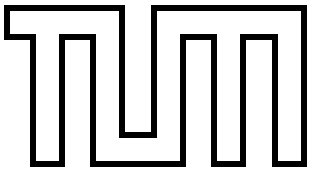


DEPARTMENT OF FINANCE

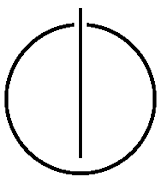
TECHNICAL UNIVERSITY OF MUNICH

Interdisciplinary Project in Finance and Informatics

Leads and Lags of Corporate Bonds and Stocks

Leads und Lags von Unternehmensanleihen und Aktien

Author:	Alex Kulikov
Supervisor:	Prof. Dr. Sebastian Müller
Advisor:	Zihan Gong
Date:	March 31, 2021



Abstract

In the scope of an Interdisciplinary Project at the Technical University of Munich, an international, extendable corporate bonds database had to be built up from scratch for future empirical research. For this purpose, an automatic extraction tool for financial securities data has been developed on top of the interface provided by the Refinitiv Datastream financial database. The required static and time series corporate bond data – consisting of over 60.000 bonds – has been downloaded, cleaned, and prepared for further research. Additionally, a matching algorithm has been developed to join corporate bond and stock data based on their issuing company. The algorithm utilizes fuzzy string matching based on company name as well as key-based matching based on the CUSIP-9 parameter, and has shown a matching ratio of around 45% without a globally unique company identifier. Further, the distribution of corporate bond returns has been found in-line with expectations, which served as a sanity check for the downloaded data. Finally, correlation and regression analyses have shown stock data to be more efficient in incorporating new information than corporate bond data based on a 1-month lead.

Contents

1. Introduction

In the scope of an Interdisciplinary Project (*IDP* in the following) at the Technical University of Munich, the lead and lag relationship between corporate bond and stock returns had to be analyzed. With the needed stock data already provided, the first step of the IDP was to develop a tool which would be able to automatically extract static and time series data from the Thomson Reuters Datastream financial database. In the second step of the IDP, the extracted bond data had to be cleaned and prepared for further analysis by various transformation techniques. Additionally, a matching approach to join the stock and bond datasets by issuing company needed to be developed. In the third and final step of the IDP, the resulting bond-stock database had to be checked for any sort of lead and/or lag relationships within bond and stock return pairs.

The extracted corporate bond data, both static and time series, has a wide array of applications, ranging from descriptive historical analysis to predictive models, and will thus be of significant value for future financial research. In order to make use of the most recent market trends and developments, the acquired bond database, as well as the existing stock information, should be extendable, such that recent data can be accumulated in a continuous manner. This is where the tool for automated data extraction from Refinitiv Datastream comes into play. It has to provide capabilities to download financial securities data in a convenient and seamless manner, as far as technology allows. Since corporate bonds and their qualities as an investment instrument are often compared to equities [7], it only makes sense to additionally develop an efficient approach to analyze the two asset classes "side-by-side". At this point, a suitable matching mechanism to join bond and equity data by their issuing company is of paramount importance, and can be used in many different analysis scenarios. At the end of any financial analysis, researchers are always interested in the insights into the functioning of financial markets and the optimal investment thesis which arises therefrom. Analyzing the lead and lag relationship of corporate stocks and bonds can provide such an investment thesis based on the efficiency with which the two asset classes incorporate new information in their pricing [11]. From a practical perspective, if I, as an investor, knew that today's stock returns predict bond returns in the next month, I could allocate my assets accordingly to gain a higher-than-usual profit from my investment. While the constraints and influence of market equilibrium conditions on such an investment strategy could be analyzed separately [7], the gain of such knowledge is undoubtedly of significant academical and practical importance.

2. Data Extraction

In order to draw any conclusions regarding the relationship of bond and stock returns, the respective static and time series data needs to be acquired first. Since equity data is already available from the beginning of the IDP, the bond data is the only one which has to be acquired. For bond data extraction, the financial database product Datastream, provided by Thomson Reuters, can be used, since it is licensed for usage by TUM students and employees.

2.1. Download Solution

As Thomson Reuters has a wide range of products which can be used for different types of data, the first thing that needed to be done, was to determine the most suitable product to download both static and time series data for corporate bonds. After some time spent reading up and gathering information on the Thomson Reuters product portfolio, it became apparent that some of the products, such as the TR Python API, are only suitable for equity data download, and not for corporate bonds, and only have a very limited number of parameters available for download. On the other hand, it was found that other Thomson Reuters products, which would normally be suitable for automated download of bond data, such as e.g. DataScope Select (DSS) or Thomson Reuters Tick History (TRTH), are not included in the existing academic license. Other products – noticeably the Datastream Web Service (DSWS) API, which is most suited for such requests – are generally not available for academic clients.

These findings were a significant setback for the bond data extraction, since the only option left to acquire large amounts of corporate bond data, was over the Datastream Add-In for Microsoft Excel. While this add-in is rather convenient for small-scale manual requests with the help of so-called request tables, it is not optimized for large data extraction queries. It does not provide an API for customizable requests. Instead, communication with the Datastream server is handled over a single API call available in VBA. This one and only callable function is implemented in C++, and can only be invoked in a black-box manner, since the provider does not give out its implementation. This leads to only one possible solution to automatically extract corporate bond data from Datastream. It can be described with the following steps:

1. Acquire Datastream codes / identifiers for all financial instruments which need to be downloaded.

2. Split these identifiers into batches small enough to be processed in a single Datastream request.

3. Fill a request table with as many requests as needed to include all the batches.

4. Launch the Datastream requests for all the batches one after the other.

5. Monitor the download process to ensure that the data is being consistently downloaded.

The programmatic development of the download tool will be based exactly on these five steps. The last step (monitoring the execution) is especially crucial and complex to implement. The reason for this is, as previously mentioned, that the Datastream Excel Add-In is not well-fit for large data downloads. Therefore, the following problems continuously arise during the download process:

- Datastream add-in eventually signs out for no obvious reason.
- Data download hangs, without any notification stating the reason or the hanging fact itself.
- Excel suspends the add-in and places it into a blacklist for repeated faulty behavior.

Since VBA is single threaded and cannot detect or react to erroneous behavior when the download is running, it is impossible to do the monitoring in the VBA/Excel environment. For this purpose, a Python wrapper has been developed as will be explained in section 2.6.

2.2. Bond Identifiers Acquisition

Since for both static and time series requests Datastream requires unique financial instrument codes, it is first necessary to obtain a list of identifiers for the securities for which the data needs to be downloaded. The most commonly used unique security identifier in Datastream is the so-called Datastream Code (short *dscd*). In the scope of the project, two different approaches have been developed for this task, and will be introduced in the following.

2.2.1. Programmatic Identifier Extraction

For the purpose of this work, we are interested in corporate bonds from all possible jurisdictions, and with any possible coupon and currency parameters. The only restriction is that we only concentrate on the issue date range between Dec 31, 1999 and June 30, 2020 (date of extraction).

Datastream allows to filter its financial security dataset by these parameters, e.g. when clicking on *Find Series* in a request table. After the securities have been filtered for the desired corporate bonds, these can be selected by repeatedly checking the box to select all bonds on the current page, and then clicking on *Next* to switch to next page. This is due to Datastream not providing an option to select all filtered securities at once if there are more than 4,000. Hence, if there are for instance 60,000 corporate bonds in the database in total, one would not be able to select them all at once. Instead, one would have to select the 15 bonds on the current page and then switch to next page $60,000/15 = 4,000$ times. This is of course very cumbersome for the user, and the repeated clicking sounds like a good process to be automated programmatically.

There are multiple tools and scripting languages which enable fast and easy click automation. Specifically for this project, I decided to go with Python 3 for this purpose, since it was already part of the environment. One of the packages which enable GUI automation in Python is *pyautogui*[1]. With build-in methods like $click()$ and $hotkey()$ it enables the user to simulate mouse clicks on the computer screen by giving the functions the screen coordinates of the buttons. Placing the commands into a loop in the right order makes it possible to simulate the entire process of selecting corporate bonds in Datastream. For a possible Python implementation see the file *datastream_pyautogui.py* in the project files. Note that the screen coordinates can significantly differ depending on the screen resolution and window settings.

While the described approach solves the problem of selecting all the needed corporate bonds from Datastream, there are two downsides to it. The first one is that it is cumbersome for the developer to determine and to enter the screen coordinates of all the buttons involved. The second is that, even when fully automated, the tool needs a lot of time to select and return all of the chosen securities if there are many of them. At this point, the second approach, even though it is manual, is both faster and easier to apply.

2.2.2. Manual Identifier Extraction

To extract the needed corporate bond identifiers manually, we can make use of the fact that Datastream allows us to select all filtered securities at once when there are less than 4,000. Because of this, we can simply split our entire data into multiple chunks that are all smaller than 4,000 bonds in total. This can be done by selecting one or more parameters (the number depends on the size of the dataset) according to which the bonds can be filtered even further. For example, an entire bond dataset with 60,000 bonds in total can be split by coupon size first, and then additionally by currency to produce bond batches of maximum 4,000 bonds each. For a visualization of this approach, see Fig. 2.1. If the split

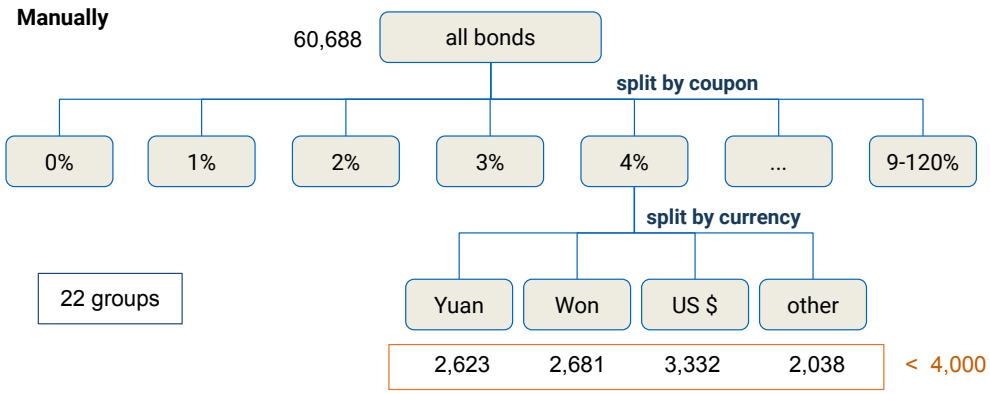


Figure 2.1.: Bond split by coupon and currency

has been done properly, it will include all of the desired securities, since each bond belongs to one particular coupon size as well as currency group. In most cases, there will be only

[1]https://pyautogui.readthedocs.io/en/latest/

few groups that need to be extracted from the interface. In the case of 60,000 bonds, one would only have $60,000/4,000 = 15$ groups in total. In reality, for this concrete use case, 22 different bond groups had to be created. This is because not all splits are perfect, and some of them just consist of 3,500 bonds instead of 4,000 for example. After the splitting work has been done, the resulting bond groups can be extracted manually with just a few clicks.

2.3. Automating the Request Table

After the bond identifiers in form of *dscd* codes have been extracted, they can be used to retrieve both static and time series data from Datastream. For this purpose, I wrote a VBA program which fully automates the download process. The only input needed from the user is the *dscd* identifiers, the desired variable codes (such as price, issued volume, etc.) as well as the desired time frames in the case of a time series request. Since the program code is rather complex, I will only cover the main approach briefly. A more in-depth description can be found in appendix A.4 to this work.

At the beginning, the VBA tool retrieves the user-provided identifiers and datatypes from the respective Excel files. Then, based on the input size, multiple calculations take place. For static information requests not much needs to be done, since these are usually relatively small and only depend on the number of securities for which the data is requested. For time series requests though, the tool estimates how large the entire request will get, depending on the dates window, the frequency of time points (e.g. daily or quarterly), the number of datatypes, and the number of identifiers. If the request is too large to be processed by Datastream in one run, it gets split into multiple smaller requests of equal size. Since there is no particular metric to estimate in advance whether a particular request will be executed by Datastream, or whether it is too large for that, the tool only computes an approximation based on an empirically measured *Bytes per Field* metric. The single requests then get entered into the request table one below the other, and each receive an own Excel file as destination to store the data.

As soon as the request table has been filled (which does not take long), the command to process the first request is issued to Datastream. This happens via a call to the single available function, which tells Datastream to process the current request table in a black-box manner. After the request ends, the tool checks whether the requested data has arrived to the destination file. If not, it checks the connection of the Datastream add-in and issues a warning to the user if the add-in unexpectedly disconnected. In both cases, the result of the request is logged, in order for the user to be able to read up on the proceedings later. To prevent the computer from sleeping or going in idle mode, the tool moves the computer mouse pointer after each request with a dedicated VBA function. When the first request of the request table has been processed, the other ones get executed in the same manner one after the other. Note that while it is possible to submit the execution for all requests at once, it is not advisable, since Datastream might issue an error due to the data being too large, or might otherwise simply hang during execution. This is exactly the reason why we had to split up the original request in multiple parts in the first place.

While the entire Datastream Extraction Tool is much more complex than what has been described here, the given explanation covers the most crucial parts of the download pro-

cess. At this point, note that the error monitoring step, which was previously mentioned as essential, cannot be completed in VBA due to its single-threaded execution engine. Section 2.6 will cover the required workaround for this functionality.

2.4. User Interface

In order to provide a graphical user interface as well as an error monitoring capacity (section 2.6) for the created VBA tool, a Python 3 wrapper program has been created. It's architecture can be seen in Fig. 2.2. At this point, note that a detailed usage manual for the Datastream Extraction Tool – including its user interface – can be found in appendix A to this work.

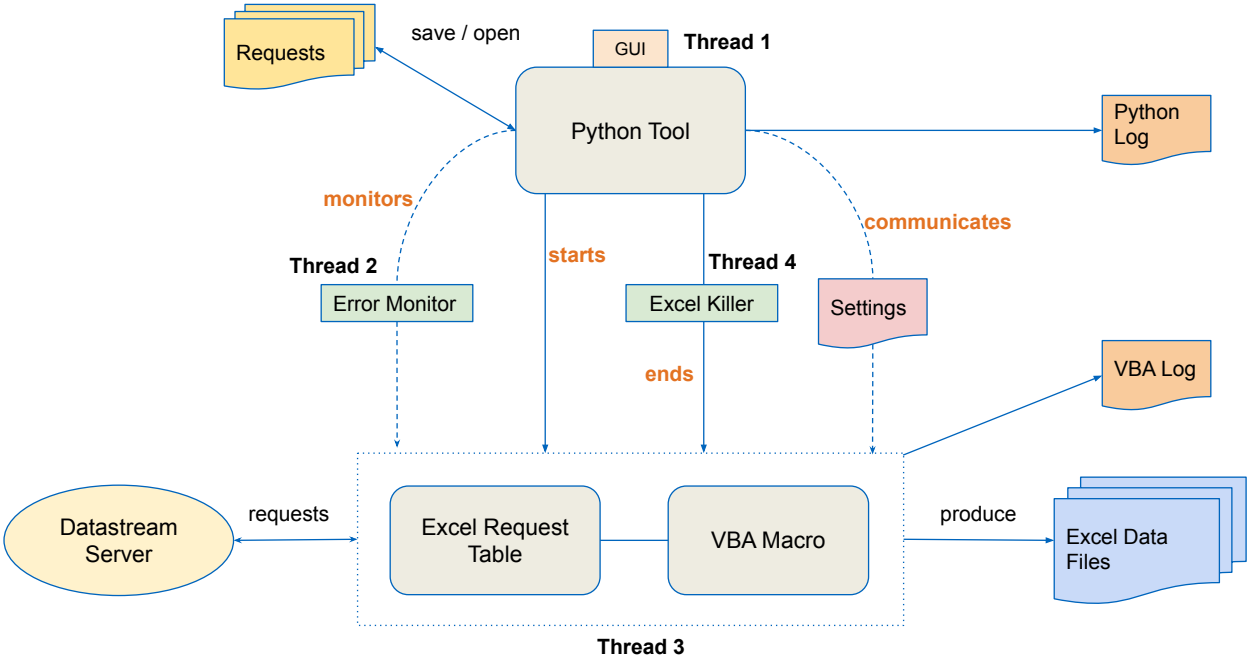


Figure 2.2.: Architecture of the Datastream Extraction Tool

As shown in the visualization, the Python program has a *GUI* component, which runs on its main thread (Thread 1). The layout of the user interface is depicted in Fig. 2.3.

Its features include:

- Request type selection (static or time series)
- Time frames and frequency for time series requests
- Request datatypes, which can be entered either in a manual list or via an Excel file
- Data identifiers, which can also be entered either manually or via Excel file

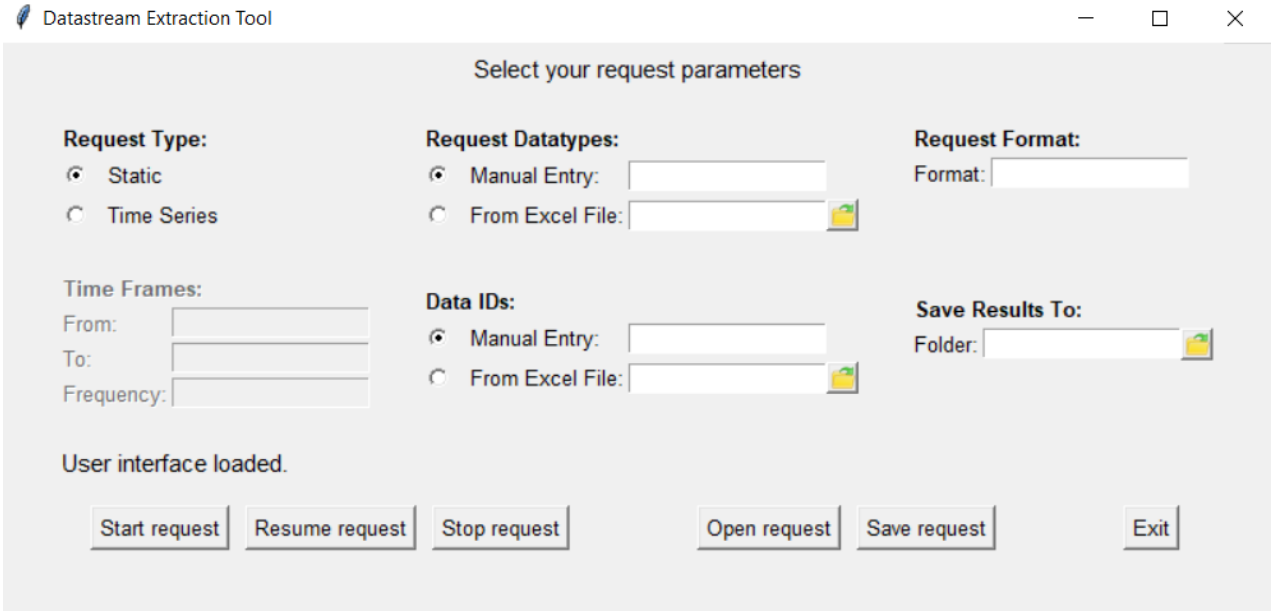


Figure 2.3.: Graphical User Interface of the Datastream Extraction Tool

- Request format with one or more Datastream request options (e.g. header row, currency, etc.)
- Destination choice for downloaded data
- Saving and reusing frequently entered requests

The user interface has been programmed using the *Tkinter*[2] package, which enables rudimentary container-based gui creation.

2.5. Other Functionality

Besides the user interface, the Python wrapper offers a logging facility for the actions taken, as well as a functionality to exchange settings and messages with the VBA component. For the latter purpose, a *settings.txt* file is provided which encompasses user-provided request details to forward the request to the VBA program. The entries are made in a key-value manner, which enables a fast and easy information exchange between the Python and the VBA parts. Besides request forwarding, the settings file is used by the Python tool to receive regular updates from the VBA on the current download status. This information is used for both status updates within the gui as well as for the error monitoring functionality which will be explained in section 2.6.

[2] `https://docs.python.org/3/library/tkinter.html`

2.6. Error Monitor

As shown in the visualization, the *Error Monitor* module runs in a separate thread (Thread 2), since Python allows execution on multiple threads simultaneously. It is started from the main thread, which controls the graphical user interface, whenever a new download request (Thread 3) has been issued to Datastream. The module maintains an internal counter which signifies the waiting time for the current request in Datastream to process. The counter gets increased each time the Python tool notices that the currently processed request in VBA has not yet changed to the next one. VBA, in its turn, keeps posting updates on the number of the request that is currently being downloaded to the settings file. The ping frequency of the error monitor can be adjusted to the user needs, and is currently set to 1 minute cycle time.

Whenever the counter of the error monitor reaches a programmer-defined threshold (currently 25 minutes), another component of the Python wrapper, the *Excel Killer* (Thread 4) comes into play. It issues an operating-system-level command to (violently) terminate the currently running Excel process. The reason why this has to be done violently is that Excel becomes entirely non-responsive whenever the Datastream download is hanging. Therefore, asking Excel to exit nicely does not result in Excel shutting down. The terminate request needs to run on a separate thread so as to not interfere with the Python gui and status updater.

After the current Excel process and thus also the running Datastream download have been shut down, the Python tool restarts the download request from the same point where it finished its download before it started hanging. In other words, the download request gets automatically resumed.

Another functionality which the error monitor provides is the ability to detect when all needed data has been downloaded. This happens in a manner similar to the error monitoring itself. The tool simply reads in the number of the last executed download request from the settings file. Based on this information and on the total number of requests to be executed, it determines when the last data batch has been downloaded and ends the download. A corresponding status message is delivered to the user interface to notify the user that the download has finished.

The entire Datastream Extraction Tool consisting of the VBA and Python parts, and including the required folder structure, is currently hosted on Google Drive under `https://bit.ly/3rB3lqg` $->$ *Datastream Extraction Tool*.

3. Data Preparation

As it is often the case with data science projects, the data preparation part takes time and effort. After the static and time series bond data has been extracted from Datastream – which will likely take several days – it is available in form of multiple data parts in Excel format. Since it is more convenient to perform further statistical analysis in a dedicated statistical environment, such as Stata or MatLab, the data needs to be brought into 'long' format, and additionally to be cleaned from null entries and outliers. The undertaken procedures are described in the following.

3.1. Data Formatting

For the static bond data, there is not much to be done in terms of formatting. Its original format, as downloaded from Datastream, is mostly suitable for further analysis and can be directly imported into Stata.

The downloaded time series data is initially in 'wide' format and has multiple bonds in one row. It looks like shown in Fig. 3.1.

Start	31.12.1999														
End	30.06.2020														
Frequency	D														
Name	LEGRAND SA	LEGRAND SA	LEGRAND SA	LEGRAND SA	LEGRAND SA	LEGRAND SA	LEGRAND SA	LEGRAND SA	LEGRAND SA	LEGRAND SA	LEGRAND SA	LEGRAND SA	LEGRAND SA	LEGRAND SA	LEGRAND SA
Code	233CX0(AC)	233CX0(YA)	233CX0(LF)	233CX0(MV)	233CX0(DM)	233CX0(CMPN	233CX0(MPD)	233CX0(CP)	233CX0(GP)	233CX0(RI)	233CX0(IY)	233CX0(RY)	6063KG(AC)	6063KG(YA)	6063KG(LF)
CURRENCY	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E
31.12.1999	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
03.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
04.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
05.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
06.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
07.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
10.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
11.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
12.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
13.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
14.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
17.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
18.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
19.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
20.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Figure 3.1.: Sample of downloaded raw time series bond data

The goal is to transform this time series data into 'long' format, as can be seen in Fig. 3.2, by saving the bonds one below the other. Additionally, the header has to be removed, and the *dscd* identifier of each bond as well as its currency have to be entered as a separate column for each date instead of being at the top.

I wrote a VBA macro with a function called *ToLongFormat* (which can be found in project files) which accomplishes the described task. While Stata might have also been able to do the formatting, I decided to work with VBA at this point, since it is native to the MS Excel environment. The main procedure is as follows:

1. Define data layout constants depending on the initial format: header height, number of time stamps, number of datatypes, bonds per block, and number of blocks. A

Date	AC	YA	LF	MV	DM	CP	CMPM	MPD	GP	RI	IY	RY	Code(dscd)	Currency
31.12.1999	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E
03.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E
04.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E
05.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E
06.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E
07.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E
10.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E
11.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E
12.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E
13.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E
14.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E
17.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E
18.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E
19.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E
20.01.2000	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	6086YF	E

Figure 3.2.: Sample of downloaded time series bond data in 'long' format

block is defined as all time stamps for multiple bonds which are located row-wise next to each other. An example can be seen in Fig. 3.1 where two bonds from the same firm, but with different *dscd* codes, are next to each other. There can be multiple such blocks one below the other in one Excel file, depending on how the data was downloaded.

2. For all data files (as there will be multiple for larger requests) and for all blocks within each file, remove the header rows, place bonds one below the other, and create columns for *dscd* and currency.

3. Add a newly created header row once at the beginning of the file. This header row can later be used to define variable names in statistical software.

Note that if the original Excel files are very large, i.e. with a high amount of securities or dates, Excel might reach its sheet length limit when running this macro. If you notice such behavior, there is another function shipped with this macro, called *SplitInSubfiles()*. You can use this function on your initial downloaded Excel data to reshape it into smaller-sized files, before transforming it to 'long' format.

3.2. Stata Import

As soon as the data has been formatted, it can be cleaned conveniently within statistical software. Since I am working with Stata, the Excel files with static as well as reshaped time series data can be simply imported to Stata with the *import excel* Stata command. It is possible to save frequently used Stata scripts in form of *.do* files to reuse them later. You can find all the do-files involved in this project in the respective folders for this project step.

The static bond data merely needs to be checked for duplicates, e.g. by using the *duplicates report* Stata command. Empirically, static data extracted from Datastream is significantly cleaner than historical pricing data. Therefore, all the following cleaning procedures only need to be applied to the time series data.

Because Stata can generally work with larger files than Excel, it makes sense to merge the imported time series files – now already in Stata format – to files of larger size, which will make further work more convenient. After the data has been imported to Stata, it can now be cleaned with the help of standard Stata procedures.

3.3. Data Cleaning

For cleaning, multiple different procedures need to be applied. Since they are all commutative, it does not make a difference in which order these are executed.

- Null values can be cleaned with the *drop* command, e.g. with drop if MPD=="NULL" | MPD==””. Be prepared for a lot of values to be deleted when working with historical bond data. This is due to many bonds having been issued recently and thus not having older price entries.
- Erroneous values which sometimes occur in Datastream typically have high length. Remove these with e.g. drop if length(var) > 40, with *var* being the variable names.
- Cast date stamps from string to date format. This can be done by generating a new variable for the date first (gen date = date(Date, "MDY")). Then the new variable should be formatted to be well-readable (format date %tdnn/dd/CCYY). After that the old variable *Date* can be dropped, so that the newly created *date* takes its place.
- Cast integer and double values that are coded as strings back to numeric, e.g. with the destring command.
- Remove duplicates based on the variables *date* and *dscd*. There should not be two or more different entries for the same security on the same date.

Keep in mind to manually check the resulting data. No matter how thorough the cleaning procedure, there might still be some erroneous data which needs to be cleaned up or removed manually. Besides the listed cleaning methods, the data should additionally be searched for outliers that can have a negative impact on the further analysis. However, different values and tuples can be considered outliers depending on the analysis scenario. Therefore, I decided to leave the (not necessarily erroneous) outliers in the dataset at this step of the project. They will be filtered out as shown in chapter 5 later on.

After the data has been cleaned, many tuples will have been deleted. To make further analysis more convenient, it therefore makes sense to merge all the single data files into one. While it depends on the size of the entire dataset, this should still be possible in most cases. Otherwise, e.g. two or three files can be produced in total. Files can be merged easily in Stata, e.g. by using the append command. Having done that, the resulting cleaned and compact data is now ready for future statistical analysis.

4. Matching

In order to analyze the lead-lag relationship of corporate bonds and stocks, we need a large, survivorship bias free database with both stock and bond returns for any given point in time. So far, we only have two separate databases – one with historical corporate bonds data, and one with historical equity data. Therefore, the two databases have to be joined into one, based on the company that issued both. The task is not as trivial as it might seem, since there is no unique company identifier available in both databases. In the following, the available matching options will be discussed, and the most suitable approach chosen.

4.1. Available Options

To begin with, the following extracted bond and equity parameters were considered for the matching:

- SEDOL code
- WKN code
- CUSIP-9 code
- ISIN code
- Worldscope identifier
- Company name

4.1.1. SEDOL

The SEDOL is a unique 7-character identification code which stands for 'Stock Exchange Daily Official List' [2]. It is issued for securities registered in the United Kingdom and Ireland by the London Stock Exchange. Despite being used to uniquely identify securities, it does not, in general, contain a unique issuing company identifier, because the codes are simply issued sequentially. For example, two bonds, which were both issued by Apple Inc., can have the SEDOL codes *BF43J24* and *BK9WPP6*, respectively. The only similarity between the two is that these were issued only two years apart, and thus have the *B* at the beginning in common. Besides, the identifier is not available in our stocks database, and only exists for securities of companies listed on the LSE.

4.1.2. WKN

The WKN is a German 6-digit alphanumeric security identification code and stands for 'Wertpapierkennnummer'. Since 2004, it is possible for companies to obtain a WKN with

a unique company identifier included [6]. A WKN includes a company identifier if it starts with at least two characters before proceeding with digits. However, not all companies make use of this opportunity when ordering a WKN for their securities. Taking into account that there are also multiple exceptions from the rule base of WKN identifiers, it is hard to use these as unique company identifiers. This is especially the case because WKNs are generally only available for German securities. Also, the parameter is not available in our existing equities database.

4.1.3. CUSIP-9

The CUSIP number is a unique identification number assigned to all equities and bonds that are registered in the United States and Canada [4]. The CUSIP consists of 9 alphanumeric characters, of which the first 6 comprise the unique issuing company identifier. The code is often used in one of its shorter forms, i.e. as CUSIP-8 and CUSIP-6. However, in our case, only the CUSIP-6 variant is of interest. It can be derived from CUSIP-9 by simply dropping the last three characters. The CUSIP-9 code is directly available in our equities database. In the bonds database, it can only be found directly for some of the securities in the so-called *local code* variable (LOC), which can be found in Datastream. Unfortunately, the CUSIP values entered in this variable are not very reliable. A workaround can be achieved by using the security ISIN, as will be explained in 4.3.

4.1.4. ISIN

The ISIN stands for 'International Securities Identification Number' and is an international standard way to uniquely identify securities [3]. The ISIN by itself is not a unique company identifier. However, it sometimes contains a company identifier as part of it. In particular, for U.S. and Canadian securities, the ISIN usually contains the Cusip-9 code, which, in its turn, contains a 6-digit unique company identifier. For U.K. and Irish securities, the ISIN usually contains the SEDOL code. And for German securities, the ISIN contains the WKN. Therefore, while the ISIN itself cannot be directly used for the matching, it can nevertheless be used to obtain missing matching code values by extracting them from the ISIN.

4.1.5. Worldscope identifier

The Worldscope Identifier is a 9-digit code issued by Worldscope, a Thomson Reuters' fundamentals product [10]. It is used to uniquely identify both issuing companies and securities. For U.S. companies, the Worldscope Identifier is identical with the CUSIP-9 code. For non-U.S. companies, a derived identifier is used, based on the country where the issuing company is domiciled, and also includes a unique company code. A more detailed explanation of the mechanics can be found in the Datastream database. Unfortunately, the Worldscope Identifier is only available in the equities database, and not for bonds. Therefore, it cannot be used for the matching.

4.1.6. Company name

As the 'method of last resort', the company name itself can be used to join the bond and equity databases. The problem with company names as identifiers is though that these are

not necessarily unique on the one hand, and also tend to have heterogeneous spelling – i.e. one and the same company can be spelled in multiple different ways across the database. To provide an example, the company names *THE WILLIAMS COMPANIES INCO* and *WILLIAMS PARTNERS L.P.* refer to the same company, but are written differently, which makes the join between the two databases ambiguous. Nevertheless, the approach can be a good starting point when other options are not available, and will be introduced in greater detail in the next section.

4.2. Fuzzy String Matching

Having considered the different options to perform the matching, it becomes apparent that the only unique identifier which can be reliably used for the task is the CUSIP-9 code. Additionally, the company name can be used to produce a solid baseline to start with. In this section, a matching approach called Fuzzy String Matching will be introduced as a means to join the datasets via company name. In the next section, a matching approach based on the CUSIP code will be explained.

The term Fuzzy String Matching [1, 9] – also called Approximate String Matching – refers to use cases when there are two or more strings which have the same meaning, but are spelled somewhat differently. There exist multiple approaches to measure the extent of 'difference' between strings. The formula which is most commonly used is the so-called Levenshtein distance [5]. It measures the minimum number of single-character edits required to change one given sequence into the other. An *edit*, in its turn, is defined as one of the three operations performed on a string:

- insertion
- deletion
- substitution

Depending on the implementation, a substitution of a character can count as either one or two edits. This is because a substitution technically consists of both an insertion and a deletion. For simplicity reasons it can be assumed to count as one edit just like the other two operations. To give an example, consider a company that is called *Apple Inc.* in one dataset and *Apple Incorp.* in the other. The Levenshtein distance between the two company names would be 3, because exactly 3 insertions need to be performed in order to produce *Apple Incorp.* from *Apple Inc.* These insertions are the 3 characters *o*, *r* and *p*. Alternatively, we can also start the other way around, from the company name *Apple Incorp.* In this case, we would need 3 deletions to produce *Apple Inc.* In particular, we would have to delete the 3 characters *o*, *r* and *p*.

While there exists a concrete formula which defines the Levenshtein distance, it is not very relevant in this context, since we will not be implementing the Levenshtein distance ourselves. Instead, we will make use of dedicated Python packages, which compute the Levenshtein distance between two strings behind the scenes in order to produce a similarity score. In this work, we consider the two packages *fuzzywuzzy*[1] and *rapidfuzz*[2] for

[1] https://pypi.org/project/fuzzywuzzy
[2] https://pypi.org/project/rapidfuzz

the task. In reality, these packages do slightly more than just computing the edit distance, depending on the particular function called. A detailed description of these packages' capabilities can be found in their respective documentation.

The concept will be used in our case to select for each company name from the bonds database the one from the equities database which has the smallest Levenshtein distance to it. This way, matching company names from the two datasets will be connected to each other to produce a join.

4.2.1. Fuzzy-Wuzzy

Fuzzywuzzy is the most commonly used package for fuzzy string matching in the Python community. Therefore, my first approach to perform the matching was with the fuzzywuzzy package. It requires additionally the package *python-Levenshtein* to be installed, so it can use its faster C implementation of the Levenshtein distance. The rest of the *fuzzywuzzy* package is programmed in Python.

To start with the implementation, the static bond and equity data needs to be read into *pandas*[3] dataframes to perform further operations on it. For this purpose, it is advisable to export the static data from Stata to the CSV[4] format, and then to import it in Python from CSV files. I explicitly discourage exporting data from Excel to CSV, because Excel has its own understanding of the CSV format, which might not be compatible with *pandas*.

After the data has been loaded into dataframes, one for bond company names, and one for stock company names, it can be fed into one of the predefined *fuzzywuzzy* functions. To start simple, two company names can be compared to each other with the built-in function ratio(), which computes the standard Levenshtein distance similarity ratio between the two sequences. Note that this function also takes into account whether the characters are capitalized or not. Thus, *Apple Inc.* and *apple Inc.* would produce a similarity ratio lower than 100% due to the difference in the capitalization of the first letter. This is a not very desired behavior for our use case, because we do not care much whether a company's name is written in upper or lower case letters. There exist modifications to this function, such as partial_ratio(), which can also detect similarities within substrings, or token_sort_ratio, which can detect similarities between substrings which are differently positioned. The function which is best-fitted to our use case is extractOne(). For each bond company name, it evaluates the Levenshtein similarity score with all stock company names. The one with the highest similarity score is considered a match.

In practice, this approach can be implemented with two nested for-loops, which is not very efficient, but necessary, because we have no index structure on our datasets. The resulting algorithm is thus somewhat similar to a standard Nested-Loop-Join. Faster approaches for unstructured data like a Sort-Merge-Join or a Hash-Join would not work, since the company names are not unique. For the resulting matching, it suffices to store the *dscd* pairs of the bonds and equities which were determined to be most likely join partners. By the *dscd* codes, any other static or time series data can be joined in later, because

[3] https://pandas.pydata.org

[4] CSV means comma-separated-values and is a frequently used data storage format. In the first row of a CSV file there are usually column headers, all separated by commas. In all further rows the single column values are stored, also separated by commas. The format is frequently used in data science applications due to it being lightweight and information-dense.

the Datastream code is a unique security identifier and present in all tuples from both static and time series databases. Keep in mind that if the Datastream code is called *dscd* for both stocks and bonds, you will first have to rename it in e.g. *bond_dscd* and *stock_dscd* to avoid ambiguity. The results of the matching can be exported from a *pandas* dataframe to CSV format again. This CSV file can then be imported into Stata for further processing.

Despite its convenience of rapid prototyping, the *fuzzywuzzy* package has turned out to be rather slow in practice. Based on time measurements for 1,000 bonds, it would need around 150 hours of computation time to complete the matching, if scaled up to all the bonds in our database. On the one hand, it is not surprising, since we have around 60,000 bonds and 80,000 equities in our respective datasets, and are running a nested loop join of a sort on them. On the other hand, a faster approach would be preferred to avoid long waiting times. A better approach to this task is provided by the *rapidfuzz* package.

4.2.2. Rapidfuzz

Rapidfuzz is a (rather unknown) Python package which takes *fuzzywuzzy* as a base, and improves it by not only implementing the Levenshtein distance in C, but also the rest of the package in C++. This and also some algorithmic improvements make it significantly faster than original *fuzzywuzzy*. It has a somewhat more complex interface, but provides a very noticeable boost to the matching program. The general approach is very similar to matching with *fuzzywuzzy*: The static data is imported to pandas dataframes in CSV format and gets joined with some of the functions the package provides. Since *rapidfuzz* is based on *fuzzywuzzy*, the author has kept some of the function definitions similar to the original package. Therefore, extractOne() is still the best-fitting matching function for our use case, because it returns the most similar matching partner to the given bond company name.

To additionally improve the matching quality, another optimization should be performed on both bond and stock company names before the matching is done. To reduce spelling differences à-priori, all company names should be cast to lower case, and all punctuation signs and special symbols should be removed beforehand. This will make sure that when the data is fed into the matching algorithm, it will only need to compare the similarity based on the semantics of the sequences, without taking into account 'unnecessary' symbols. The resulting performance is significantly improved compared to the *fuzzywuzzy* approach. The updated matching algorithm only needs around 7 hours to compute the matching, compared to the 150 hours from before, which is a running time reduction of 95%.

Just like before, the results should be stored in dataframe format first, and then exported to CSV for future needs. It is sensible to not only export *dscd* pairs, but also the similarity score between the two. This makes it possible to cut off insufficiently matched data for certain analyses depending on the needs. I set the original score cut-off at runtime to 90% similarity to avoid having many false positives. More on this in section 4.4. The entire matching program code can be found in the file *matching.py* as part of this project.

4.3. CUSIP Matching

Having accomplished a baseline matching with the Fuzzy String Matching approach, it can be further refined by using the CUSIP-9 identifier. As mentioned in section 4.1, the identifier is already available in the equities database. However, in the bonds dataset, it is only given as part of the *local code* (LOC), where it is not reliably entered, and thus cannot be used for matching. At this point, we can make use of the fact that the ISIN of U.S. and Canadian securities includes the CUSIP-9 identifier as part of it. This means that in order to obtain the CUSIP code for the bonds we simply have to take the alphanumeric characters 3 to 11 of the ISIN. Taking into account that we are actually interested in the CUSIP-6 instead of CUSIP-9, because only its first 6 digits represent a unique company identifier, it is enough to take the characters 3 to 8 of the ISIN.

As there is no complex data science involved here, the procedure can be done directly in Stata. At first, the static bond and equity data needs to be loaded into Stata, if not already done so during data preparation as explained in section 3. Since the CUSIP identifier can only be used to match U.S. and Canadian securities, both bonds and equities need to be filtered by these two countries. The rest of the countries can be dropped from the dataset for this purpose. Remember to always work on a local copy of the original dataset so as to not irreversibly lose data. The next step is to generate new variables for the CUSIP-6 identifier. For equities, this can be done with the Stata command gen cusip_6 = substr(cusip_9, 1, 6). For bonds – with gen cusip_6 = substr(ISIN, 3, 6). Finally, the two datasets need to be merged. This can be done with the merge command on a N:M relation. Herewith, the matching over the CUSIP-9 identifier is accomplished and the results can be saved.

4.4. Evaluation

The results of the two matching approaches are as following:

- With Fuzzy String Matching, around 26,032 bonds have found an equity matching partner. This equals appx. 42.89% of the total 60,688 bonds.
- With the CUSIP-9 approach, 6,460 (North American) bonds have found an equity match. This equals appx. 10.64% of the total 60,688 bonds.

While the Fuzzy String Matching approach has a significantly higher matching ratio, one should keep in mind that the CUSIP-9 results are more reliable, because it is a unique key attribute and not a fuzzy one. For further analysis, it makes sense to merge the two matching tables to come up with one single matching database of highest possible accuracy. For this, we can use the *append* command from Stata to simply concatenate the two datasets. Remember to make sure that both datasets have the same variables and same variable names in order for the union to work. For example, one might need to create an empty similarity_score variable in the CUSIP matching first, because it is present in the fuzzy string matching.

As soon as the matching tables have been concatenated, the resulting dataset will contain duplicates in respect to bond_dscd and stock_dscd parameters. You can check this by running the command duplicates report bond_dscd stock_dscd. This is due to the two

matchings having overlaps. While the duplicates should be removed, it is important to keep the CUSIP matching pair and not the fuzzy string matching one for each encountered duplicate. This is because for the CUSIP matching we can be 100% sure that it is accurate. The information that a bond and an equity are perfect matches, and not just based on a similarity score of e.g. 92%, might be useful in later analyses.

After the duplicates have been removed, the total matching ratio increases from 42.89% (26,032 bonds with fuzzy matching only) to 44.91% (28,254 bonds with both approaches combined). While the improvement is only minor, keep in mind that some of the matching pairs are now more reliable than with fuzzy string matching alone. It is hard to give a concise measure of how many matches are perfect matches without manually checking all of them. But since the similarity score was already over 90% for the fuzzy string matching, my estimation is that around 35%-40% of the pairs are perfect matches.

5. Statistical Analysis

Having prepared the bond data and matched it with the stocks, some statistical analysis can now be performed. In particular, we are interested in summary statistics of corporate bond returns [8], as well as in the analysis of the lead-lag relationship of corporate bond and stock returns. For this, the returns themselves first need to be calculated from daily prices. Additionally, as the matching has so far only been done for static data, it needs to be extended to historical return data as well. All in all, the following procedure for the statistical analysis arises:

1. calculate monthly bond returns
2. match bond and equity returns
3. calculate summary statistics
4. analyze lead-lag relationship

5.1. Monthly Bond Returns

In order to calculate the monthly bond returns, the following formula will be used:

$$R_n = \frac{P_n}{P_{n-1}},$$

where R_n stands for bond return in month n, and P_n for the bond price on last day of month n. To implement the formula in Stata, in the time series bond data only the last price of each bond for each month needs to be kept. The rest of the data can be dropped, as it is not relevant for the current analysis. The pricing parameter which will be used to calculate the returns is *MPD*. It represents the Datastream Selected Default Price, and is the most reliable price parameter in the extracted database. Other price parameters, such as e.g. *CP* (Clean Price), have significantly more missing values and are thus less suited for the purpose. Having kept the last monthly prices, a variable group consisting of the variables *dscd* and *month* has to be generated (*egen group* command). This group can then be set as a Stata time-series (*tsset*). Based on this time-series, the monthly return variable can be generated with the introduced formula.

5.2. Matching Bond and Equity Returns

To match the bond and equity returns with each other, we can make use of the matching that we accomplished in chapter 4. For this, we load the corporate bond returns, which we

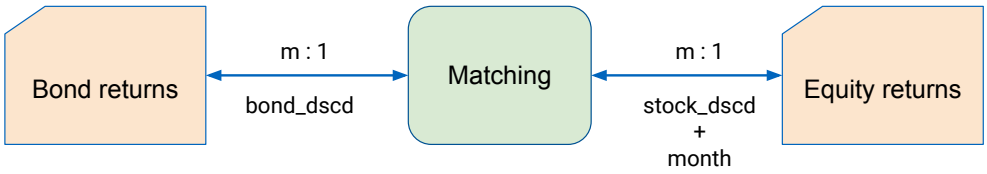


Figure 5.1.: Matching of Bond and Equity Returns

just calculated, into Stata. These bond returns can then be merged with the existing matching over the $bond\_dscd$ parameter, as shown in Fig. 5.1. After that, the resulting dataset has to be merged with the provided stock time series data, which also includes the required monthly stock returns. This merge should take place over the two parameters $stock\_dscd$ and $month$. This is because we want the bond and stock returns to be comparable for one and the same month later on. When executing the merges in Stata, keep in mind to make sure that the Datastream code parameter is named the same in the bond dataset and the matching (i.e. $bond\_dscd$), and similarly for the stocks and the matching (i.e. $stock\_dscd$). Otherwise, the matching will fail. The bonds-matching merge is performed on a M-to-1 relation, because multiple bonds can be mapped to the same equity over the issuing company. The merge with the equities is performed on a M-to-1 relation as well for the same reason. The resulting database builds the foundation for further statistical analysis.

5.3. Summary Statistics

Stata offers some very useful tools to determine the underlying distribution of the data, as well as to perform correlation and regression calculations. In order to obtain summary statistics separately for different bond groups, I split the bond returns dataset by geography and by credit rating.

Group	World	USA / CA	EUW	EUE	South America	China
Mean (%)	0.03%	0.06%	0.01%	0.00%	0.00%	-0.02%
Std. deviation (%)	2.82%	2.96%	2.72%	4.82%	3.68%	1.06%
Correlation with equity returns	0.1892	0.2298	0.1851	0.1010	0.1213	0.0290
Correlation with equity returns (1-m. lead)	0.0460	0.0557	0.0323	0.0494	0.0859	-0.0451
Correlation with equity returns (1-m. lag)	0.0404	0.0515	0.0366	0.0029	0.0397	0.0336
No. bond-month observations	683,086	403,967	99,928	4,837	24,258	29,202

Figure 5.2.: Monthly Return Statistics by Bond Geography

Fig. 5.2 features monthly return statistics for:

- The entire world
- North America region (NA), represented by USA and Canada
- Western Europe region (EUW), represented by Germany, France, UK, Italy, Ireland,

Luxembourg, Netherlands, Spain, Portugal, Belgium, Austria, Switzerland, and Scandinavian countries

- Eastern Europe region (EUE), represented by Poland, Czech Republic, Bulgaria, Greece, Serbia, Croatia, Hungary, Montenegro, Ukraine, Turkey, and Georgia
- South America region, represented by Brazil, Argentina, Chile, Colombia, Uruguay, Paraguay, Panama, Peru, and Mexico (though NA)
- China

Note that the list is neither regionally exhaustive, nor is it 100% geographically accurate. Its sole purpose is to gain a general idea of the distribution of corporate bond returns for countries with a similar financial structure.

Fig. 5.3 features monthly return statistics for different credit ratings of corporate bonds. Please note that the number of bond-month observations for the single credit ratings does not sum up to to the number in the *All* column. This is due to the procedure which was used to split the bonds up by their rating. To give an example, bonds listed in rating category *BBB* are those bonds which have either a *BBB* rating as given by the S&P agency, or a *Baa* rating as given by the Moody's agency. In some cases, it occurs that e.g. the Moody's rating is given as *A*, and the S&P rating as *BBB*. In this case, the bond would be listed in both the *A* and the *BBB* categories, which results in intersections between the single categories. The procedure has been chosen this way to receive a larger amount of data for analysis, as e.g. entries for the Moody's rating alone were somewhat scarce. If needed, further analysis can be done based on ratings of a single agency only.

Group	All	AAA	AA	A	BBB	BB	B	CCC	Other / Unk.
Mean (%)	0.03%	0.14%	0.08%	0.09%	0.10%	0.08%	-0.03%	-0.05%	-0.05%
Std. deviation (%)	2.82%	2.36%	1.56%	2.02%	2.36%	2.91%	4.09%	4.62%	4.64%
Correlation with equity returns	0.1892	0.0815	0.0160	0.0360	0.0986	0.2920	0.3716	0.3671	0.1882
Correlation with equity returns (1-m. lead)	0.0460	-0.0546	-0.0435	-0.0105	0.0351	0.0430	0.0472	0.1214	0.042
Correlation with equity returns (1-m. lag)	0.0404	0.1132	0.0479	0.0638	0.0631	0.0240	0.0099	0.0455	0.0163
No. bond-month observations	683,086	4,853	40,457	136,515	308,298	81,106	32,656	13,166	123,768

Figure 5.3.: Monthly Return Statistics by Bond Rating

The distribution mean and standard deviation of the bond returns are largely in-line with my expectations. Bonds in regions with a majority of developed countries generally have a higher mean return and a lower standard deviation than bonds in regions with more developing countries. Similarly, investment grade bonds have higher mean return and lower volatility than non-investment grade bonds.

The correlation with equity returns looks higher for developed regions, but at the same time lower for investment grade bonds. Additionally, the correlation of monthly bond returns with lagged stock returns strictly monotonically increases with falling credit rating. As such, the lowest bond-lead correlation can be seen for corporate bonds rated *AAA*, while the highest correlation is listed for *CCC* rated bonds. This already explains quite well why existing research on the topic of the lead-lag relation of corporate bonds and stocks mostly focuses on high yield bonds, as these have the highest lead correlation with stocks.

The correlation results of stock returns with lagged bond returns are less conclusive, but tend to be higher for investment grade bonds. From a geographical point of view, there seems to be no significant insight for the lead and lag correlations with equity returns.

5.4. Lead-Lag Relationship

In order to estimate the existence of a potential lead or lag relationship between corporate bonds and stocks, multiple regression analyses have been run. In particular, the calculations were conducted for the following bond groups:

- all corporate bonds
- split by geographical region: North America, South America, Western Europe, Eastern Europe, and China
- split by bond credit rating: AAA, AA, A, BBB, BB, B, CCC, other
- split by aggregated bond credit rating: investment grade and non-investment grade bonds
- two rating-based approaches to high-yield bonds

All regression analyses have been run as a duo of:

1. regression with monthly bond returns as dependent variable and 1-month lagged stock returns as independent variable
2. regression with monthly stock returns as dependent variable and 1-month lagged bond returns as independent variable

For shortness, not all the regression results will be introduced here in detail. This section will mostly focus on high yield bonds, as these have shown the most likely lead correlation with equities in section 5.3, and also because the leads and lags of high yield bonds have seen the highest attention in the existing literature on this subject so far (e.g. [11]). All the other regression results can be found either in appendix B to this work, or as part of the project files.

Fig. 5.4 features the regression results for high-yield corporate bond return as dependent variable, and stock return as independent variable. Specifically, high-yield bonds have been taken as those with either a *Ca* or a *C* current rating provided by Moody's. From a statistical perspective, the following interpretation of the results can be given:

- The *model sum of squares* (*MSS*) is not very close to the *total sum of squares* (*TSS*), which means that the fit of the model function is not very tight.
- The *p-value* (here $Prob > F$) is noticeably lower than 0.05. This means that the results of the regression are statistically significant.
- The r^2 is the amount of variance of bond returns that is explained by the lagged equity returns. The number is rather low, though it is the highest achieved in all the regression analyses run. Apparently, equities are best in explaining the variance of high-yield bonds, compared to other bond groups.

Source	SS	df	MS	Number of obs	=	2,120
				F(1, 2118)	=	89.30
Model	.23193612	1	.23193612	Prob > F	=	0.0000
Residual	5.50074596	2,118	.002597142	R-squared	=	0.0405
				Adj R-squared	=	0.0400
Total	5.73268208	2,119	.002705371	Root MSE	=	.05096

bond_return_w	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
lagged_stock_return_w	.0583903	.0061788	9.45	0.000	.0462732	.0705075
_cons	-.0013249	.0011069	-1.20	0.231	-.0034957	.0008459

Figure 5.4.: Results: regression with high-yield corporate bonds (as by Moody's) as dependent variable, and stocks as independent variable

- The *root mean-squared-error* is the standard deviation of the regression model. It is close to 0, which means that the single bond return values do not deviate much from the distribution mean on average. On the other hand, the bond returns themselves are not very high, so even an absolutely small standard deviation can have a significant impact on the spread.
- The modeled regression function itself has the shape $bond\_return_n = -0.0013 + 0.0584 * stock\_return_{n-1}$, with n being the month, at the end of which the return occurred. The magnitude of the steepness factor and the intercept seems in-line with our expectations.
- The *t-values* signify the importance of the variable in the regression model. An absolute *t-value* greater than 1.96 is usually considered as good, which is here the case for the steepness coefficient.
- The *two-tail p-value* (here $P > |t|$)is lower than 0.05 for the growth coefficient, which means that it is statistically significant. The value for the intercept is not very significant at 0.231.

Overall, the results show that stock returns lead the returns of high yield bonds on a 1-month basis, though the results are not entirely conclusive. My recommendation is to use a more sophisticated (e.g. portfolio-based) regression procedure, and potentially to also try using a non-linear regression model to receive more reliant results.

Fig. 5.5 shows the regression results for high-yield corporate bond return as independent variable, and stock return as dependent variable. The same definition of 'high-yield' applies as for the first regression. Compared to regression results with bonds as dependent variable, the results with stocks as dependent variable are somewhat less significant, as can be seen e.g. by looking at the p-value an the two-tailed p-values. The model function fit is also somewhat worse when looking at the model sum of squares compared to the total sum of squares, as well as based on the r^2 parameter and the standard deviation of the model. Based on the t-values and the two-tail p-values, the results are statistically significant for the growth coefficient, but not so for the intercept. The model function itself

Source	SS	df	MS	Number of obs	=	2,124
				F(1, 2122)	=	11.43
Model	.3648643	1	.3648643	Prob > F	=	0.0007
Residual	67.7542046	2,122	.031929408	R-squared	=	0.0054
				Adj R-squared	=	0.0049
Total	68.1190689	2,123	.032086231	Root MSE	=	.17869

stock_return_w	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
lagged_bond_return_w	.2345471	.0693841	3.38	0.001	.0984792	.3706149
_cons	-.0012946	.0038796	-0.33	0.739	-.0089028	.0063136

Figure 5.5.: Results: regression with high-yield corporate bonds (as by Moody's) as independent variable, and stocks as dependent variable

is shaped as $stock\_return_n = -0.0013 + 0.2345 * bond\_return_{n-1}$, with n being the month like before.

As the results of the second regression model are not as reliable and significant as those of the first one, it can be concluded that stocks are likely faster in incorporating significant market information than high-yield bonds, for one and the same company. However, due to the chosen procedure being very basic, it is advisable to conduct a more thorough regression analysis to draw up more reliable conclusions.

6. Conclusion

To conclude the presented work, a brief summary of the results as well as an outlook to future research is provided in the following.

6.1. Summary

In the scope of the Interdisciplinary Project on the topic *Leads and Lags of Corporate Bonds and Stocks*, a data extraction tool tailored to the interface of Refinitiv Datastream has been developed. It allows the user to conveniently enter request details for the desired financial data and to download it with significantly lesser overhead than manually via request tables. To prepare the downloaded data for future usage, a comprehensive formatting and cleaning procedure has been developed and documented. In order to make the extracted bond data comparable with equities, a matching algorithm based on the fuzzy string matching and the CUSIP-6 approaches has been derived, and can be extended to further use cases. Finally, a brief statistical analysis of the extracted data, in particular of calculated corporate bond returns has been conducted, and found to be in-line with widely accepted assumptions of the underlying statistical distributions. A basic analysis of the lead and lag relationships between corporate bond and stock returns has produced statistically significant results, according to which stock returns are faster in incorporating new information, and thus predict future corporate bond returns in the case of high-yield bonds.

6.2. Outlook

While the bond and equity data extracted from Datastream is of relatively high quality, it could still benefit from a cross-check with corresponding data samples from other financial data providers, such as e.g. Bloomberg or Markit. Both for this purpose, and to further improve the existing bond-stock matching, it is recommended to estimate automated download potential for other financial data sources. A notable mention would be for example the Bloomberg Python API and its likes. For the matching procedure specifically, it would make sense to search for and evaluate additional unique company identifiers to make key-based matching possible for the entire dataset.

Concerning the statistical analysis part, I suggest to conduct a more complex and intensive series of regression analyses to extend the existing lead and lag relationship results. The returns for the regression could be prepared portfolio-wise and the model chosen to be a non-linear one to improve on the current r^2 statistic by reducing variance that is currently not induced by the independent variable. If desired to go a step further, a neural network approach with 2 to 5 hidden layers could be tried to fit the prediction model more tightly. The need for such an extensive approach should be analyzed thoroughly though,

as there might be too few learnable features, which can cause such a complex model to overfit. Finally, the calculated regression model could be tested in real life on newly incoming bond data over a one or more years period to estimate its practical usability for the most reliable bond classes and regions.

Appendix

A. Datastream Extraction Tool Manual

The full version of the Datastream Extraction Tool Manual can be found in the project folder of the Google Drive repository under `https://bit.ly/3rB3lqg`.

A.1. Introduction

The Datastream Extraction Tool is an application that allows the user to download financial data from Thomson Reuters Datastream without the overhead of doing it manually in Excel. The tool was created specifically for large static and time-series requests, and supports long-duration downloads of data.

A.2. Installation

Please, be aware of the following:

- The application can only be run on Windows computers. It cannot be ported to Mac-OS or Linux, as the Thomson Reuters Excel add-in is only available for Windows, and also because the VBA (Visual Basic for Applications) version on Unix-based computers is a different one than on Windows computers.
- While the application can generally be run on remote computers as well, it can be rather inconvenient. On the one hand, remote computers tend to be less stable in terms of internet connection and add-in connectivity. And additionally, the remote computers of TUM-SOM have rather poor performance (especially disk access!), which would constrain the data download. Furthermore, remote computers automatically shut down after some time, which makes downloading large data loads problematic. For smaller requests though, that take up to 1 hour of download time, the remote execution should suffice.

A.2.1. Prerequisites

Please make sure that the following environment is established before you proceed with the installation.

- Make sure you have a valid **Microsoft Office Installation**. The application was tested with Office 365, but it should also run with other recent Office environments.
- Make sure that **Thomson Reuters Eikon & Datastream** is installed. If you install it for the first time, please ensure that the Datastream add-on in the Excel plug-in is enabled.

- Make sure that you have an up-to-date version of **Python 3** installed. Python 3 is different from Python 2; with Python 2 the application will not work.
- Make sure that in Windows Power Options you select that your PC "never" goes to sleep. Since otherwise this can interrupt long-running downloads.

A.2.2. Download

Follow these steps to install the application on your computer:

1. Download the *Shippable* folder from `https://bit.ly/2VfT058` to your computer. Choose a location with enough free space, as the downloaded data will be stored in-place.
2. Unzip the downloaded file. You can rename the root folder from *Shippable* to a different name at your convenience. Please, do **not** rename any of the folders or files inside. Such changes would need to be propagated to the program code.
3. In the *Shippable* folder, you will find a file called ***prerequisites.py***. You need to run this file with Python. You can do so by e.g. right-clicking on the file, selecting "Open with" and then choosing Python. It will then automatically install external Python packages needed for the app.

A.2.3. Settings

Finally, you will need to adjust some settings in Excel the first time you install the application. For this purpose, open the file called *RequestTable.xlsm* in the *Shippable* folder.

The first time you open it, you might get prompted to activate file contents or to allow modifications, etc. Please agree to all such messages.

Further, take care of the following settings:

- Make sure that both the *Thomson Reuters* tab and the *Thomson Reuters Datastream* tab show in the tab panel of the Excel document.
- In the Thomson Reuters tab go to Settings => Sign-In => choose to automatically sign-in whenever Office is started.
- Go to File => Options => Trust Center => Trust Center Settings and do:
  - Macro Settings => select "*Enable all macros*" and check "*Trust access to the VBA project object model*".
  - Protected View => uncheck all boxes except for "*Outlook attachments*".
  - Add-ins => uncheck all boxes.
  - External Content => select "*Enable all Data Connections*", "*Enable automatic update for all Workbook Links*", "*Enable all Linked Data Types*".
- Go to File => Options => Customize Ribbon => select *Main Tabs* on the right => check the *Developer, Add-ins, Thomson Reuters*, and *Thomson Reuters Datastream* tabs.

- Go to File => Options => Advanced => section *General* => uncheck option *Ask to update automatic links*.
- Go to File => Options => Add-ins and make sure that the add-in *Thomson Reuters Eikon - Microsoft Office* is listed among *Active Application Add-ins*. It should be. If not, refer to the Troubleshooting section.
- Press "Alt + F11" (the VBA editor will appear) => go to Tools => References => check the box near "*Microsoft Scripting Runtime*"

Now, you should be well set to run the application.

A.3. Usage

You can start the Datastream Extraction Tool by running the file ***ds_extraction_tool.py*** with Python.

If you encounter a problem, some other prerequisites on your computer might be missing that might not have been covered in this guide. In that case, please contact the project developer.

A.3.1. User Interface

Once you start the application, the user interface (in the following gui) as in Fig. A.1 will show. It has the sections:

- Request Type
- Time Frames
- Request Datatypes
- Data IDs
- Request Format
- Save Results To

In the **Request Type** section you can select your request to encompass either static or time-series data for the stocks or bonds that you want to download.

If you select the option *Time Series*, the **Time Frames** section will become enabled. There, you can enter the start and the end date of the period for which you want to download data. The dates have to be entered in the format "dd/mm/yy" (e.g. 01/01/99 for the 1st of January 1999, or 30/06/20 for the 30th of June 2020). The program assigns year numbers from 51 to 99 automatically to the years 1951 - 1999, and year numbers from 00 to 50 to the years 2000 - 2050. In the *Frequency* text field, you can type in the frequency of the time points that you want to get. Here, the values "Daily", "Weekly", "Monthly", "Quarterly", and "Yearly" are allowed.

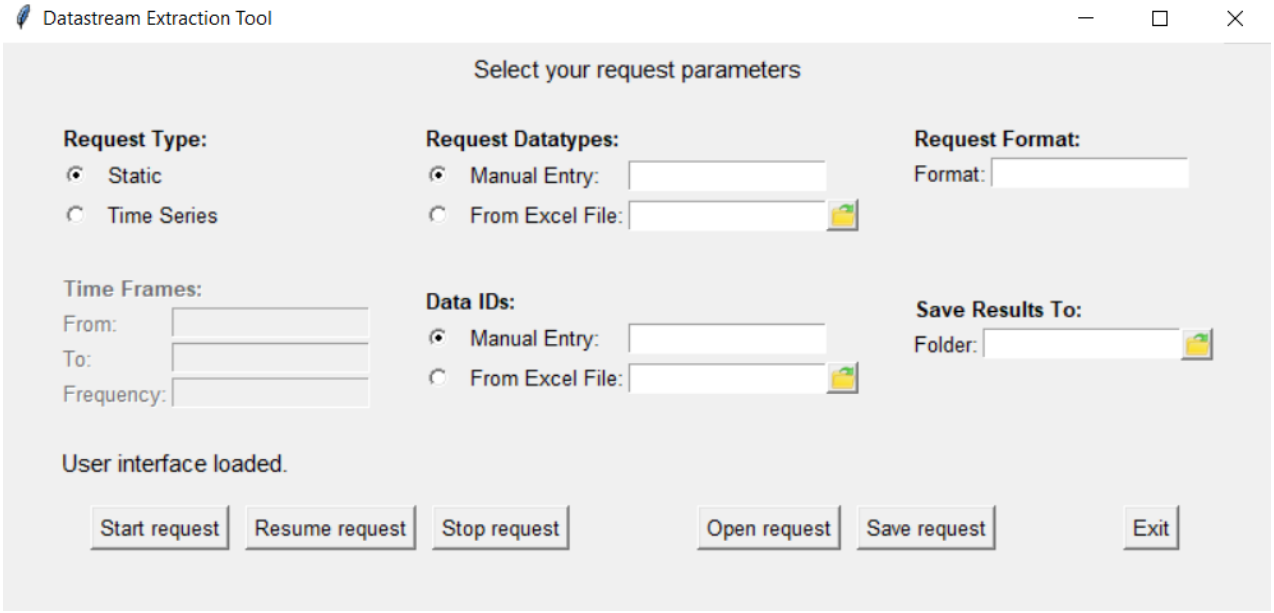


Figure A.1.: Graphical User Interface of the Datastream Extraction Tool

Within the **Request Datatypes** section, you can enter the codes of the parameter types that you want to get for your stocks or bonds. These could be e.g. "C" for coupon, or "ISIN". You can enter the values either manually, or by choosing the excel file from which you want to get the datatypes. If you choose to enter the datatype codes manually, please do so in a comma-separated manner (e.g. "C,ISIN,AIS,BSTAT"). If you choose to enter the codes via excel file, you have to create an excel file with all the datatype codes listed in column A, one below the other (see Fig. A.2). Then, within the app, browse for the created file and select it.

	A	B
1	AIS	
2	BSTAT	
3	BTYP	
4	BGEO	
5	CCDE	
6	DSCD	
7	GEOG	
8	INDC	
9	ISIN	
10	LOC	
11	SECD	

Figure A.2.: A sample file with datatypes

In the **Data IDs** section, you have to enter the datastream codes of the financial instruments - i.e. stocks or bonds - that you are interested in. Just like for datatypes, this can be done either manually, or by giving the excel file name, in which you have previously

stored the IDs. When entering them manually, separate the IDs by commas again. When entering them via file, store them in the A column of your excel document, like before, and select the created file in the app by clicking on the "Browse" button.

In the **Request Format** section you can (optionally) enter the format in which you want your data to appear. You can give the format by concatenating the letters of the different format options available in Datastream. For example, "CRM$" would mean to include column headers (C), row headers (R), instrument code (M), and currency ($).

Finally, in the **Save Results To** section of the interface you can enter the folder location to which you want your request results to be saved. For this, either enter a valid folder path by hand, or select it by clicking on the button to browse for a suiting folder.

Below the parameter sections, there is a **status bar** that shows you the current status of your request. This can be especially useful for longer requests. Please keep in mind that the status bar can have a delay of up to 2 minutes, depending on the current request phase. For real-time request updates, please refer to Logs in the Folder Structure section of this guide.

Additionally, at the bottom of the gui, there are several control buttons. The "**Start Request**" button first checks the format of the fields you entered, and subsequently launches the request. The current status of the request will be shown in the status bar.

The "**Stop Request**" button shuts down the request if there is one currently running. In its essence, it kills the running Excel instance to stop the data download. Depending on the phase of the request, it might take up to 2 minutes until the request has been stopped. Please abstain from using the gui during this time.

The "**Resume Request**" button is there to resume a request that was previously started and then stopped. This action ignores any entered parameter fields, as it simply resumes the last request, with the settings it has internally saved. Requests that were resumed automatically continue to download data, starting from the last data chunk that has previously been downloaded. Use the "Resume Request" button if your download crushed or if you stopped it manually.

The "**Open Request**" and "**Save Request**" buttons are there to either save current parameters into a file, or to conveniently load request parameters from a file. Use this for recurring requests to avoid entering them by hand each time.

Finally, the "**Exit**" button first stops the request if there is one running, and then closes the application. If you want to close the application without stopping the request, close the window as usually with the cross at the top right of the window. Note that in this case Excel might proceed running in the background!

A.3.2. Application Folders

The application is shipped with several folders that it requires to run. These are:

- Data
- DataTypes
- IDs
- Logs
- PythonPackages
- Requests
- Settings

In **Data**, the downloaded data from the users' requests is getting stored. It is stored in the form of Excel files. The number of files varies depending on the request size. The file size is centered around 40 MB each, but can also vary, depending on the particular request and the available data. You can use the files with downloaded data for whichever purpose you want. It is advisable though to avoid further manipulating these files in-place, and rather to copy them elsewhere for further processing.

In the **DataTypes** folder, you can put Excel files with the datatypes of financial instruments that you want to download. These have to be stored in column A, one datatype per row. The application is shipped with several examples.

In **IDs**, you can put Excel files containing Datastream identifiers of financial instruments, for which you would like to download data. Just like for datatypes, you have to store these in the A column of the respective Excel file, one ID per row. Refer to section IDs Preparation for hints on how to do it conveniently.

In the folder **Logs**, you will find two files, which are generated when the program is running. The first one is called *log_python.txt*. It contains real-time information on the progress of the data download. Use the file to either check the current download status, or to track occurring errors, should the application misbehave. The second file is called *log_request_table.txt*. It contains real-time information on the progress of the data download on the Excel side of the program (i.e in the request table). You can similarly use it either for progress updates, or for troubleshooting.

The folder **PythonPackages** is of technical nature. It contains python packages that are needed for the application to run. Please, do not touch this folder, unless you know what you are doing.

In the folder **Requests**, you can store parameters of previous requests, in order to reuse these later. The requests are stored as .txt files. The file names you can choose yourself at

creation.

The folder **Settings** contains one single file called *settings.txt*. This file is used for multiple purposes, including:

- communication between the python and the excel part of the application
- download progress monitoring
- error monitoring
- status updates in the gui
- the "resume request" functionality

The settings and communication data within the file is stored in the key-value-format. In most cases, you should **not** open or modify the file. For exceptions, refer to the Troubleshooting section.

A.3.3. Troubleshooting

3.3.1 Thomson Reuters Eikon add-in does not show in the list of active Excel add-ins

Problem Description:
Thomson Reuters Eikon add-in does not show in the list of active Excel add-ins.

Solution Proposal:
In the *RequestTable.xlsm* document, navigate to File => Options => Add-ins. If the add-in is listed under "Deactivated Application Add-ins" in the Add-ins tab of Options, then at the bottom select *Manage: Deactivated elements* and click on "Go". In the appearing window, if the Thomson Reuters add-in is listed, check it and click on "Activate".
If the Reuters add-in is listed under "Inactive Application Add-ins" instead, then at the bottom select *Manage: COM Add-ins* and click "Go". In the appearing window, if *Thomson Reuters Eikon - Microsoft Office* is not checked, check it. If the Reuters add-in is not in the list, click on "Add" and navigate to the add-in location. In many cases, it is stored under `C:\Users\[current_user]\AppData\Local\ThomsonReuters\Eikon\EikonOfficeShim.dll`. Select it and add it to the list. If not, find out where the file *EikonOfficeShim.dll* is stored on your computer and add it to the list of add-ins.

3.3.2 Sign-In Problems with Thomson Reuters Eikon Add-in

Problem Description:
The Excel add-in of Thomson Reuters Eikon & Datastream sometimes logs-out for unclear reasons, or because a different user logged-in with the same account data. In most cases, this results in an Excel message during download, which states that the user is not signed-in to Thomson Reuters.

Solution Proposal:
Sign-in to Thomson Reuters manually in Excel. Then restart or resume the download. If this does not solve the issue, try restarting Windows.

3.3.3 Download Problems

Problem Description:
One of the following hanging problems applies:

- The download seems to hang, but the app does not recognize it.
- The app recognized the hanging, but cannot shut down the download.
- The app shut down the download due to hanging, but Excel keeps running.
- You stopped the request manually, but it seems to still be running. Neither the app, nor Excel respond, or the entire computer seems to hang.

Solution Proposal:
Open Task Manager. Kill the Excel task first, then the extraction tool. The extraction tool will be shown as "Python" in the task list. Then restart the extraction tool and resume the download.

A.4. Technical Aspects

In this section, a technical description of the application architecture will be provided. For an illustration of the entire architecture, see Fig. A.3.

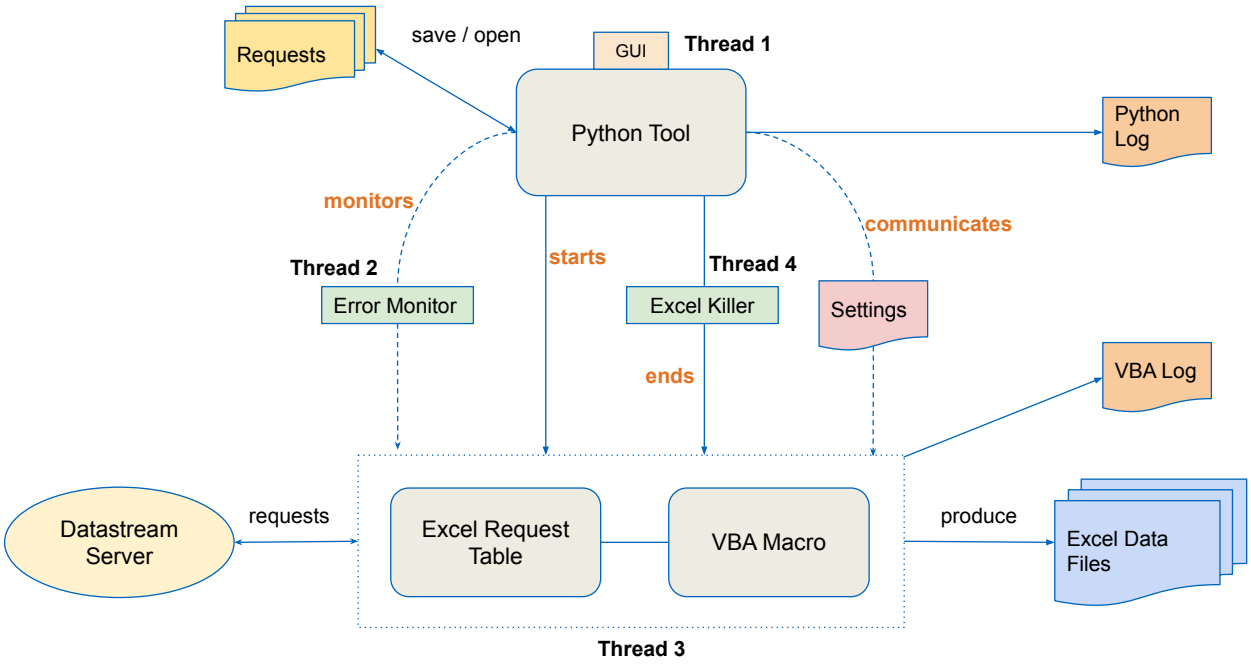


Figure A.3.: Datastream Extraction Tool Architecture

A.4.1. Excel / VBA Part

The Visual Basic for Applications part of the data extraction tool consists of four modules, each serving its own purpose:

- modDeclares
- modVariables
- modUtils
- modTS

Within *modDeclares*, operating system level constants and functions are defined. The declaration of such functions with *Declare* and of constants with *Const* equals an import of this functionality to the project. If you are not familiar with the reasoning behind the usage of *PtrSafe* and *LongPtr* keywords, you can read it up here: .

The module *modVariables* simply contains all the global variables used within the project, split by their usage area as described in the annotations.

In *modUtils*, a large amount of helper functions are defined. About half of them are self-written, the others come from solutions available online. While I cannot guarantee for all of them to always work correctly, it is unlikely for them to contain bugs, since I extensively tested most of them in different contexts. The most important functions are the ones I wrote to wrap the communication with the *settings.txt* file (InitDict, GetFromDict, UpdateDict, PutKV, and GetKV), and those responsible for the Datastream Add-In connection (Connect_COM_AddIn, InstallCOMAddIn, iAddInStatus, CheckPluginConnection). Make sure to familiarize yourself with them if you are going to make changes or extensions to the tool.

The module *modTS* is the main module of the tool, in which most of the action happens. It is well-modularized by itself and contains the following functions:

- getTSData
- start
- init
- setDefaultSettings
- getIDs
- getDatatypes
- doCalculations
- fillTableIDs
- fillTableFields

- request

The order of execution when the tool is started directly corresponds with the position of the functions in the code (like above). *getTSData* is the only function declared as *public* in the module, and is the entry point to the program. This function gets called from the Python part of the tool whenever a request has been started by the user. It calls the function *start*, which sets default constants, checks the Datastream add-in connection, unpacks user-provided settings from the *settings.txt* file, and calls the *init* function if this is the first time this request is executed. The *first_time* is set in the *settings.txt* file originally, and is received from there via the key-value dictionary wrapper. If this is the first time one and the same request is being started, the function *init* initializes global variables with values from the settings dictionary. Additionally, the function initializes a number of other functions which fill the request table with contents needed to execute the user request.

At first, the current request table sheet is cleared from previous contents (*ClearCurrentSheet*). Then, data identifiers are loaded from the user-provided file with IDs (*getIDs*). After that, the same is being done with datatypes (*getDatatypes*). Having received all the required input data, some calculations are being carried out (*doCalculations*), which are there to estimate the optimal number of identifiers for a single request to Datastream. This is necessary, because Datastream has a maximum request size, both by number of submitted IDs and by the amount of data downloaded. Make sure you fully understand the calculations before making changes to this function. It is the likely the most complex one in the entire program.

As the next step, the request table gets filled with contents. For this purpose, first the function *fillTableIDs*, and then *fillTableFields* is executed.

After all these steps have been carried out, the actual download of the data starts. It is initiated in the main function *getTSData* for each of the parts that need to be downloaded. A single request is being handled by the function *request*. This is the only place in the code, where the call to the actual Datastream API happens via the provided interface function *btnProcessTable_Click*. The function itself can be found in the *basEvents* module, which is always available in a Datastream request table by default.

In terms of additional functionality, the tool provides logging throughout the code to enable troubleshooting when needed. The logs are stored in the *Logs* folder of the tool.

A.4.2. Python Part

In order to fully understand the architecture of the Python side of the application, take a look at Fig. A.3 again. The main point of entry into the user-side application is the user interface (gui). It runs on the main thread of the program, and is maintained by the module called *ds_extraction_tool*. The user interface has been created with the package *Tkinter*. It provides a way to relatively fast create user interfaces, even though they are not very flexible in terms of appearance. Most of the *ds_extraction_tool* module deals with gui elements and user interaction with them.

The most interesting functions are start_*request*, *resume_request* and *stop_request*. start_*request* and *resume_request* both put user-defined request settings, which were previously entered by the user in the user interface, into the *settings.txt* file, which in its turn is used to communicate with the VBA part of the program. The settings file on the Python file is wrapped into a key-value dictionary, similarly to the way it is done in VBA. The dictionary implementation is self-made and can be found in the module *kv_file_manager*.

After filling the settings file with user contents, a new thread (Thread 2) gets started, in which the module *excel_controller* runs. The module is responsible for starting up Excel, and for submitting the actual download request to VBA. A new thread is needed in order to keep the gui responsive throughout the download. Additionally, the module performs error monitoring and progress control at runtime. This is the most complex module on the Python side of the program. Take some time to familiarize yourself with it. The module receives the call from *ds_extraction_tool* which lands in the *start* function. The *start* function, in its turn, calls the *run* function. The *run* function creates a new thread (Thread 3), in which Excel is started, the request table opened, and the VBA macro *getTSData* launched. In order to submit all this to Thread 3, this functionality needs to be wrapped in a non-class function called *run_rt*, also available in the module.

After the VBA macro and thus the download have been launched, the module *excel_controller* keeps monitoring the execution by regularly checking the contents of the *settings.txt* file. This is done by the two boolean functions *finished_request* and *download_hangs*. If the module notices that the download has finished (by comparing currently downloaded data part with total number of parts in the settings file), it stops the execution and issues a message to the user that the download has completed. If it notices that the download is likely hanging (by reaching a counter timeout on the same data part being downloaded for a long time), Excel, and thus the running macro, gets killed so that the hanging download can be stopped. This is handled by the module *excel_killer*. The process responsible for stopping Excel also runs on a separate thread (Thread 4), in order to prevent interference with the *excel_controller* module and the gui. After that, Excel gets restarted, the request table reopened, and the download resumed at the same download stage where it was interrupted.

Additionally, the Python side of the application also writes a log, which can be found in the *Logs* folder of the application, and delivers download status updates to the gui module to notify the user of what is currently happening. The latter is done over the object called *obj_update*, which is reached over to the *excel_controller* module by the gui when the user request is being forwarded to Thread 2.

The module *variables* stores all the global variables utilized on the Python side. The module *registry_killer* is there to kill a registry entry Excel keeps settings when the Datastream add-in misbehaves. If the entry is not deleted at each program start, it might be that the Datastream add-in is blacklisted by Excel, and thus cannot start. The functionality is essential for the tool to work reliable. Finally, the module *log_file_manager* is responsible for writing the log file.

B. Lead-Lag Regression Results

B.1. By Geography

Source	SS	df	MS		
Model	.95594816	1	.95594816	Number of obs	= 668,343
Residual	450.569341	668,341	.000674161	F(1, 668341)	= 1417.98
				Prob > F	= 0.0000
				R-squared	= 0.0021
				Adj R-squared	= 0.0021
Total	451.525289	668,342	.00067559	Root MSE	= .02596

bond_return_w	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
lagged_stock_return_w	.0135846	.0003608	37.66	0.000	.0128775	.0142917
_cons	.0007835	.0000318	24.63	0.000	.0007211	.0008458

Figure B.1.: Regression result: World; bonds as dependent, lagged stocks as independent variable.

Source	SS	df	MS		
Model	8.48338375	1	8.48338375	Number of obs	= 668,673
Residual	5188.10594	668,671	.007758832	F(1, 668671)	= 1093.38
				Prob > F	= 0.0000
				R-squared	= 0.0016
				Adj R-squared	= 0.0016
Total	5196.58932	668,672	.007771507	Root MSE	= .08808

stock_return_w	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
lagged_bond_return_w	.1260595	.0038123	33.07	0.000	.1185875	.1335315
_cons	.0055323	.0001077	51.36	0.000	.0053212	.0057434

Figure B.2.: Regression result: World; stocks as dependent, lagged bonds as independent variable.

Source	SS	df	MS	Number of obs	=	396,350
				F(1, 396348)	=	1231.53
Model	.931992058	1	.931992058	Prob > F	=	0.0000
Residual	299.946514	396,348	.000756776	R-squared	=	0.0031
				Adj R-squared	=	0.0031
Total	300.878507	396,349	.000759125	Root MSE	=	.02751

bond_return_w	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
lagged_stock_return_w	.0177841	.0005068	35.09	0.000	.0167908	.0187773
_cons	.0011006	.0000438	25.13	0.000	.0010148	.0011865

Figure B.3.: Regression result: U.S. & Canada; bonds as dependent, lagged stocks as independent variable.

Source	SS	df	MS	Number of obs	=	396,518
				F(1, 396516)	=	1056.40
Model	7.86906774	1	7.86906774	Prob > F	=	0.0000
Residual	2953.62393	396,516	.00744894	R-squared	=	0.0027
				Adj R-squared	=	0.0027
Total	2961.493	396,517	.007468767	Root MSE	=	.08631

stock_return_w	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
lagged_bond_return_w	.1504664	.0046294	32.50	0.000	.1413929	.1595398
_cons	.0068372	.0001371	49.88	0.000	.0065686	.0071059

Figure B.4.: Regression result: U.S. & Canada; stocks as dependent, lagged bonds as independent variable.

Source	SS	df	MS	Number of obs	=	23,770
				F(1, 23768)	=	176.60
Model	.198172834	1	.198172834	Prob > F	=	0.0000
Residual	26.6710869	23,768	.001122143	R-squared	=	0.0074
				Adj R-squared	=	0.0073
Total	26.8692598	23,769	.001130433	Root MSE	=	.0335

bond_return_w	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
lagged_stock_return_w	.035411	.0026647	13.29	0.000	.0301881	.0406339
_cons	.0000728	.0002175	0.33	0.738	-.0003535	.0004992

Figure B.5.: Regression result: South America; bonds as dependent, lagged stocks as independent variable.

Source	SS	df	MS	Number of obs	=	23,780
				F(1, 23778)	=	37.46
Model	.247308369	1	.247308369	Prob > F	=	0.0000
Residual	156.97818	23,778	.006601824	R-squared	=	0.0016
				Adj R-squared	=	0.0015
Total	157.225488	23,779	.006611947	Root MSE	=	.08125

stock_return_w	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
lagged_bond_return_w	.0874942	.0142953	6.12	0.000	.0594746	.1155138
_cons	.0040957	.0005269	7.77	0.000	.0030629	.0051285

Figure B.6.: Regression result: South America; stocks as dependent, lagged bonds as independent variable.

Source	SS	df	MS	Number of obs	=	97,752
				F(1, 97750)	=	101.90
Model	.063352847	1	.063352847	Prob > F	=	0.0000
Residual	60.7728379	97,750	.000621717	R-squared	=	0.0010
				Adj R-squared	=	0.0010
Total	60.8361907	97,751	.000622359	Root MSE	=	.02493

bond_return_w	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
lagged_stock_return_w	.0099087	.0009816	10.09	0.000	.0079848	.0118327
_cons	.0004349	.0000798	5.45	0.000	.0002784	.0005913

Figure B.7.: Regression result: Western Europe; bonds as dependent, lagged stocks as independent variable.

Source	SS	df	MS	Number of obs	=	97,806
				F(1, 97804)	=	131.35
Model	.874554341	1	.874554341	Prob > F	=	0.0000
Residual	651.220599	97,804	.006658425	R-squared	=	0.0013
				Adj R-squared	=	0.0013
Total	652.095153	97,805	.006667299	Root MSE	=	.0816

stock_return_w	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
lagged_bond_return_w	.1099202	.0095911	11.46	0.000	.0911217	.1287187
_cons	.003898	.0002609	14.94	0.000	.0033866	.0044094

Figure B.8.: Regression result: Western Europe; stocks as dependent, lagged bonds as independent variable.

Source	SS	df	MS	Number of obs	=	4,688
				F(1, 4686)	=	11.46
Model	.011260426	1	.011260426	Prob > F	=	0.0007
Residual	4.60549804	4,686	.000982821	R-squared	=	0.0024
				Adj R-squared	=	0.0022
Total	4.61675847	4,687	.000985014	Root MSE	=	.03135

bond_return_w	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
lagged_stock_return_w	.0180611	.0053359	3.38	0.001	.0076003	.028522
_cons	-.0003332	.000458	-0.73	0.467	-.0012312	.0005647

Figure B.9.: Regression result: Eastern Europe; bonds as dependent, lagged stocks as independent variable.

Source	SS	df	MS	Number of obs	=	4,689
				F(1, 4687)	=	0.04
Model	.000289666	1	.000289666	Prob > F	=	0.8433
Residual	34.7331534	4,687	.00741053	R-squared	=	0.0000
				Adj R-squared	=	-0.0002
Total	34.7334431	4,688	.007409011	Root MSE	=	.08608

stock_return_w	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
lagged_bond_return_w	.0052053	.0263284	0.20	0.843	-.0464106	.0568213
_cons	.0018778	.0012571	1.49	0.135	-.0005868	.0043424

Figure B.10.: Regression result: Eastern Europe; stocks as dependent, lagged bonds as independent variable.

Source	SS	df	MS	Number of obs	=	28,077
				F(1, 28075)	=	57.18
Model	.005402885	1	.005402885	Prob > F	=	0.0000
Residual	2.65262982	28,075	.000094484	R-squared	=	0.0020
				Adj R-squared	=	0.0020
Total	2.6580327	28,076	.000094673	Root MSE	=	.00972

bond_return_w	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
lagged_stoc~w	-.0036787	.0004865	-7.56	0.000	-.0046322	-.0027252
_cons	.000174	.000058	3.00	0.003	.0000603	.0002877

Figure B.11.: Regression result: China; bonds as dependent, lagged stocks as independent variable.

Source	SS	df	MS	Number of obs	=	28,102
				F(1, 28100)	=	31.77
Model	.467069982	1	.467069982	Prob > F	=	0.0000
Residual	413.087833	28,100	.014700635	R-squared	=	0.0011
				Adj R-squared	=	0.0011
Total	413.554903	28,101	.014716733	Root MSE	=	.12125

stock_retur~w	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
lagged_bond~w	.3796472	.0673531	5.64	0.000	.247632	.5116625
_cons	.0009118	.0007234	1.26	0.207	-.000506	.0023297

Figure B.12.: Regression result: China; stocks as dependent, lagged bonds as independent variable.

B.2. By Credit Rating

For brevity purposes, the lead-lag regression results for bond groups split by credit rating are provided separately, and can be found in the project folder in the **Statistics** part.

B.3. By Aggregated Credit Rating

Source	SS	df	MS	Number of obs	=	439,381
				F(1, 439379)	=	201.87
Model	.086184678	1	.086184678	Prob > F	=	0.0000
Residual	187.585542	439,379	.000426933	R-squared	=	0.0005
				Adj R-squared	=	0.0005
Total	187.671727	439,380	.000427129	Root MSE	=	.02066

bond_return_w	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
lagged_stock_return_w	.0064125	.0004513	14.21	0.000	.0055279	.0072971
_cons	.0013091	.0000314	41.75	0.000	.0012477	.0013706

Figure B.13.: Regression result: Investment Grade Bonds; bonds as dependent, lagged stocks as independent variable.

Source	SS	df	MS	Number of obs	=	439,513
				F(1, 439511)	=	1701.85
Model	8.11977769	1	8.11977769	Prob > F	=	0.0000
Residual	2096.96766	439,511	.004771138	R-squared	=	0.0039
				Adj R-squared	=	0.0039
Total	2105.08744	439,512	.004789602	Root MSE	=	.06907

stock_return_w	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
lagged_bond_return_w	.195274	.0047335	41.25	0.000	.1859965	.2045515
_cons	.0081583	.0001042	78.26	0.000	.007954	.0083626

Figure B.14.: Regression result: Investment Grade Bonds; stocks as dependent, lagged bonds as independent variable.

Source	SS	df	MS	Number of obs	=	112,802
				F(1, 112800)	=	392.57
Model	.391430348	1	.391430348	Prob > F	=	0.0000
Residual	112.47203	112,800	.000997092	R-squared	=	0.0035
				Adj R-squared	=	0.0035
Total	112.86346	112,801	.001000554	Root MSE	=	.03158

bond_return_w	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
lagged_stock_return_w	.0172893	.0008726	19.81	0.000	.015579	.0189996
_cons	.0005757	.0000941	6.12	0.000	.0003913	.0007601

Figure B.15.: Regression result: Non-Investment Grade Bonds; bonds as dependent, lagged stocks as independent variable.

Source	SS	df	MS	Number of obs	=	112,892
				F(1, 112890)	=	71.66
Model	.833779461	1	.833779461	Prob > F	=	0.0000
Residual	1313.53826	112,890	.011635559	R-squared	=	0.0006
				Adj R-squared	=	0.0006
Total	1314.37204	112,891	.011642842	Root MSE	=	.10787

stock_return_w	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
lagged_bond_return_w	.0800967	.009462	8.47	0.000	.0615513	.0986421
_cons	.0046224	.000321	14.40	0.000	.0039932	.0052516

Figure B.16.: Regression result: Non-Investment Grade Bonds; stocks as dependent, lagged bonds as independent variable.

B.4. High Yield

Source	SS	df	MS	Number of obs	=	2,120
				F(1, 2118)	=	89.30
Model	.23193612	1	.23193612	Prob > F	=	0.0000
Residual	5.50074596	2,118	.002597142	R-squared	=	0.0405
				Adj R-squared	=	0.0400
Total	5.73268208	2,119	.002705371	Root MSE	=	.05096

bond_return_w	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
lagged_stock_return_w	.0583903	.0061788	9.45	0.000	.0462732	.0705075
_cons	-.0013249	.0011069	-1.20	0.231	-.0034957	.0008459

Figure B.17.: Regression result: High Yield Bonds by Moody's (Ca & C); bonds as dependent, lagged stocks as independent variable.

Source	SS	df	MS	Number of obs	=	2,124
				F(1, 2122)	=	11.43
Model	.3648643	1	.3648643	Prob > F	=	0.0007
Residual	67.7542046	2,122	.031929408	R-squared	=	0.0054
				Adj R-squared	=	0.0049
Total	68.1190689	2,123	.032086231	Root MSE	=	.17869

stock_return_w	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
lagged_bond_return_w	.2345471	.0693841	3.38	0.001	.0984792	.3706149
_cons	-.0012946	.0038796	-0.33	0.739	-.0089028	.0063136

Figure B.18.: Regression result: High Yield Bonds by Moody's (Ca & C); stocks as dependent, lagged bonds as independent variable.

Source	SS	df	MS	Number of obs	=	9,295
				F(1, 9293)	=	150.69
Model	.289455534	1	.289455534	Prob > F	=	0.0000
Residual	17.8505759	9,293	.001920863	R-squared	=	0.0160
				Adj R-squared	=	0.0159
Total	18.1400315	9,294	.0019518	Root MSE	=	.04383

bond_return_w	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
lagged_stock_return_w	.0393445	.0032051	12.28	0.000	.0330618	.0456272
_cons	-.0008916	.0004546	-1.96	0.050	-.0017827	-4.93e-07

Figure B.19.: Regression result: High Yield Bonds by S&P (CCC); bonds as dependent, lagged stocks as independent variable.

Source	SS	df	MS	Number of obs	=	9,297
				F(1, 9295)	=	27.92
Model	.558162495	1	.558162495	Prob > F	=	0.0000
Residual	185.813496	9,295	.019990694	R-squared	=	0.0030
				Adj R-squared	=	0.0029
Total	186.371659	9,296	.020048586	Root MSE	=	.14139

stock_return_w	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
lagged_bond_return_w	.1637786	.0309949	5.28	0.000	.1030217	.2245355
_cons	.0005872	.0014668	0.40	0.689	-.002288	.0034624

Figure B.20.: Regression result: High Yield Bonds by S&P (CCC); stocks as dependent, lagged bonds as independent variable.

Bibliography

[1] Francisco Javier Carrera Arias. *Fuzzy String Matching in Python*. https://www.datacamp.com/community/tutorials/fuzzy-string-python. Datacamp Platform, Last accessed Mar. 31, 2021.

[2] James Chean. *Stock Exchange Daily Official List (SEDOL)*. https://www.investopedia.com/terms/s/sedol.asp, November 2020. Last accessed Mar. 31, 2021.

[3] James Chen. *International Securities Identification Number (ISIN)*. https://www.investopedia.com/terms/i/isin.asp, July 2019. Last accessed Mar. 31, 2021.

[4] James Chen and Julius Mansa. *CUSIP Number*. https://www.investopedia.com/terms/c/cusipnumber.asp, November 2020. Last accessed Mar. 31, 2021.

[5] Cuelogic Software Consultancy. *The Levenshtein Algorithm*. https://www.cuelogic.com/blog/the-levenshtein-algorithm#:~:text=The%20Levenshtein%20distance%20is%20a,one%20word%20into%20the%20other., January 2017. Last accessed Mar. 31, 2021.

[6] WM Datenservice. *Struktur und Aufbau der deutschen Wertpapier-Kenn-Nummer (WKN)*. https://www.wmdaten.de/pdf_wkn_isin/Vergaberegeln_WKN.pdf, July 2012. Last accessed Mar. 31, 2021.

[7] Edwin J. et al Elton. *Modern Portfolio Theory and Investment Analysis*. Wiley Custom, April 2017. 9th edition.

[8] Benedikt Franke, Sebastian Müller, and Sonja Müller. *The q-factors and expected bond returns*. Journal of Bankind and Finance, June 2017.

[9] Catherine Gitau. *Fuzzy String Matching*. https://towardsdatascience.com/fuzzy-string-matching-in-python-68f240d910fe. Towards Data Science Platform, Last accessed Mar. 31, 2021.

[10] Parameter Description in Refinitiv Datastream Navigator. *Worldscope Identifier*, 2021. Last accessed Mar. 31, 2021.

[11] Konstantinos Tolikas. *The lead-lag relation between the stock and the bond markets*. https://pure.aston.ac.uk/ws/files/23811392/The_lead_lag_relation_between_the_stock_and_the_bond_markets.pdf, May 2017. Last accessed Mar. 31, 2021.

[12] Oscar Torres-Reyna. *Linear Regression using Stata*. https://www.princeton.edu/ otorres/Regression101.pdf, 2007. Last accessed Mar. 31, 2021.