARMA(2,2)* model which is then used to produce the first forecast. Subsequently, we use a recursive estimation scheme, i.e. the size of the estimation sample for model fitting is extended by one observation as one makes forecasts for successive observations. As a result, a total of 50,000 estimations is performed across all time series in the dataset. The forecasts as well as the final model, fitted on the full sample for each time series, are stored in the S3 file system. For sake of simplicity, only one-step ahead forecasts are generated.

Table 1 shows the runtime for two different execution schemes. In the first scenario, the forecasting algorithm is executed on the master node in a non-distributed fashion and, thus, mirrors a single-core single-machine execution scheme. In this

[35]Note that Spark does not automatically distribute the number of elements evenly across partitions. Therefore, it is likely that some partitions contain more and others contain less than 10 elements. In order to maximize the gains from parallelization, we recommend to create partitions of balanced size via a custom function.

[36]Note that we use *foreach* rather than *map* since we do not collect any data back to the driver.

setting, models and forecasts are produced by iterating through all time series identifiers using a for loop. This scenario is used as a benchmark case to evaluate the performance gain from the distributed execution scheme.

The cluster hardware has been configured to 13 EC2 instances of type *m4.2xlarge*, comprising a total of 192 virtual CPUs and 384 GiB of RAM for the 12 worker nodes. The number of RDD partitions containing collections of distinct time series IDs is set to 100. Table 3 shows the runtime results for the two different scenarios.

Table 3: Runtime for different execution schemes

The results are based on an AWS EC2 instance type *m4.2xlarge*. Runtime is measured in minutes, Memory is measured in GiB, *Virtual CPUs* refers to the number of *virtual processing units* and *# Partitions* defines the number of RDD partitions, containing subsets of distinct time series IDs.

Scenario	Parallel	Virtual CPUs	Memory	# Partitions	Runtime
1	no	16	32	-	201.27
2	yes	192	384	100	6.20

The total runtime for the non-distributed scheme is about 200 minutes. This compares to roughly 6 minutes execution time for the distributed scheme, reducing runtime by about 95%. Clearly, the runtime of the distributed approach is strongly affected by the hardware configuration and the number of RDD partitions. An increase in the number of RDD partitions and a more powerful cluster with more CPUs and memory will likely lead to higher performance gains. While the impact of different hardware settings on the performance gain is beyond the scope of this paper, the results show that the distributed scheme can be used to complete large model fitting and forecasting workloads that would be intractable without substantial parallelization.

5 Conclusion

This paper presents a unified framework for handling large datasets for empirical research. It enables economists to handle and analyse ever growing datasets which are computationally difficult to evaluate on retail-grade computers using their existing data handling pipelines. With datasets becoming larger and larger, these computational constraints are more likely to be binding in the future. With data coming in ever higher frequency, dimensions and potential for being linked, being able to handle such data sources will likely result in novel empirical research designs. By lowering the threshold of employing cloud computing solutions to handle these kind of data sets, we aim to contribute to this process.

The cloud computing solution we elucidate, is built in Apache Spark and the distribution scheme is suitable for many established econometric methods as well as now popular machine learning models. After providing some background on distributed computing architectures, we demonstrate ease of use and (sizeable) computational gains. We do so by providing codes and configuration instructions for easily reproducible examples featuring a range of applications from micro-, panel- and time-series econometrics. In a first step, we demonstrate how Spark compares to a local execution of base R and Python codes. Intuitively, the computational overhang of mapping data cross the spark cluster is inefficient on small datasets. Yet the empirical results are identical and the Spark code comes at almost no additional complexity. We then take the operation to the cloud: Running the same codes we ran locally on a subset of data, we are able to handle and analyse datasets which would have been difficult to handle on retail grade computers. We provide an overview of popular statistical models which (i) are implemented in Spark as of now, (ii) can be estimated using simple modifications of existing commands and (iii) are difficult to run on Spark.

The presented approach requires minimal installation and configuration effort and it can be implemented with little background in computer science and parallel/distributed computing and without physical access to high performance computers. Additionally, the appendix of this paper contains extremely detailed instructions on how to lunch a computing cluster and provides minimal examples, which can easily be adapted to the readers needs.

References

Apache, Software Foundation (2020). *Apache Spark 2.4.4: Linear Methods – RDD-based API*. Online Documentation; accessed 01/02/2020. URL: https://spark.apache.org/docs/latest/mllib-linear-methods.html.

Arellano, Manuel (1987). "Practitioners' corner: Computing robust standard errors for within-groups estimators". In: *Oxford bulletin of Economics and Statistics* 49.4, pp. 431–434.

Aruoba, S. Borağan and Jesús Fernández-Villaverde (2015). "A comparison of programming languages in macroeconomics". In: *Journal of Economic Dynamics and Control* 58, pp. 265–273.

Aruoba, S. Boragan, Jesus Fernandez-Villaverde, and Juan F. Rubio-Ramirez (Nov. 2003). *Comparing Solution Methods for Dynamic Equilibrium Economies*. PIER Working Paper Archive 04-003. Penn Institute for Economic Research, Department of Economics, University of Pennsylvania. URL: https://ideas.repec.org/p/pen/papers/04-003.html.

Athey, Susan and Guido W. Imbens (2017). "The State of Applied Econometrics: Causality and Policy Evaluation". In: *Journal of Economic Perspectives* 31.2, pp. 3–32.

Baltagi, B. (2008). *Econometric Analysis of Panel Data*. John Wiley & Sons.

Bonaccorso, Giuseppe (2018). *Machine Learning Algorithms: Popular algorithms for data science and machine learning*. Packt Publishing Ltd.

Boneva, Lena et al. (2019). "Derivatives transactions data and their use in central bank analysis". In: *Economic Bulletin Articles* 6.

Böse, Joos-Hendrik et al. (2017). "Probabilistic demand forecasting at scale". In: 10, pp. 1694–1705.

Bottou, Léon (2010). "Large-scale machine learning with stochastic gradient descent". In: *Proceedings of COMPSTAT'2010*. Springer, pp. 177–186.

Cameron, A Colin and Pravin K Trivedi (2005). *Microeconometrics: methods and applications*. Cambridge university press.

Caraiani, Petre (2018). *Introduction to Quantitative Macroeconomics Using Julia: From Basic to State-of-the-Art Computational Techniques*. Academic Press.

Cavallo, Alberto and Roberto Rigobon (2016). "The Billion Prices Project: Using Online Prices for Measurement and Research". In: *Journal of Economic Perspectives* 30.2, pp. 151–78.

Chambers, B. and M. Zaharia (2018). *Spark - The Definitive Guide: Big Data Processing Made Simple*. O'Reilly Media, Incorporated.

Chun, Byung-Gon et al. (2016). “Dolphin: Runtime Optimization for Distributed Machine Learning”. In: *The ML Systems Workshop at ICML*.

Clemen, Robert T. (1989). “Combining forecasts: A review and annotated bibliography”. In: *International Journal of Forecasting* 5.4, pp. 559–583. URL: https://ideas.repec.org/a/eee/intfor/v5y1989i4p559-583.html.

Correia, Sergio (2016). *Linear Models with High-Dimensional Fixed Effects: An Efficient and Feasible Estimator*. Tech. rep. Working Paper.

Dean, Jeffrey and Sanjay Ghemawat (2004). “MapReduce: Simplified Data Processing on Large Clusters”. In: *OSDI’04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, pp. 137–150.

Dick-Nielsen, Jens, Peter Feldhütter, and David Lando (2012). “Corporate bond liquidity before and after the onset of the subprime crisis”. In: *Journal of Financial Economics* 103.3, pp. 471–492.

Druedahl, Jeppe (2019). “A Guide On Solving Non-Convex Consumption-Saving Models”. In:

Duchin, Ran and Denis Sosyura (2014). “Safer ratios, riskier portfolios: Banks response to government aid”. In: *Journal of Financial Economics* 113.1, pp. 1–28.

Edwards, Amy K, Lawrence E Harris, and Michael S Piwowar (2007). “Corporate bond market transaction costs and transparency”. In: *The Journal of Finance* 62.3, pp. 1421–1451.

Einav, Liran and Jonathan Levin (2014). “Economics in the age of big data”. In: *Science (New York, N.Y.)* 346.6210, p. 1243089.

Ferguson, Thomas S (2017). *A course in large sample theory*. Routledge.

Fernández-Villaverde, Jesús and David Zarruk Valencia (2018). *A Practical Guide to Parallelization in Economics*. Cambridge, MA: National Bureau of Economic Research.

Flom, Peter (2013). *Hypothesis Testing with Big Data*. Cross Validated. (version: 2013-08-13). URL: https://stats.stackexchange.com/q/67335.

Foster, Ian et al. (2016). *Big data and social science: A practical guide to methods and tools*. Chapman and Hall/CRC.

Galarnyk, Michael (2017). “Install Spark on Windows (PySpark)”. In: *Medium*. URL: https://medium.com/@GalarnykMichael/install-spark-on-windows-pyspark-4498a5d8d66c.

Gao, Haoyu, Hong Ru, and Xiaoguang Yang (2019). *What Do a Billion Observations Say About Distance and Relationship Lending? Working Paper*.

Gaure, Simen (2019). *lfe: Linear Group Fixed Effects*. URL: https://CRAN.R-project.org/package=lfe.

Gentzkow, Matthew, Bryan T. Kelly, and Matt Taddy (2019). "Text as Data". In: *Journal of Economic Literature (Forthcoming)*.

Ghemawat, Sanjay, Howard Gobioff, and Shun-Tak Leung (2003). "The Google File System". In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. Bolton Landing, NY, pp. 20–43.

Gilje, Erik P., Elena Loutskina, and Philip E Strahan (2016). "Exporting Liquidity: Branch Banking and Financial Integration". In: *The Journal of Finance* 71.3, pp. 1159–1184.

Gray, Jim (2008). "Distributed computing economics". In: *Queue* 6.3, pp. 63–68.

Greenwald Michael; Khanna, Sanjeev et al. (2001). "Space-efficient online computation of quantile summaries". In: *ACM SIGMOD Record* 30.2, pp. 58–66.

Grimmer, Justin and Brandon M. Stewart (2013). "Text as Data: The Promise and Pitfalls of Automatic Content Analysis Methods for Political Texts". In: *Political Analysis* 21.03, pp. 267–297.

Hamermesh, Daniel S. (2013). "Six Decades of Top Economics Publishing: Who and How?" In: *Journal of Economic Literature* 51.1, pp. 162–172.

Hamilton, James Douglas (1994). *Time series analysis*. Princeton, NJ: Princeton Univ. Press.

Hansen, Christian (Feb. 2007). "Asymptotic Properties of a Robust Variance Matrix Estimator for Panel Data When T Is Large". In: *Journal of Econometrics* 141, pp. 597–620.

Irving-Fisher-Committee (2020). "2019 IFC Annual Report". In: *Irving Fisher Committee on Central Bank Statistics*. (accessed 15/01/2020). URL: https://www.bis.org/ifc/publ/ifc_ar2019.pdf.

Jankowitsch, Rainer, Florian Nagler, and Marti G Subrahmanyam (2014). "The determinants of recovery rates in the US corporate bond market". In: *Journal of Financial Economics* 114.1, pp. 155–177.

Karau, Holden, Andy Konwinski, et al. (2015). *Learning Spark: Lightning-Fast Big Data Analytics*. 1st. O'Reilly Media, Inc.

Karau, Holden and Rachel Warren (2017). *High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark*. 1st. O'Reilly Media, Inc.

Kleinberg, Jon et al. (2015). "Prediction Policy Problems". In: *The American economic review* 105.5, pp. 491–495.

Leamer, Edward E. (1985). “Sensitivity Analyses Would Help”. In: *The American Economic Review* 75.3, pp. 308–313. URL: http://www.jstor.org/stable/1814801.

Millo, Giovanni (2017). “Robust Standard Error Estimators for Panel Models: A Unifying Approach”. In: *Journal of Statistical Software* 82.3, pp. 1–27.

Mullainathan, Sendhil and Jann Spiess (2017). “Machine Learning: An Applied Econometric Approach”. In: *Journal of Economic Perspectives* 31.2, pp. 87–106.

Munnell, Alicia H et al. (1996). “Mortgage lending in Boston: Interpreting HMDA data”. In: *The American Economic Review*, pp. 25–53.

Munro, J.I. and M.S. Paterson (1980). “Selection and sorting with limited storage”. In: *Theoretical Computer Science* 12.3, pp. 315–323. URL: http://www.sciencedirect.com/science/article/pii/0304397580900614.

Ng, Serena (2017). *Opportunities and challenges: Lessons from analyzing terabytes of scanner data.* Tech. rep. National Bureau of Economic Research.

Sagade, Satachit et al. (2019). “A Tale of Two Cities – Inter-Market Latency, Market Integration, and Market Quality”. In: *Safe Working Paper Series* 234.1, pp. 1–57.

Sala-I-Martin, Xavier X. (1997). “I Just Ran Two Million Regressions”. In: *The American Economic Review* 87.2, pp. 178–183. URL: http://www.jstor.org/stable/2950909.

Samadi, Yassir, Mostapha Zbakh, and Claude Tadonki (2018). “Performance comparison between Hadoop and Spark frameworks using HiBench benchmarks”. In: *Concurrency and Computation: Practice and Experience* 30.12. e4367 cpe.4367, e4367. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4367. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4367.

Sheppard, Kevin (2019). *linearmodels: Models for Panel Data.* URL: https://bashtage.github.io/linearmodels/doc/panel/models.html.

Stokely, Murray, Farzan Rohani, and Eric C. Tassone (2011). “Large-Scale Parallel Statistical Forecasting Computations in R”. In: *JSM Proceedings, Section on Physical and Engineering Sciences, American Statistical Association.*

Team, R. Core (2016). *R: A Language and Environment for Statistical Computing.* Vienna, Austria.

Timmermann, Allan (2006). “Forecast Combinations”. In: *Handbook of Economic Forecasting.* Ed. by G. Elliott, C. Granger, and A. Timmermann. Vol. 1. Handbook of Economic Forecasting. Elsevier. Chap. 4, pp. 135–196. URL: https://ideas.repec.org/h/eee/ecofch/1-04.html.

Varian, Hal R. (2014). "Big Data: New Tricks for Econometrics". In: *Journal of Economic Perspectives* 28.2, pp. 3–28.

Wooldridge, Jeffrey M (2010). *Econometric analysis of cross section and panel data.* MIT press.

Zaharia, Matei et al. (2010). *Spark: Cluster Computing with Working Sets.* Boston, MA: USENIX AssociationBoston, MA.

Zinkevich, Martin et al. (2010). "Parallelized stochastic gradient descent". In: *Advances in neural information processing systems*, pp. 2595–2603.

A Appendix

The appendix contains all codes and setup details necessary to replicate the results above. Moreover, we hope it is a useful guide for researchers who are not familiar so far with distributed computing solutions. Our set up is built on Amazon Web Services (AWS) and while it would easily translate to another provider[37], the instruction below reference solely to the AWS platform. There is a massive amount of documentation for AWS available online and many user written tutorials describe different applications. In what follows we condensed this information into the minimal steps necessary to get your own data handling and analysis pipeline running on the cloud framework.[38]

A word of caution

Please be aware you will be billed by AWS for running computing instances and trying to replicate our setup will require running AWS computing instances. Also, uploading data to AWS may results in cyber-security risks. Make sure that you have the relevant permissions for your data and jurisdiction. This guide comes with absolutely no warranty. You may find some instructions helpful but you use it at your own risk!

The appendix is structured as follows. Section A.1 describes how to run spark on your local machine. More specifically, section A.1.1 describes `pyspark` and section A.1.2 describes `sparklyr`. The next subsection introduces Amazon Web Servies (AWS) and presents how to initialize a minimal setup to reproduce the results of this paper. The last subsection A.3 demonstrates how to deploy `pyspark` and `sparklyr` in section A.3.1 and A.3.2 respectively.

[37]such as Cloudera, Microsoft Azure, Google Cloud Platform, etc ...

[38]We benefitted greatly both from Amazon's official documentation as well as various resources from third parties. Some of those explained certain steps better than we ever could, so we merely restate them here such that the readers of our paper have all information in one place. Such cases are clearly marked at the beginning of each section.

A.1 Running Spark on your local machine

In this section, we show how to run Spark in local mode from a Jupyter notebook[39] using `pyspark` and from RStudio using `sparklyr`.

A.1.1 PySpark on Jupyter Notebook

The first step in the installation process is to download the latest Spark release from the official website and to decompress the folder into the home directory of your machine and (of course) have a Python distribution installed.[40] As a prerequisite, make sure to install the latest version of the Java Development Kid 8.[41] Afterwards you need to add the home directory of your local Spark installation to the path variable in your system's environment:

- On **MacOS** you need to edit the `.bash_profile` file, which is stored in your home directory, and add the following statement:[42]

```
export PATH=$PATH:/Users/home/spark-2.4.0-bin-hadoop2.7
export SPARK_PATH=/Users/home/spark-2.4.0-bin-hadoop2.7
export PYSPARK_DRIVER_PYTHON="jupyter"
export PYSPARK_DRIVER_PYTHON_OPTS="notebook"
export PYSPARK_PYTHON=python3
alias spark_notebook='source activate your-env; $SPARK_PATH/bin/pyspark --master local[1]'
```

  To start a Spark session, simply execute the following lines in your Jupyter notebook:

```
## in jupyter type:
from pyspark.sql import SparkSession
spark_session = SparkSession.builder.appName('econometrics_at_scale').getOrCreate()
```

- On **Windows** the process is slightly more involved. Galarnyk (2017) provides an excellent explanation, which we restate here merely for sake of completeness.[43] Download and decompress Spark to a local directory (avoid blanks

[39]To keep this part as simple as possible we do not elaborate on setting up PySpark in other commonly used Python developer tools such as PyCharm IDE where the configuration steps are slightly more involved.

[40]There are several Python distributions available. We would recommend you get an Anaconda distribution (available for Windows, MacOS and most Linux versions) which is great for scientific computing, as it comes with many popular libraries. You can download the latest version here: https://www.anaconda.com/distribution/#download-section

[41]You can find it here: https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

[42]You can edit the file using a standard text editor such as for example *nano*. Open the command line interface and after changing into your home directory, type *nano .bash_profile* to edit the file (changes can be saved via *ctrl + x*). Subsequently, the changes need to be activated via command *source .bash_profile*.

[43]not claiming authorship of the material.

in the path), e.g. to `C:\opt\spark`. Next, download the *winutils.exe*[44] into the bin folder of your Spark directory (e.g. `C:\opt \spark\spark-2.4.0-bin-hadoop2.7\bin`) and add the relevant environmental variables by running the following commands from the command line interface (CLI)[45]:

```
setx SPARK_HOME C:\opt\spark\spark-2.4.0-bin-hadoop2.7
setx HADOOP_HOME C:\opt\spark\spark-2.4.0-bin-hadoop2.7
setx PATH "%PATH%;C:\opt\spark\spark-2.4.0-bin-hadoop2.7\bin"
```

Close the terminal window and reboot your computer. Restart the terminal and enter `pyspark` which will open a Jupyter notebook. In the Juypter nootebook enter the following codes to create a spark context and test whether you are on the right version:

```
## in Juypter type:
sc = SparkContext.getOrCreate()
sc.version
>> '2.4.0'
```

A.1.2 Sparklyr on RStudio

If you are using R, chances are you are using the RStudio editor, which is free of charge for personal and academic use at the time of writing this paper. The developers of RStudio provide a magnificent introduction to `sparklyr`, which can be found here.[46] In what follows we borrow greatly from those resources, yet focus a bit more on the "social scientist" perspective of data handling and analysis.

To install `sparklyr` simply run the following commands in your R console:

```
install.packages("sparklyr")
library(sparklyr)
spark_install(version = "2.1.0")
```

Once installed, load the library and create a connection:

```
library(sparklyr)

# define spark configuration
conf <- spark_config()
   conf$`sparklyr.cores.local` <- 2  # number of CPU cores spark can use
   conf$`sparklyr.shell.driver-memory` <- "8G" # memory size of shell driver
   conf$spark.memory.fraction <- 0.6  # fraction of total computer's memory available to spark

# establish a spark connection
sc <- spark_connect(master = "local",
                    config = conf )
```

The official documentation continues by exploring the well-known flights dataset and we suggest you follow it along. Here however, we would like to continue with an

[44]You can find it here: https://github.com/steveloughran/winutils/blob/master/hadoop-2.6.0/bin/winutils.exe?raw=true

[45]You can open the command line interface by pressing *ctrl + R* and type `cmd`, which will open the console.

[46]https://spark.rstudio.com/

example which might be a bit closer to what economists are used to. Suppose you have a very large dataset and would like to develop your Spark application in local mode (to avoid paying cloud services fees) but the dataset is too large to be read into memory. Since your local spark instance is limited by the physical memory of your computer, we will read in only a fraction of the entire dataset. As an example, we chose the popular HMDA data provided by the *Federal Financial Institutions Examination Council* (FFIEC), which contains loan application data for the US. To develop locally we created a subsample containing the first 10,000 observations from the yearly HMDA datasets. Since it is only a small subsample you can fit it into your computers memory and develop your Spark application locally:

```
# import small subset to R
df = read.csv("HMDA_subsample.csv")

# copy the R dataframe df to spark using copy_to()
df_spark = copy_to(sc, df)

# or load it directly to spark
hmda_spark= spark_read_csv(sc, name = "hmda_spark", header= TRUE, delimiter = ",",
                path= "HMDA_subsample.csv")
```

Now, let's compute the mean loan amount by year in R using `dplyr` and in Spark usings `sparklyr`:

```
meanbyyear = hmda %>%
 group_by(as_of_year) %>%
 summarise(mean_loan_amounts_000s =
             mean(loan_amount_000s))

meanbyyear
# A tibble: 10 x 2
   as_of_year mean_loan_amounts_000s
        <int>                  <dbl>
 1       2007                   176.
 2       2008                   201.
 3       2009                   201.
 4       2010                   219.
 5       2011                   224.
 6       2012                   225.
 7       2013                   230.
 8       2014                   242.
 9       2015                   246.
10       2016                   259.
```

```
meanbyyear = hmda_spark %>%
 group_by(as_of_year) %>%
 summarise(mean_loan_amounts_000s =
             mean(loan_amount_000s)) %>%
 collect()
meanbyyear
# A tibble: 10 x 2
   as_of_year mean_loan_amounts_000s
        <int>                  <dbl>
 1       2008                   201.
 2       2009                   201.
 3       2011                   224.
 4       2014                   242.
 5       2015                   246.
 6       2010                   219.
 7       2016                   259.
 8       2007                   176.
 9       2012                   225.
10       2013                   230.
```

A great feature of `sparklyr` is that you can use `dplyr` syntax for your Spark application. Note how both approaches yield the same result. Also, note that the Spark results are *not* ordered. This is because the `group_by()` command in `sparklyr` distributes the data across executors and collects them back after computing the mean, which does not necessarily preserve the order. This example illustrates the basic data handling pipeline in Spark using `sparklyr`:

1. Establish a Spark connection

2. Load the data into Spark
3. Using `dplyr` syntax, manipulate the Spark dataframe
4. Load the results back to the R environment using `collect()`

Finally, we can disconnect the local Spark interface:

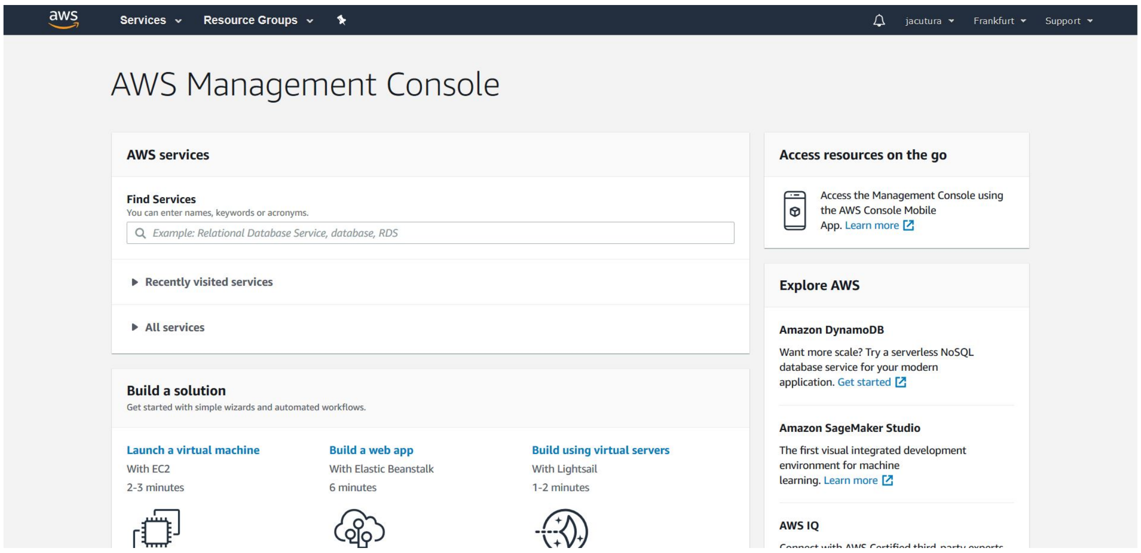
```
spark_disconnect(spark)
```

A.2 Setting up a cloud computing environment (on AWS)

A.2.1 Setting up AWS and upload data

In this section we walk through the process of creating an Amazon Web Services (AWS) account. Before we get into the details, note that if you are running computing instances on AWS you will be billed and a credit card is necessary for the setup. Also, note that uploading data to AWS and using computing instances may result in data vulnerabilities. This manual comes with no warranty whatsoever.

You can create your own account at https://aws.amazon.com/. Upon login you will see the AWS Management Console:



AWS Management Console

AWS services

Find Services
You can enter names, keywords or acronyms.
Example: Relational Database Service, database, RDS

Recently visited services

All services

Build a solution
Get started with simple wizards and automated workflows.

Launch a virtual machine
With EC2
2-3 minutes

Build a web app
With Elastic Beanstalk
6 minutes

Build using virtual servers
With Lightsail
1-2 minutes

Access resources on the go

Access the Management Console using the AWS Console Mobile App. Learn more

Explore AWS

Amazon DynamoDB
Want more scale? Try a serverless NoSQL database service for your modern application. Get started

Amazon SageMaker Studio
The first visual integrated development environment for machine learning. Learn more

AWS IQ

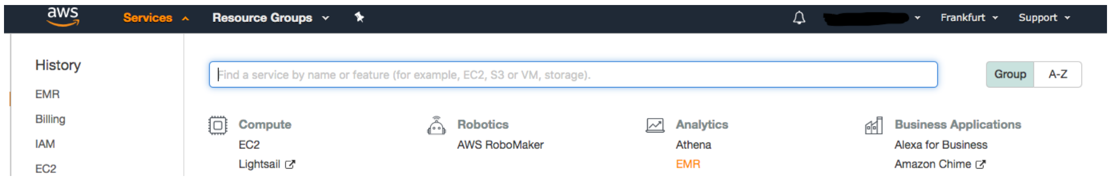
As a first step, we will upload data to AWS storage (called "S3"). To this end click *Services* and in the *Storage* section choose *S3*. First, we create a bucket for the project. To do so click *+ Create Bucket*. Choose any (DNS compliant) name. We named our bucket "econometricsatscalebucket" (Note that you cannot use the same name, since all buckets have to be unique on S3.). Under *Configure options* and *Set permissions* you may leave the standard settings.

In the bucket, we create several folders, where we store research data, bootstrap scripts and outputs. Specifically we create 3 folders: "data", "scripts", "output". In order to replicate our analysis in section 4.1 and 4.2 upload the HMDA[47] data to your S3 in a subfolder called *data/micro*. For the panel econometrics exercise in section 4.3, upload our simulated data[48] to a subfolder *data/panel*. For the time series exercise in section 4.4, upload our simulated time series data to a subfolder called *data/time_series*. Additionally, we need to create subfolders to store outputs from the exercise, namely parquet files (*output/time_series/parquet*), forecasts (*output/time_series/forecasts*) and fitted models (*output/time_series/models*). Also make sure to upload the python modul *fit_model_and_forecast.py* for computing forecasts and saving fitted models to a subfolder *scripts/time_series*. Instructions regarding the necessary bootstrap scripts for sparklyr will be provided at the respective subsections below.

A.2.2 Creating an *EMR* cluster and installing custom software

In this section we walk through the process of creating a Spark cluster using AWS *EMR* service. As emphasized earlier there are many other cloud vendors which provide similar services, so this section only contains one out of many other existing solutions that may be equally well or possibly even more suited for your application.

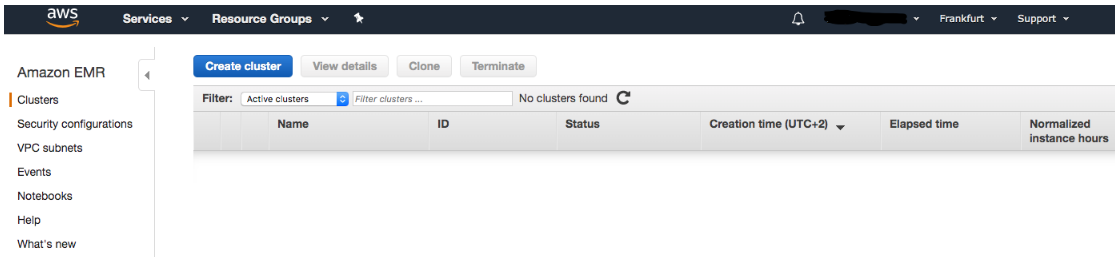
The firs step in creating a cluster is to go to the AWS *Services* menu and select *EMR*:



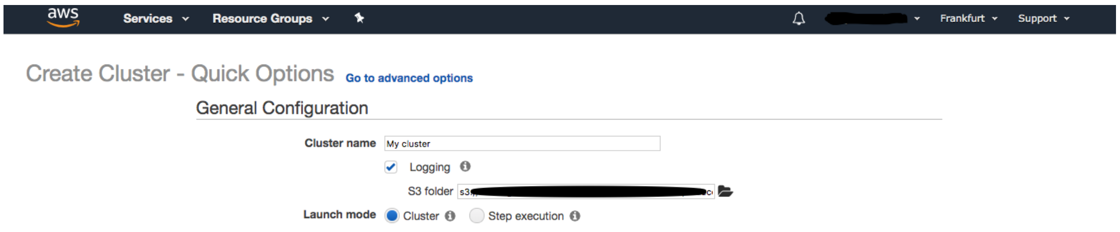
The *EMR* dashboard will open and provide you with the option to create a cluster as shown below:

[47] You can access the data from the FFIEC website. For convenience we provide a copy of the data on our dropbox. You can download it at `https://www.dropbox.com/sh/y5vrc3fnhwvw14o/AAAkgKja5YVpTT2vSUM0dW6-a?dl=0`. Note that we do not own or maintain this data. The latest version can be found at `https://www.ffiec.gov/hmda/`.

[48] You can simulate it yourself or access a copy we provide at: `https://www.dropbox.com/sh/vk2ra1ufupi0yky/AABHUX6FZxIOWdk9LMnNTy5ea?dl=0`

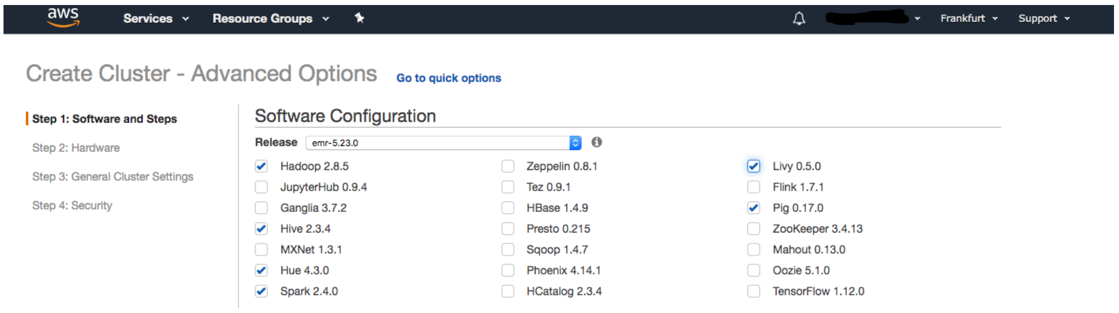


By clicking on the *"Create cluster'* button we enter the cluster configuration dashboard, noting that we have actually not yet created a cluster so we do not have to worry about any service charges at this stage. Next we need to configure our cluster to install Spark (at the time of writing this paper the latest version available in *EMR* is 2.4.0) and also Livy which is required for running a Jupyter notebook on the cluster. In order to configure the cluster software, we click on the *"Go to advanced options"* field:



Cluster name: My cluster
✓ Logging
S3 folder: [redacted]
Launch mode: ● Cluster ○ Step execution

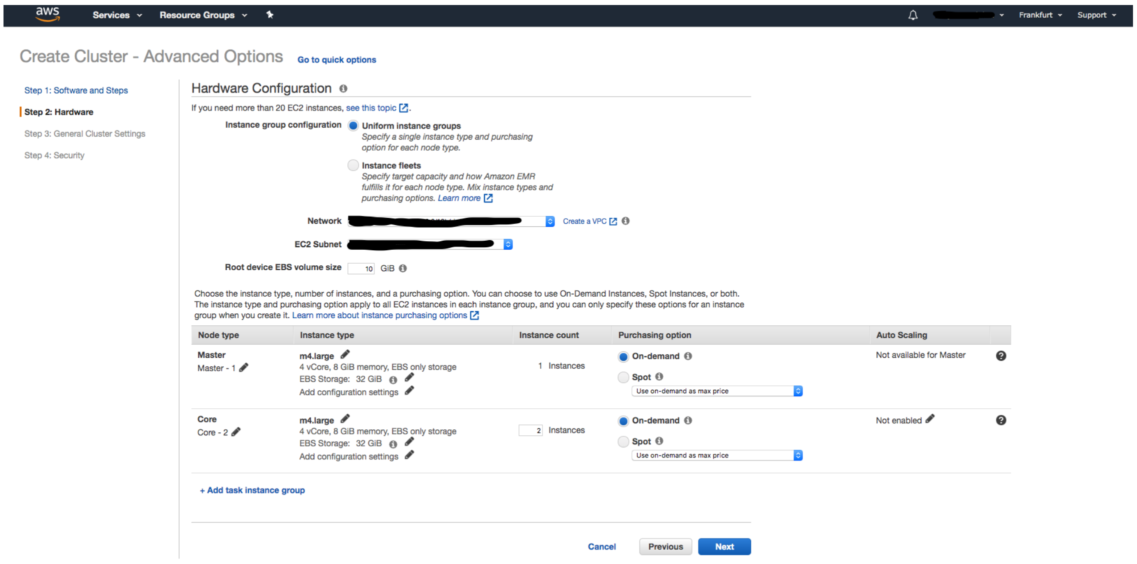
In *Release* choose the EMR version you want to run. To replicate the finding of this paper, choose "emr-5.23.0". In the *Software Configuration* section, check the Spark and Livy boxes and leave all other configurations at their default values:



Release: emr-5.23.0

✓ Hadoop 2.8.5	☐ Zeppelin 0.8.1	✓ Livy 0.5.0
☐ JupyterHub 0.9.4	☐ Tez 0.9.1	☐ Flink 1.7.1
☐ Ganglia 3.7.2	☐ HBase 1.4.9	✓ Pig 0.17.0
✓ Hive 2.3.4	☐ Presto 0.215	☐ ZooKeeper 3.4.13
☐ MXNet 1.3.1	☐ Sqoop 1.4.7	☐ Mahout 0.13.0
✓ Hue 4.3.0	☐ Phoenix 4.14.1	☐ Oozie 5.1.0
✓ Spark 2.4.0	☐ HCatalog 2.3.4	☐ TensorFlow 1.12.0

We proceed to configure the hardware settings of the cluster including the instance type and the number of instances. The hardware configuration strongly depends on the resource requirements (and budget considerations) of the specific application. We recommend to start with a small cluster with limited resources to familiarize yourself with the process of deploying and running your locally developed Jupyter notebook or RStudio script. For example, running one master and two core instances of type *m4.large* will give you enough flexibility to deploy and test your application on a small scale, while it will barely cost you more than a few dollars over a couple of hours:[49]


[redacted] Frankfurt Support

Create Cluster - Advanced Options Go to quick options

Step 1: Software and Steps
Step 2: Hardware
Step 3: General Cluster Settings
Step 4: Security

Hardware Configuration

If you need more than 20 EC2 instances, see this topic.

Instance group configuration [✓ **Uniform instance groups**] *Specify a single instance type and purchasing option for each node type.*
Instance fleets *Specify target capacity and how Amazon EMR fulfills it for each node type. Mix instance types and purchasing options. Learn more*

Network [redacted] Create a VPC

EC2 Subnet [redacted]

Root device EBS volume size 10 GiB

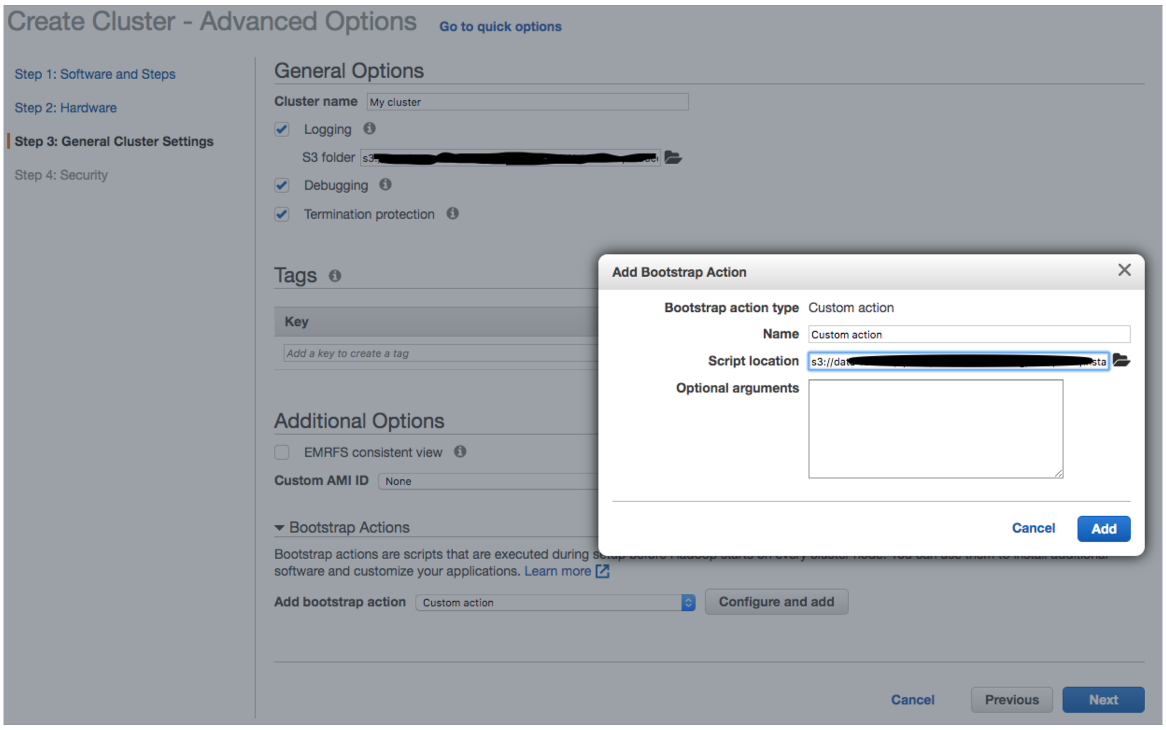
Choose the instance type, number of instances, and a purchasing option. You can choose to use On-Demand Instances, Spot Instances, or both. The instance type and purchasing option apply to all EC2 instances in each instance group, and you can only specify these options for an instance group when you create it. Learn more about instance purchasing options

Node type	Instance type	Instance count	Purchasing option	Auto Scaling
Master Master - 1	**m4.large** 4 vCore, 8 GiB memory, EBS only storage EBS Storage: 32 GiB Add configuration settings	1 Instances	[✓ **On-demand**] **Spot** Use on-demand as max price	Not available for Master
Core Core - 2	**m4.large** 4 vCore, 8 GiB memory, EBS only storage EBS Storage: 32 GiB Add configuration settings	2 Instances	[✓ **On-demand**] **Spot** Use on-demand as max price	Not enabled

+ Add task instance group

Cancel Previous Next

Once you have selected your preferred hardware configuration, go to the next section *General Cluster Settings*:


Create Cluster - Advanced Options Go to quick options

Step 1: Software and Steps
Step 2: Hardware
Step 3: General Cluster Settings
Step 4: Security

General Options

Cluster name My cluster
[✓] Logging
S3 folder [redacted]
[✓] Debugging
[✓] Termination protection

Tags

Key
Add a key to create a tag

Additional Options

EMRFS consistent view
Custom AMI ID None

Bootstrap Actions

Bootstrap actions are scripts that are executed during s
software and customize your applications. Learn more

Add bootstrap action Custom action Configure and add

Cancel Previous Next

Add Bootstrap Action

Bootstrap action type Custom action
Name Custom action
Script location [redacted]
Optional arguments

Cancel Add

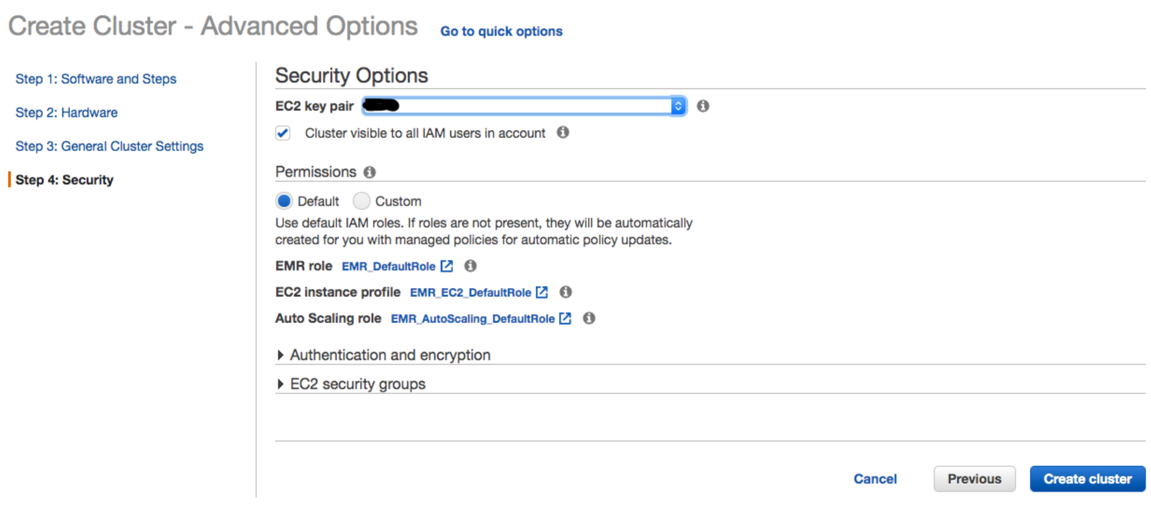
[49] A list of available instance types and prices can be found at: https://aws.amazon.com/de/ec2/pricing/on-demand/.

At the bottom of the page there is a *Bootstrap Actions* field where custom actions can be specified to install additional software or to customize the configuration of cluster instances. In essence, bootstrap actions are scripts that run on all nodes after the cluster is launched. For this purpose, a shell script specifying your custom installations must be uploaded to a folder in the S3 bucket. For example, if you want to install `scikit-learn` on all cluster nodes you would upload a shell script with the following content:

```
#!/bin/bash -xe
sudo pip install -U sklearn
```

Add the bootstrap action by selecting *"Custom action"* and, under the *"Configure and add"* button, browse the S3 path to the shell script (no optional arguments needed) and click *"Add"*.

After adding the custom bootstrap action, move on to the last step *Security* where you have to add your previously created EC2 key pair. Finally, you can create the cluster via the *"Create cluster"* button:


Create Cluster - Advanced Options   Go to quick options

Step 1: Software and Steps
Step 2: Hardware
Step 3: General Cluster Settings
Step 4: Security

Security Options

EC2 key pair [redacted]

✓ Cluster visible to all IAM users in account

Permissions

● Default ○ Custom

Use default IAM roles. If roles are not present, they will be automatically created for you with managed policies for automatic policy updates.

EMR role EMR_DefaultRole

EC2 instance profile EMR_EC2_DefaultRole

Auto Scaling role EMR_AutoScaling_DefaultRole

▸ Authentication and encryption

▸ EC2 security groups

A.3 Running Spark on a cluster

A.3.1 Running PySpark

This section describes the procedure of creating a Jupyter notebook for running PySpark jobs on an AWS EMR cluster. Note that the notebook will automatically be saved to your S3 bucket so after terminating the cluster you can still use the notebook when you start another cluster at a later stage.

To create a notebook, go to the EMR dashboard, select *Notebooks* and choose *Create notebook*. In the notebook configurations window, you are asked to provide a notebook name and under the *Cluster** option you can either choose to create a new cluster (via *Create cluster*) or alternatively you can attach to an existing running cluster. As mentioned above, you may want to start with a small cluster to familiarize yourself with the notebook workflow on EMR. For example, if you specify three instances, one instance is devoted to the master node and two instances are devoted to the worker nodes. You can also select an S3 location to store your notebook.[50] Finally, you can create the notebook by choosing *Create notebook*. A new view will open showing the configuration details of the notebook. Once the cluster is ready the notebook can be accessed via the *Open* button.[51]

In the notebook, a Spark session is automatically started which can be verified by following the example below:

```
data = [1, 2, 3, 4, 5]
sc = spark.sparkContext
dist_data = sc.parallelize(data)
dist_data.collect()
```

A.3.2 Running sparklyr

In this section we show how to deploy sparklyr on AWS EMR. The general steps are:

1. Configure and upload a bootstrap script
2. launch a cluster using the appropriate bootstrap script
3. Connect remotely to your RStudio on AWS

Configuring and uploading a bootstrap script

For the first step, we suggest to upload a bootstrap script which will install RStudio Server along with sparklyr. One such script is provided by Cosmin Catalin on his GitHub account.[52] For reference, we forked it to our repository and you can download the `install-rstudio-server.sh` bootstrap script here. You can open it using any editor and (if you wish) edit the following parameters:

```
USER="drwho"
PASS="tardis"
SPARK="2.1.1"
```

This allows to change the username and password for the web-login to the AWS RStudio Interface and the spark version running on it. Finally, upload the `install-rstudio-server.sh` to your S3 storage to any folder (for example into `econometricsatscale\bootstrap\sparklyr`).

[50] If you leave the default value a directory will automatically be created in your bucket.

[51] The user guide for setting up an AWS EMR notebook instance can be found at: https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-managed-notebooks.html

[52] https://gist.github.com/cosmincatalin/a2e2b63fcb6ca6e3aaac71717669ab7f0

Launch a cluster

When launching a cluster as described in A.2.2, choose the `install-rstudio-server.sh` as bootstrap script (and leave the additional parameters field blank). Create the cluster. While the cluster is launching you may edit the security group of your master node and enable TCP Port 8787 from Anywhere on your master node in order to allow a remote connect to RStudio Server. To do so, click Security ⇒ Create a security group. As a security group name you can choose anything, similarly for the Description. Upon creation, click "Inbound Rules" in the ribbon, and click "Edit rules" and then "Add Rule". For Type choose "Custom TCP Rule", and set the port range to 8787. As source, choose "Anywhere". Finish by clicking "Save rules".

Connect remotely (from a Windows machine)

As a first step, you will need to install Putty[53], an SSH client, which lets you run sparklyr on AWS from the comfort of your local machine's RStudio interface. Amazons keypair format is not supported by putty, which is why you need to convert it first. To do so, execute puttygen, click File ⇒ Conversions ⇒ Import Key and choose the AWS keypair (*".pem" file*) which you downloaded when creating it. Finish by choosing *Save Key*. Now, you can configure PuTTy. How to do so is documented on AWS[54] and merely restated here for reference:

1. Click Category List ⇒ Session and in the Host name field, type "`hadoop@MasterPublicDNS`", where "`MasterPublicDNS`" is your master's node address, for example: "`hadoopec2-###-##-##-###.compute-1.amazonaws.com`"

2. Click Category List ⇒ Session > SSH ⇒ Auth ⇒ Browse and select the `.pkk` file which you generated earlier.

3. Click Category List ⇒ Session > SSH ⇒ Tunnels and in the source port field, type 8157. Leave the Destination field blank and select the `Dynamic` and `Auto` options

4. Choose "Add" and "Open" and choose "Yes" to dismiss the PuTTy security alert.

The AWS console should open. In the AWS consolte type:

[53] You can find the latest version here: https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html

[54] https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-ssh-tunnel.html

```
sudo rstudio-server start
```

and press enter. This will launch RStudio on the AWS master node. Finally, to access RStudio on AWS open a browser and copy and paste:

`@ec2-###-##-##-###.compute-1.amazonaws.com:8787`

to your browser. You will be asked to enter the username and password created above and stored in your `install-rstudio-server.sh` file.

Connect remotely (from a Mac)

How to do so is documented on AWS[55] and merely restated here for reference:

1. Open a terminal window. On Max OS X, choose Applications ⇒ Utilities ⇒ Terminal. On other Linux distributions, terminal is typically found at Applications ⇒ Accessories ⇒ Terminal.

2. To establish a connection to the master node, type the following command. Replace "~\mykey.pem" with the location and filename of the private key file (.pem) used to launch the cluster:

```
ssh -i ~/mykey.pem hadoop@ec2-###-##-##-###.compute-1.amazonaws.com
```

The AWS console should open. In the AWS console type:

```
sudo rstudio-server start
```

and press enter. This will launch RStudio on the AWS master node. Finally, to access RStudio on AWS open a browser and copy and paste:

`@ec2-###-##-##-###.compute-1.amazonaws.com:8787`

to your browser. You will be asked to enter the username and password created above and stored in your `install-rstudio-server.sh` file.

[55] https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-ssh-tunnel.html

Recent Issues