KMS ABGJ

Bulk Analysis of Mortgage Data with Cluster Computing

A Major Qualifying Project (MQP)
Submitted to the Faculty of

WORCESTER POLYTECHNIC INSTITUTE

In fulfillment of the requirement for the
Degree of Bachelor of Science

Submitted by:
Evan King
Thyagarajan Ramachandran

January 11th, 2018

Submitted to:
Project Advisors:
Professor Kevin Sweeney, Worcester Polytechnic Institute
Professor Michael Ciaraldi, Worcester Polytechnic Institute
Professor Renata Konrad, Worcester Polytechnic Institute

Project Liaisons
Scott Burton, Angelo, Gordon, & Co. Managing Director

Abstract

Angelo, Gordon & Co. is developing a statistical model of loan delinquency status. This project designed and implemented a software package to process a public data set from Wells Fargo to build a rudimentary model. The data requires significant manipulation to convert it into a useful form. A series of compartmentalized modules were created, each of which are combined to form a “Tech Stack”, which runs each step in the sequence. This ends in an upload to a cloud storage provider. Once the Tech Stack had processed the relevant data, several sample analyses were run to demonstrate the data’s capabilities. The size of the data set made computations with a single computer impractical, so a cluster was used to analyze the data.

Acknowledgements

We would like to thank Angelo, Gordon & Co. for their consistent support and commend them on their excellent hospitality. Thanks to Scott Burton, who was always available despite his demanding schedule. We would like to thank Xinkai Wu, his mentorship throughout was invaluable. His presence in the project was that of a teammate. We would also like to thank Cindy Aguilar for her patience with all administrative issues and facilitating us throughout. Special thanks to Oren Efrati, constantly solving our issues and going out of his way to do so.

We would like to thank the Director of the Wall Street project center Kevin Sweeney and our advisors Michael Ciaraldi and Renata Konrad for the continuous care and devotion throughout the process from applying to the MQP till submission. We would also like to thank WPI for providing us with this opportunity. Finally, a special thanks to Jacob Aki for taking time out of his schedule to coordinate with everyone to ensure a great MQP experience overall.

Authorship

Section	Author
1. Introduction	Evan & Thyagarajan
1.1. Objectives	
2. Background	
2.1. Angelo, Gordon & Co.	Thyagarajan
2.2. Mortgage-Backed Securities	Thyagarajan
2.3. Wells Fargo CTSLink	Evan
2.4. Cluster Computing and Spark	Thyagarajan
2.4.1. Spark	
2.5. Technologies Used	Evan
2.5.1. Python	
2.5.2. PARQ format	
2.5.3. Azure Data Lake	
2.5.4. Visual Studio Team Services	
2.5.5. Azure DataBricks	
2.5.6. HDInsight	
2.5.7. Selenium & Geckodriver	
3. Methodology	
3.1. Project Planning & Analysis	Thyagarajan
3.2. Design	Evan
3.2.1. Object Oriented versus Imperative	

	5.7. Schema Scanner	
	5.8. Azure Datalake Upload	
	5.9. Ingestion	
	5.10. Miscellaneous tools	
	5.10.1. Installer	
	5.10.2. DocGenerator	
6.	Data Wrangling and Analysis	Thyagarajan
	6.1. Data Wrangling	
	6.1.1. Duplicate Loan Numbers	
	6.1.2. Checking Integrity of Original Balance Values	
	6.1.3. Delinquency Status of Loans	
	6.2. Data Processing	
	6.2.1. Delinquency Status Transition Matrix	
	6.2.2. Pay Off Rate in Different States of USA	
	6.2.3. Foreclosure in Various States	
7.	Conclusions	Thyagarajan
8.	Recommendations	Thyagarajan
9.	Reflection on the Project	Thyagarajan
	9.1. Design Component	
	9.2. Constraints and Alternatives	
	9.3. Need for Life-Long Learning	

Table of Contents

List of Figures

List of Tables

1. Introduction

Angelo, Gordon & Co. is an investment management company. Like many firms, Angelo Gordon is looking for quantitative data insights to inform their investment decisions. Specifically, Angelo Gordon seeks to build a statistical model of the probability for loans to default or be paid in full. Although a third party vendor provides necessary reports to make decisions, the existing data sets are paid and narrow. There is no freedom to perform personal analysis on the mortgage data as the data is not owned by the firm.

Because of federal banking regulations, there is a great deal of public data available on the Internet. The data is distributed across many files, in a proprietary format, and likely contains errors. The firm requires a software package to download, process and upload the relevant data. The data once uploaded also needs to be checked for quality and consistency. Additionally, sample analysis on the data will give the Risk Management Team of Angelo Gordon a head start towards building the necessary statistical model.

1.1. Objectives

There were three essential tasks that needed to be completed. First, we had to acquire the data. This needed to be done in an automated way, as there was too much data to download it all manually. Once this was complete, we needed to process the data to prepare it for analysis. When the data was uploaded, the data had to be cleansed for proper formatting and field values. Once the data was ready, we needed to prove that our processing was successful by demonstrating sample analyses. This also allowed the firm to have a basic foundation to build the necessary statistical model upon.

In addition to the practical objectives of the project, we also had to keep in mind the long-term stability of whatever tool we chose to build. The data we harvested was not static— every month, new data points are added. This means that the tool needs to be rerun every month. As a direct byproduct of this, ease of use was made a priority. If the tool was too hard to install or use, the newest dataset would not be incorporated with the larger data set. Other than delivering the tool itself, proper documentation such as readme files were also provided.

2. Background

2.1. Angelo, Gordon & Co.

Angelo, Gordon & Co. (AG) is a privately held, registered alternative investment advisor founded in 1988. Headquartered in New York City, NY, it has approximately $28 billion in assets (Angelo, Gordon & Co.) under management around the globe. Although relatively small compared to banks, AG has over 360 employees which is large relative to most hedge funds. The firm is also less regulated compared to investment banks as it manages assets for high net worth individuals and acts as an advisor (Staff, 2017).

The company is dedicated to alternative investing, capitalizing on various market situations in the world. Angelo Gordon is invested in distressed debt, real estate, private equity, and some multi-strategy investments. Angelo Gordon remains a leader in alternative investments by exploiting market inefficiencies and seeking to capitalize on new investments that are not mainstream. For such a firm, information about current market trends is invaluable if it is to exploit these situations. The project serves the same purpose as the ultimate goal is to use the data set to obtain information about the real estate market.

2.2. Mortgage-Backed Securities

Mortgage-backed securities (MBS) are bonds that are secured upon home and other real estate loans. A group of loans with similar attributes such as credit score are pooled together. A fixed amount worth of such mortgages is sold to a government agency such as Ginnie Mae or a government sponsored-enterprise (GSE) such as Fannie Mae. The majority is sold as such and the MBS carries the guarantee of the issuer. Unlike traditional bonds, since most home owners pay mortgages monthly, MBS bondholders receive monthly interest payments (FINRA, 2018).

MBS played a central role in the 2008 financial crisis which wiped out trillions of dollars and brought down Lehman Brothers. Mortgage backed securities allow a bank to move its mortgages off the books by turning them into securities and selling them to investors such as Angelo Gordon. Many loans are bundled together, which distributes the default risk across many loans. This is compared to selling loans individually, which is referred to as whole loan. This flexibility for banks gave room for more lending capital and encouraged banks to reach further down in credit worth to supply more investors. GSE and other agencies supported this aggressive sale of MBS and when poor performing loans began to default, the quality of the MBS started to decline. Loans defaulted at an unacceptable rate, which meant that the otherwise safe investment in MBS was now unstable. These non-payments led to the MBS market being over valued and eventually drying up. MBS are still bought and sold today but the Federal Reserve still holds onto the MBS purchased for $1.75 trillion in 2008. (Beattie, 2018)

2.3. Wells Fargo CTSLink

Wells Fargo CTSLink is a website provided by Wells Fargo. It contains publicly available mortgage data on deals from 2000 to present. Data is divided across a multitude of shelves, largely by what department, subsidiary or organization issued the loans.

There are other mortgage data sources available, most notably from Fannie Mae (Fannie Mae n.d.) and Intex (Intex n.d.). Intex provides models based on data they have collected, rather than the raw data itself. However, those models are very expensive. Fannie Mae's data is a limited subset of all their loans, while Wells Fargo is a complete set of roughly 8,000,000 loans. For those reasons, the Wells Fargo data set was selected for analysis. Additionally, maintaining a single data source means that inconsistencies between data sets are kept to a minimum.

2.4. Cluster Computing and Spark

A cluster refers to a group of computers and other resources that are connected through hardware, networks and software to behave as a single system. The incentive to use clusters is to improve performance and availability over that of using a single computer but still maintaining the cost-effectiveness of having a single computer with the same performance or availability. The aim of cluster computing has been to make the power of many computers from the "outside world" or internal user departments available as if they were one and always working fast.

The desire for this came from using a series of low-cost off-the-shelf computers to do high performance tasks. This was preferred by smaller firms as compared to purchasing high performing computers from the market. Adding a layer of software over these nodes (computers) has allowed firms to do computationally intensive tasks without the necessary costly machinery. Cluster computing is used because clusters provide high availability, load balancing, parallel processing, systems management and scalability (IBM 2002).

2.4.1. Spark

Apache Spark is an open-source cluster-computing framework. While there are a variety of different APIs, we are using PySpark, which is a Python frontend for the Spark framework (Zaharia, 2010). The core of Spark is centered on resilient distributed datasets (RDD). By passing a function to Spark, one can invoke parallel operations such as filter or join. RDD operations are lazy evaluated and immutable. Jupyter Notebook is an application that allows these operations and allows one to edit, run and share the Python code.

2.5. Technologies Used

2.5.1. Python

Python is a popular programming language that can be used to solve a variety of problems. It is a loosely typed language executed from compiled bytecode. This means that the language is exceptionally flexible. While this can pose problems for developing very large applications, it makes development of smaller applications much faster.

2.5.2. PARQ format

PARQ is a compression format useful for storing dataframes. A dataframe is a data set organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python. Our Spark software uses PARQ files as data sources. We use the Python package fastparquet to serialize data into the PARQ format. Spark is capable of reading PARQ files natively which reduces additional and unnecessary pre-processing.

2.5.3. Azure Data Lake

Azure Data Lake is a cloud storage platform offered by Microsoft. It can be accessed via a variety of different APIs, which allows us to integrate uploads and downloads directly into our Python tech stack.

2.5.4. Visual Studio Team Services

Version control is handled as a Visual Studio Team Services (VSTS) backed Git repository. Strong version control software is essential to the success of a project.

2.5.5. Azure DataBricks

Azure DataBricks is a Microsoft cloud product that provides access to a Spark/PySpark cluster owned by Angelo Gordon. DataBricks is used to create isolated cluster instances hosted in the cloud. It is managed and run slightly different from HDInsight, but there are otherwise limited differences between the two. While both are offered on the Azure platform, DataBricks is different from Azure Data Lake.

2.5.6. HDInsight

HDInsight is another cluster computing client similar to Azure DataBricks. Both clients are used interchangeably. There are some inconsistencies in Spark versioning between HDInsight and DataBricks. Most notably, HDInsight supports large integer types which means it can store native Python DateTime objects.

2.5.7. Selenium & Geckodriver

In order to access CTSLink, we used a popular Python package called Selenium. Selenium is mostly used for UI test automation. In this case, we use it to emulate a user on CTSLink's website.

Selenium provides an API to emulate browser interaction, but modern browsers are extremely complicated. Rather than create their own browser, Selenium requires a driver which links it with an installed version of a particular browser. We use Mozilla's Firefox driver, Geckodriver, for this purpose.

It is important to note that although Selenium is part of the tool provided, all initial download was done manually to ensure accuracy.

3. Methodology

3.1. Project Planning & Analysis

When this project was first initiated, the initial thought was that this project will follow three separate and distinct phases. Hence, a work breakdown structure (WBS) was created to display all the then known defined activities necessary to accomplish this project. A WBS is a tool used to plan projects as it not only defines a structure for necessary tasks but it also provides great clarity while creating a project schedule.

A WBS allows the team to have a clear division of labor and create time estimates on how long each phase or task may take. This is also why it is important to create a final process map to include any changes that have occurred in the WBS and include greater detail for each step within the map. This can be done through key symbols for each task/decision and with clearer titles. Including streams where each set of tasks are classified within also gives clarity to the process map. Once the business process tool to download all the data and upload them on Azure Data Lake was achieved, a second process map was created to understand the new and developed process at a whole level.

The exercise of creating a process map allows one to understand the limitations and achievements of the project. The process map allows one to highlight what tasks need more attention than others and allows handing the process over to a different stake holder to be much clearer. This is important as the process needs to be reproducible and owned by the sponsor even if the team is not present at the site. This is because new data sets are posted every month, so the software package must be run every month. If there is not ownership and understanding of the process, the data will become outdated.

3.2. Design

Simplicity and ease of use are at the core of the business process tool. We opted not to add any features that were not requested by management. Each responsibility in the tech stack is split into a module and there are no circular dependencies.

3.2.1. Object Oriented versus Imperative

Python does support the creation of classes, but we decided not to use classes in the tech stack. There are several reasons for this. Classes and objects serve as a way of associating functionality and data, but the data is loaded externally and none of the modules require any state. Also, the division of modules helps isolate methods from each other. Creating a class around a single function simply adds extraneous scaffolding

Classes and objects are useful when the path of execution is dynamic and abstract. This is not the case. We have a very well defined process, with steps which are explicitly separate from each other. The order does not change from execution to execution, and each step does not need to relate to any other step. Furthermore, there are no fields or properties needed in any of the modules. Once they finish executing their step, they resume the next module.

Repeated code was refactored into functions, but ultimately there was very little repeated code. The only compelling example for deduplication was in the ask_token method for datalake_upload because it is used in the main function for datalake_upload and ingestion.

There are no dependencies between any of the modules; ingestion serves as a “master” script that runs each module in sequence, but each module does not depend on each other module.

3.2.2. YAGNI

The common programming axiom "You Ain't Gonna Need It" held true here. We planned for the future to the extent that there is little to no coupling between modules. Adding new modules or modifying existing ones will not result in changes throughout the codebase. We did not speculate about what may be needed in the future, and only planned for three certainties:

1. The requirements for the software will change over time as the technology it is built on does.
2. Processing phases may be added. Therefore, there can be no dependency between phases.
3. Phases may have to be modified. There are no dependencies between phases.

3.2.3. Usability vs flexibility

Usability was the single most important goal for the creation of this business tool. The tool is only useful if it is run every month. If installation and execution were confusing due to the number of features, it would be less likely that an average user would be able to build and run it.

We chose to embed the geckodriver binary within the Git repository directly to speed up installation. Typically, build artifacts and required packages are installed by the user; however, installing geckodriver is nontrivial.

Dependencies on projects can be installed with an installer script. This means that if external packages are added, the installer package must be updated. Angelo Gordon's primary interest was making sure that the application can be rapidly deployed on a new machine.

3.2.4. Automation

Automation was also important, as it is peripherally related to usability. The user does not need to interact with the script once it is running. The Selenium script eliminates a labor-intensive step of the process. The only time the script will stop during execution is in the event of an irreparable failure. In all other cases, the exception is logged and execution continues.

3.3. Data Wrangling

Data wrangling or data munging is the process of transforming data into a more desirable and valuable format to be able to leverage it downstream. This is essential for any downstream process such as data visualization, data analytics, training a statistical model or even further wrangling. This generally follows steps such as extracting the data from whichever data source in question, sorting or filtering the data set for the necessary data, parsing it into the necessary data structures, depositing the resulting data into a data sink for storage, and finally wrangling the data for field values.

The data transformations are performed on specific entities (e.g. fields, rows, columns, data values, etc.) within a data set. These transformations can be extractions, parsing, joining, standardizing, cleansing, and even creating new necessary data. These actions increase the quality of the data set by ensuring their reliability, integrity, and reducing the need to do preprocessing before being used for any downstream purpose.

The data set from Wells Fargo is open source and can only be downloaded through the CTSLink website. Each issuer of a report deposits their data in Wells Fargo’s website in their respective shelves. Since there are a variety of shelves, this already creates problems for standardization of the data and the need to filter for the exact data amongst other files that may have been uploaded by the issuer. Another problem that requires filtering is file names across different issuers have a different naming convention. Our tool had to accommodate such irregularities and correct them before uploading them to the data lake.

4. Process and Analysis

4.1. Work Breakdown Structure

As mentioned previously, a WBS was made to have a greater clarity over this project. Figure 1 illustrates how the WBS for this project as perceived at the beginning of this project.

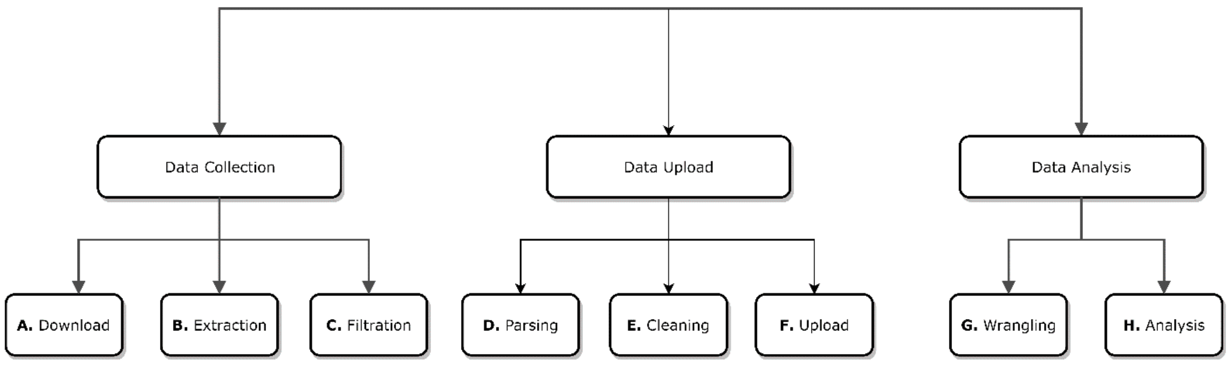


Figure 1 Work Breakdown Structure of the Project

The tasks are grouped in three bigger categories that were defined by the sponsor at the beginning of this project. The “Data Collection” stage has three separate tasks that involve acquiring the data set. “Data Upload” involves the tasks necessary to upload the collected data set into Azure Data Lake, a service the sponsor is moving towards to store all data. Finally, “Data Analysis” involves the tasks of transforming the data into a workable data set that has the highest level of integrity to perform analysis and derive reliable findings. This is very important because if the sponsor is to use this data to make investment decisions or assess risk, the data quality must be at its highest.

At the third level, the specific tasks are outlined in order to complete this project. The lettering before the task names indicates the sequence of tasks which is crucial for project scheduling. These tasks must be completed in a linear fashion. This is definitely a drawback as no two tasks can be finished simultaneously to save time. Below is a detailed explanation of each task in chronological order:

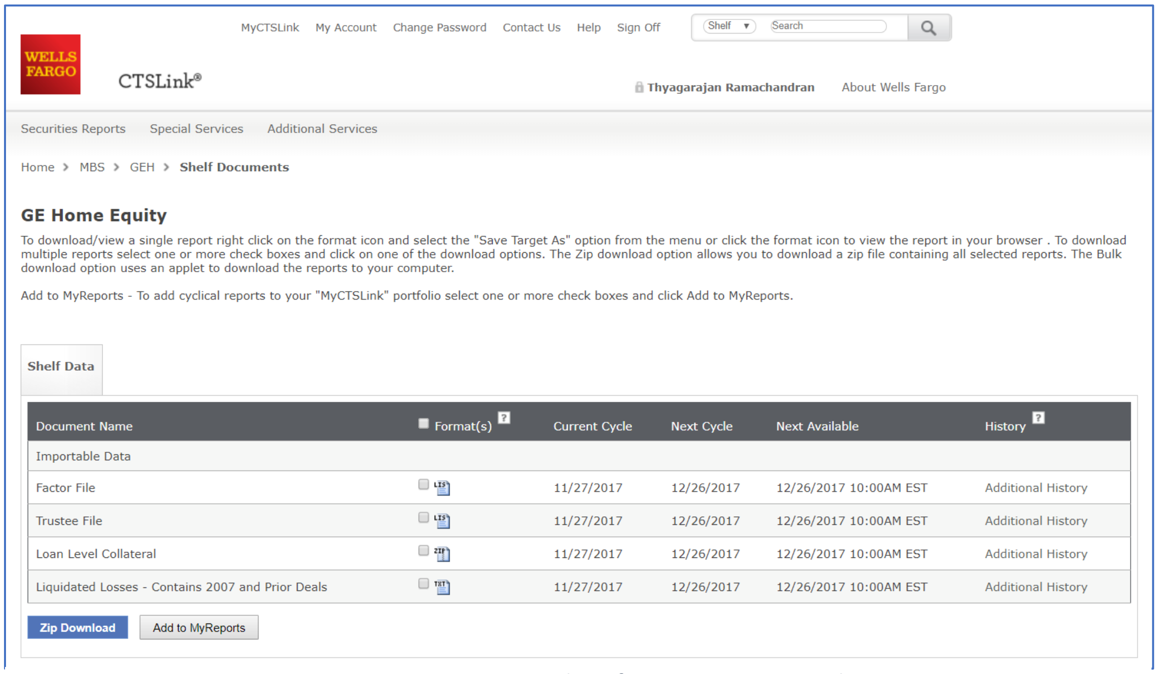
4.1.1. Download

This task is the first and foremost in the process of gaining insight from the data. Wells Fargo has uploaded data on their website called "Corporate Trust Services" (CTSLink) on a whole loan level. After the 2008 crisis, Wells Fargo released large amounts of redacted mortgage data. All the personal data regarding the loan e.g. addresses, names, contact information, etc. has already been removed from the data set. The website does not have any API and therefore the download must be done manually.

In the business process tool provided, we created a browser automation which eliminated the need to manually download the files. Without automation this step would have been labor intensive because there are 186 shelves to download. The tool provided is to collect the data set for each month as new data for the previous month is uploaded every month.

However, at the beginning of this project, all the data had to be acquired manually as automation of this task was completed later in the project. This was also to ensure that the data set for all the previous years is accurate and no shelf is left missing.

There were three stages to download data for each shelf. These steps were done with AutoHotkey (AHK) scripting which is a software package with its own language to assign actions on the screen that are triggered with a certain key stroke. Figure 2 below shows a screenshot of a particular issuer once opened.



WELLS FARGO
CTSLink®
MyCTSLink My Account Change Password Contact Us Help Sign Off
Shelf
Search
Thyagarajan Ramachandran
About Wells Fargo
Securities Reports Special Services Additional Services
Home > MBS > GEH > Shelf Documents
Shelf Data
Zip Download
Add to MyReports

GE Home Equity

To download/view a single report right click on the format icon and select the "Save Target As" option from the menu or click the format icon to view the report in your browser . To download multiple reports select one or more check boxes and click on one of the download options. The Zip download option allows you to download a zip file containing all selected reports. The Bulk download option uses an applet to download the reports to your computer.

Add to MyReports - To add cyclical reports to your "MyCTSLink" portfolio select one or more check boxes and click Add to MyReports.

Document Name	Format(s)	Current Cycle	Next Cycle	Next Available	History
Importable Data					
Factor File	LTS	11/27/2017	12/26/2017	12/26/2017 10:00AM EST	Additional History
Trustee File	LTS	11/27/2017	12/26/2017	12/26/2017 10:00AM EST	Additional History
Loan Level Collateral	ZIP	11/27/2017	12/26/2017	12/26/2017 10:00AM EST	Additional History
Liquidated Losses - Contains 2007 and Prior Deals	TXT	11/27/2017	12/26/2017	12/26/2017 10:00AM EST	Additional History

Figure 2 Screenshot of Issuer Page on CTSLink

The liquidated losses and the loan level collateral are the files that is the data set which is to be downloaded. The first step was to check if a certain issuer/shelf had a data file uploaded. Figure 3 shows the script used that was triggered with “Ctrl+b” and used the find function built-in Chrome. If a data file is detected, then the next step is to download or else the user has to move to the next issuer. Figure 4 shows the code to download the data set which can be done by clicking the “Additional History” tab. The following code is entered in the Chrome console within the hotkey to select all the check boxes (Giesbrecht, 2015):

```
var getInputs = document.getElementsByTagName("input");
for (var i = 0, max = getInputs.length; i < max; i++){ if (getInputs[i].type === 'checkbox')
getInputs[i].checked = true; }
```

The hotkey script downloads all the files and returns to the website. The third step is to go to the main page and start the search for the next available data set. This continues until all the data set has been downloaded to a location in the device.

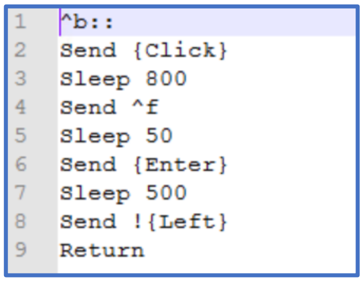

```
^b::
Send {Click}
Sleep 800
Send ^f
Sleep 50
Send {Enter}
Sleep 500
Send !{Left}
Return
```

Figure 4 Ahk Script to Check for Data Files

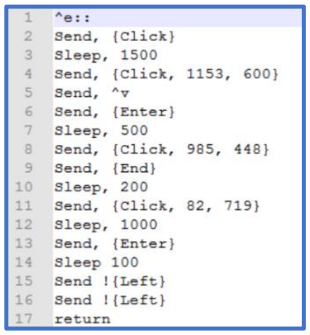
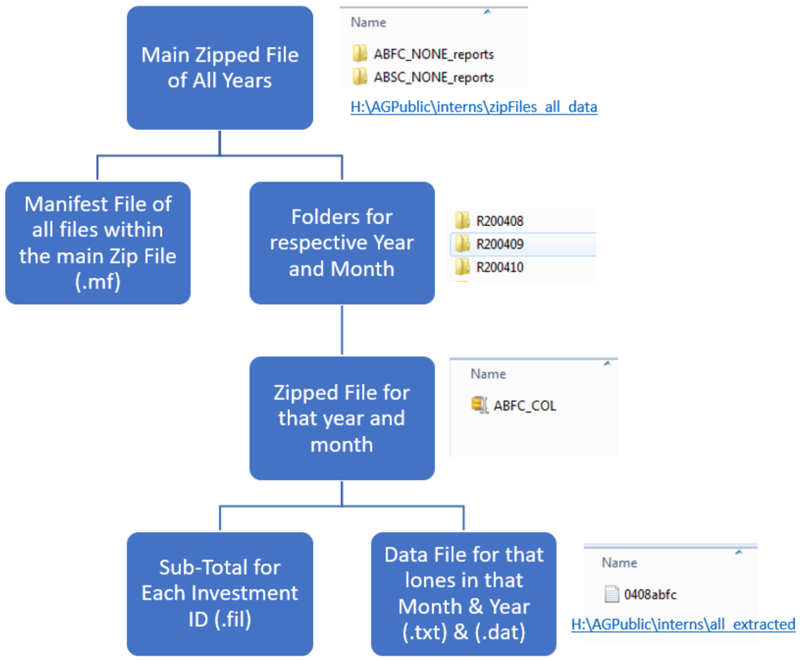

```
^e::
Send, {Click}
Sleep, 1500
Send, {Click, 1153, 600}
Send, ^v
Send, {Enter}
Sleep, 500
Send, {Click, 985, 448}
Send, {End}
Sleep, 200
Send, {Click, 82, 719}
Sleep, 1000
Send, {Enter}
Sleep 100
Send !{Left}
Send !{Left}
return
```

Figure 3 Ahk Script to Click and Download All Files

4.1.2. Extraction

Once the data is collected in one folder, the files need to be extracted. The format they are downloaded in are compressed files with two levels within them. They are zipped as the entire shelf and they are also zipped within the shelf by each month for all the years. Figure 5 below shows the file structure of each download with the data file at the end.



Main Zipped File of All Years
Name
ABFC_NONE_reports
ABSC_NONE_reports
H:\AGPublic\interns\zipFiles_all_data
Manifest File of all files within the main Zip File (.mf)
Folders for respective Year and Month
R200408
R200409
R200410
Zipped File for that year and month
Name
ABFC_COL
Sub-Total for Each Investment ID (.fil)
Data File for that lones in that Month & Year (.txt) & (.dat)
Name
0408abfc
H:\AGPublic\interns\all_extracted

Figure 5 File Structure from Downloading Each Shelf File

The aim of the extraction step is to convert all the files from a compressed folder to their native format. This was achieved through a Python script that has been included in the business process tool and is in Appendix 10.3.

4.1.3. Filtration

There are other files that exist within the folders that are not contained within the data set itself. Shelves often contain extraneous Excel, PDF and LIS (manifest) files. All these files are extraneous and require removal. A Python script allows the detection of .txt & .dat files and delete anything else that is extraneous. These files are later binned by shelf folder names once they have been filtered. In this process, if there are data files that do not follow the same naming convention as others, they are filed under a separate folder than the others. Encoded within the name of the file is the shelf name, month and year.

4.1.4. Cleaning

Once all the data files have been collected in one folder, a quick visual inspection is done to ensure all the data is relevant and the folder contains only data files. Files with erroneous file names must be corrected manually by inspecting the data. Opening the data file hints as to how to name the file correctly. The file itself has a column of remittance cycle that indicates the date. This can also be cross referenced with the next due date column. As for most loans, the due date is a month from the current cycle since most loans are in good standing. Once these checks are performed, the files can enter the folder they would have according to the previous task.

4.1.5. Parsing

This step compresses the data for each shelf in parquet format which is most compatible and favored with cluster computing. This process assigns fields (columns) according to Wells Fargo's positional layout schema. The Python script with the fields are given in Appendix 10.4.

Once fields are assigned, they are converted to .parq format and compressed by each shelf file. This allows one .parq file for each shelf and reduces the total file size of all data from 500GB to 30GB. This definitely makes the next task of uploading much faster.

4.1.6. Upload

In this step, all the data using a script is uploaded into Microsoft Azure Data lake which is the software used for data storage by the sponsor.

4.1.7. Wrangling

Using Spark, the field values are transformed to a more desirable format. New necessary columns are added to conduct analysis and the data set is verified for variety of issues. The chapter of Data Wrangling and Analysis will explain this task in detail.

4.1.8. Analysis

The data is then analyzed using Spark and Excel. Spark allows for the necessary processing of data and it also allows the export of the results. Excel acts as a great graphical analysis software for our purposes but the sponsor can use other powerful Data Visualization BI Tools such as Tableau or Power BI.

4.2. Process Flow Diagram

After the project was completed, a detailed process flow diagram was necessary in order to map the final process. This helps in evaluating the project’s success and understanding the process. The business process tool also follows the same process. Hence mapping the process flow will also result in mapping the business process tool which can be quite useful when the software package needs maintenance. The map also ensures transparency amongst the various stakeholders.

A swim lane diagram was created and is shown in Figure 6. The first line indicates who performed which particular task. The same headings from the work break down structure is used for tasks. Each lane (column) represents a single task and the smaller tasks that constitute the main task are shown in that particular column.

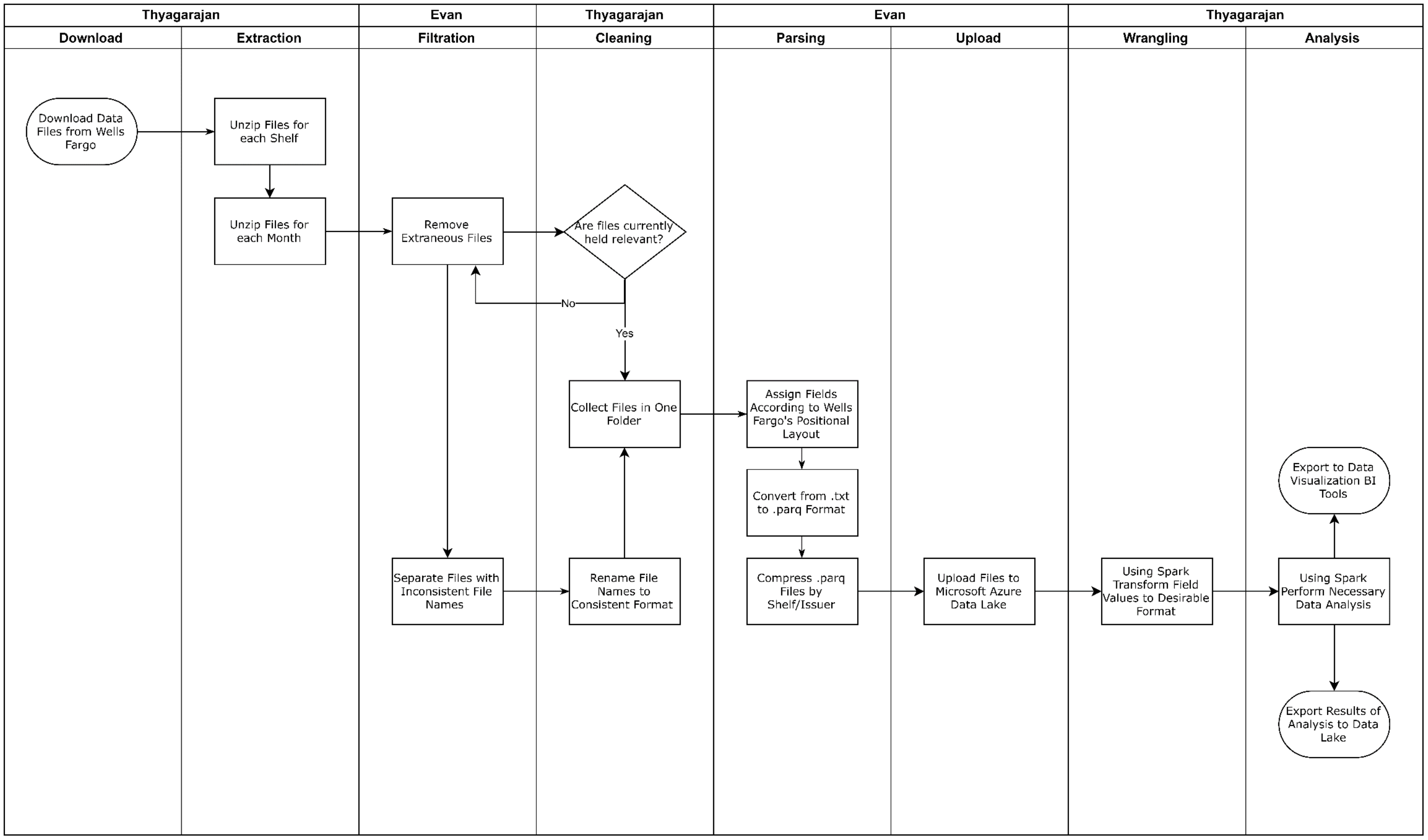

Thyagarajan
Evan
Thyagarajan
Evan
Thyagarajan
Download
Extraction
Filtration
Cleaning
Parsing
Upload
Wrangling
Analysis
Download Data Files from Wells Fargo
Unzip Files for each Shelf
Unzip Files for each Month
Remove Extraneous Files
Are files currently held relevant?
No
Yes
Collect Files in One Folder
Assign Fields According to Wells Fargo's Positional Layout
Convert from .txt to .parq Format
Separate Files with Inconsistent File Names
Rename File Names to Consistent Format
Compress .parq Files by Shelf/Issuer
Upload Files to Microsoft Azure Data Lake
Using Spark Transform Field Values to Desirable Format
Using Spark Perform Necessary Data Analysis
Export to Data Visualization BI Tools
Export Results of Analysis to Data Lake

Figure 6 Swim Lane Diagram

5. Implementation

The “Tech Stack” represents all of the technology used to download data from CTSLink all the way to processing it on Spark. Each module is independent, but when linked together, they form a pipeline that performs all the necessary actions to get data ready for analysis. This section details each module. It describes the key features, as well as explains some of the design choices.

5.1. Automated Downloader

The automated downloader script was created as an optimization of the Download step in the process flow diagram. It automates browser interactions to the Wells Fargo site CTSLink.

Wells Fargo CTSLink is a website that appears to be built-in ASP.NET. It is the data source from which all data is gathered. It shows data on MBS securities Wells Fargo and its subsidiaries own.

CTSLink presents some issues. First, the site itself has an incorrectly signed SSL certificate, which means that navigating to ctslink.com directly results in a security warning. Not all of the shelves available on the website contain the necessary data files. Additionally, CTSLink has poor session handling, which can result in random logoffs and disconnections.

Data on CTSLink is divided into shelves based on the loan issuer. While Wells Fargo’s WFMBS shelf is the largest shelf, the bulk of the loans appear to be from other subsidiaries and departments.

Manual capture of the data in CTSLink is time consuming as it requires clicking through many menus manually. There is no API for downloading files— everything must be done via user interaction.

To solve that problem, we created a Selenium script to download shelf documents. Selenium is a package typically used to automate GUI testing, but a companion package Splinter adds some useful automation tools. The script searches for HTML elements on the page, then clicks them.

Selenium uses geckodriver to emulate operations in Firefox. In order to maximize the ease of use, we embed this directly into the Git repository. While it is unusual to embed build artifacts into the repository, we prioritized ease of use over convention. Downloading geckodriver can be confusing and ensuring it is properly loaded into your PATH environment variable is difficult. To that end, we opted to include it into the Git repository directly.

There are several UI features in Firefox that are useful to humans but not to software, such as file confirmation dialogs. To ensure consistency as the script is run, we load a custom Firefox profile into the Selenium instance:

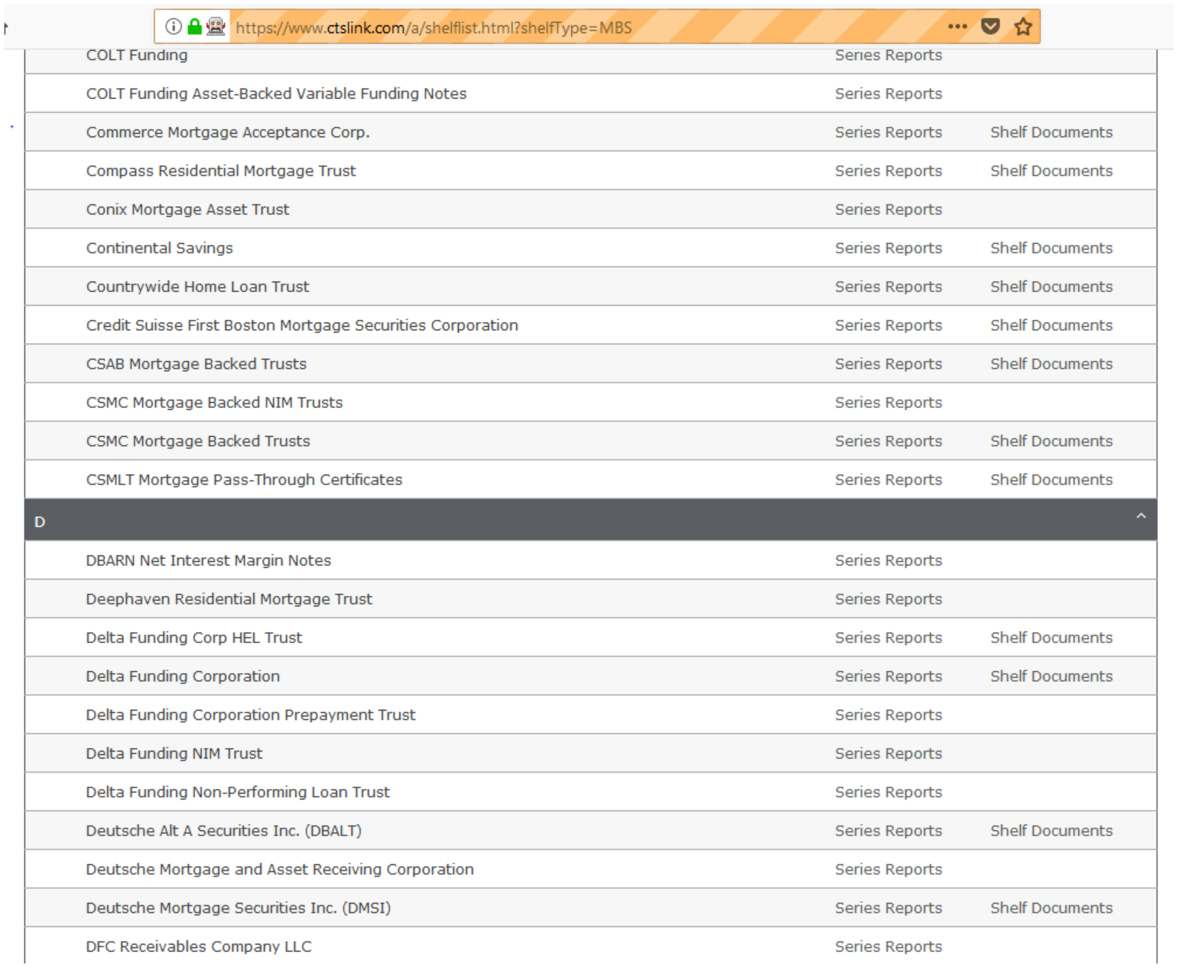
```
    prof = {}
prof['browser.download.manager.showWhenStarting'] = 'false'
prof['browser.helperApps.alwaysAsk.force'] = 'false'
prof['browser.download.dir'] = output_path
prof['browser.download.folderList'] = 2
prof['browser.helperApps.neverAsk.saveToDisk'] = 'application/zip'
prof['browser.download.manager.useWindow'] = 'false'
prof['browser.helperApps.useWindow'] = 'false'
prof['browser.helperApps.showAlertonComplete'] = 'false'
prof['browser.helperApps.alertOnEXEOpen'] = 'false'
prof['browser.download.manager.focusWhenStarting'] = 'false'

browser = Browser('firefox', profile_preferences=prof)
```

Selenium allows interaction with elements only if the DOM was generated at the time of interaction. That means that if a list of links is enumerated, then only one can be used before the DOM is considered stale. For example, the following code will fail:

```
shelves = browser.find_by_text("Shelf Documents")
for shelf in shelves:
	shelf.click()
```

This code fails because after the shelf has been clicked, the page we are currently on no longer contains a Shelf Documents link. Even though the URL is still accessible, that element is no longer on the page, and as a result the entire shelves list is invalidated. In fact, even if we navigate back before clicking on the next one, it will still fail, because there is no way for Selenium to know for sure that the element still exists.

To fix that, we download shelves recursively. We still use an iterator, but skip shelves that have already been downloaded, and only download one shelf per function instance. When all the shelves are complete, the recursion will complete. It is non-intuitive to iterate over a list as well as operate recursively, but because we skip shelves that we've already processed, each shelf is only downloaded once. This guarantees that the DOM is not stale.



https://www.ctslink.com/a/shelflist.html?shelfType=MBS

COLT Funding	Series Reports	
COLT Funding Asset-Backed Variable Funding Notes	Series Reports	
Commerce Mortgage Acceptance Corp.	Series Reports	Shelf Documents
Compass Residential Mortgage Trust	Series Reports	Shelf Documents
Conix Mortgage Asset Trust	Series Reports	
Continental Savings	Series Reports	Shelf Documents
Countrywide Home Loan Trust	Series Reports	Shelf Documents
Credit Suisse First Boston Mortgage Securities Corporation	Series Reports	Shelf Documents
CSAB Mortgage Backed Trusts	Series Reports	Shelf Documents
CSMC Mortgage Backed NIM Trusts	Series Reports	
CSMC Mortgage Backed Trusts	Series Reports	Shelf Documents
CSMLT Mortgage Pass-Through Certificates	Series Reports	Shelf Documents
D		
DBARN Net Interest Margin Notes	Series Reports	
Deephaven Residential Mortgage Trust	Series Reports	
Delta Funding Corp HEL Trust	Series Reports	Shelf Documents
Delta Funding Corporation	Series Reports	Shelf Documents
Delta Funding Corporation Prepayment Trust	Series Reports	
Delta Funding NIM Trust	Series Reports	
Delta Funding Non-Performing Loan Trust	Series Reports	
Deutsche Alt A Securities Inc. (DBALT)	Series Reports	Shelf Documents
Deutsche Mortgage and Asset Receiving Corporation	Series Reports	
Deutsche Mortgage Securities Inc. (DMSI)	Series Reports	Shelf Documents
DFC Receivables Company LLC	Series Reports	

Figure 7 Screenshot of CTSLink with Various Shelves

5.2. Unzipper

As part of the Extraction step, we must retrieve files from the archives CTSLink provides. Downloads are always double compressed in zip format. The unzipper module simply walks through a directory and unzips all the files within it. This is run twice during ingestion.

Python's builtin zip module is useful for this task. In order to extract information about the zipfile, we must build an infolist object, which returns a list of enumerable ZipInfo objects. These contain the metadata for the file. In order to determine the file type, unzipper uses the magic number of the file. We have yet to find any corrupted zip files, so this step appears to be very stable.

5.3. Move Valid Files

The only thing relevant to our extraction is .txt and .dat files. Shelf documents often include PDFs, Excel files, and manifest files, all of which are not relevant or easily mined data.

```
    def move_valid_files(rootDirectory, outputDirectory):
    """
    Moves files from rootDirectory to outputDirectory that have a .dat or
.txt extension.
    Mostly used to filter out pdfs and excel files which are occasionally
included in shelf reports.
    """
    file_count = 0
    total_start = datetime.now()
    for root, dirs, files in os.walk(rootDirectory):
        for file in files:
            start = datetime.now()
            name = Path(file).resolve().stem
            try:
                extension = os.path.splitext(file)[1]
            except IndexError:
                print("No extension found for file " + file + ". Skipping")
                continue
            if os.path.isdir(os.path.join(root, file)):
                print("Skipping directory.")
                continue
            if "dat" in extension or "txt" in extension:
                # left intentionally blank; easier to understand non-inverted
case
                pass
            else:
                print("Skipping file " + file + " as it is not a loan file.")
                continue
            if not os.path.exists(outputDirectory):
                # Create if missing
                os.makedirs(outputDirectory)
            print("Moving " + file + " to " + outputDirectory)
            if not os.path.exists(os.path.join(outputDirectory, file)):
                shutil.move(os.path.join(root, file), outputDirectory)
            else:
                print("Skipping file already found in output directory.")
            finish = datetime.now()
            elapsed = finish - start
            print('Time elpased on move job (hh:mm:ss.ms)
{}'.format(elapsed))
    total_end = datetime.now()
    total_elapsed = total_end - total_start
    print('Total time (hh:mm:ss.ms) {}'.format(total_elapsed))
```

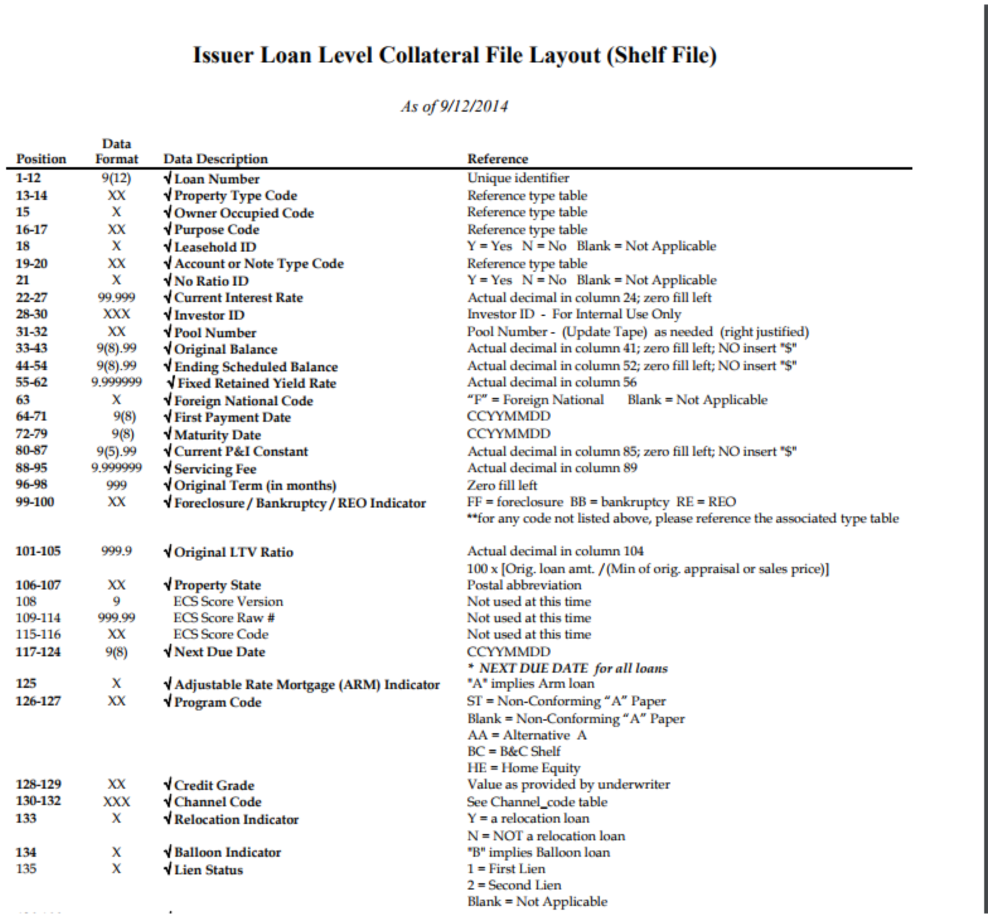
5.4. Sort By Shelf

We next sort the files based on their shelf, as part of the Filtration step. They are placed into folders corresponding to the value that follows the date entry. The Python Path module is useful for this purpose as it can reliably determine the stem of a file.

```
  def sort_by_shelf(rootDirectory):
    """Sorts in place files by their four letter shelf name."""
    file_count = 0
    total_start = datetime.now()
    for root, dirs, files in os.walk(rootDirectory):
        for file in files:
            start = datetime.now()
            name = Path(file).resolve().stem
            try:
                shelfname = name[4:]
            except IndexError:
                print("Invalid shelf name for file " + file)
                print("Skipping file.")
                continue
            dirPath = os.path.join(rootDirectory, shelfname)
            if not os.path.exists(dirPath):
                # Create if missing
                os.makedirs(dirPath)
            print("Moving " + file + " to " + dirPath)
            if not os.path.exists(os.path.join(dirPath, file)):
                shutil.move(os.path.join(root, file), dirPath)
            else:
                print("Skipping already present file " + file)
            finish = datetime.now()
            elapsed = finish - start
            print('Time elpased on move job (hh:mm:ss.ms)
{}'.format(elapsed))
    total_end = datetime.now()
    total_elapsed = total_end - total_start
    print('Total time (hh:mm:ss.ms) {}'.format(total_elapsed))
```

5.5. Parquet From Positional

This module is the most important in the tech stack. This parses Wells Fargo data one line at a time and loads them into a Python dictionary. The positions for each field are arbitrary, so we created a script that loads the positions based on what is outlined in the CTSLink shelf file layout. ParquetFromPositional is the Parsing step in the process map.



Issuer Loan Level Collateral File Layout (Shelf File)

As of 9/12/2014

Position	Data Format	Data Description	Reference
1-12	9(12)	√ Loan Number	Unique identifier
13-14	XX	√ Property Type Code	Reference type table
15	X	√ Owner Occupied Code	Reference type table
16-17	XX	√ Purpose Code	Reference type table
18	X	√ Leasehold ID	Y = Yes N = No Blank = Not Applicable
19-20	XX	√ Account or Note Type Code	Reference type table
21	X	√ No Ratio ID	Y = Yes N = No Blank = Not Applicable
22-27	99.999	√ Current Interest Rate	Actual decimal in column 24; zero fill left
28-30	XXX	√ Investor ID	Investor ID - For Internal Use Only
31-32	XX	√ Pool Number	Pool Number - (Update Tape) as needed (right justified)
33-43	9(8).99	√ Original Balance	Actual decimal in column 41; zero fill left; NO insert "$"
44-54	9(8).99	√ Ending Scheduled Balance	Actual decimal in column 52; zero fill left; NO insert "$"
55-62	9.999999	√ Fixed Retained Yield Rate	Actual decimal in column 56
63	X	√ Foreign National Code	"F" = Foreign National Blank = Not Applicable
64-71	9(8)	√ First Payment Date	CCYYMMDD
72-79	9(8)	√ Maturity Date	CCYYMMDD
80-87	9(5).99	√ Current P&I Constant	Actual decimal in column 85; zero fill left; NO insert "$"
88-95	9.999999	√ Servicing Fee	Actual decimal in column 89
96-98	999	√ Original Term (in months)	Zero fill left
99-100	XX	√ Foreclosure / Bankruptcy / REO Indicator	FF = foreclosure BB = bankruptcy RE = REO **for any code not listed above, please reference the associated type table
101-105	999.9	√ Original LTV Ratio	Actual decimal in column 104 100 x [Orig. loan amt. /(Min of orig. appraisal or sales price)]
106-107	XX	√ Property State	Postal abbreviation
108	9	ECS Score Version	Not used at this time
109-114	999.99	ECS Score Raw #	Not used at this time
115-116	XX	ECS Score Code	Not used at this time
117-124	9(8)	√ Next Due Date	CCYYMMDD * *NEXT DUE DATE for all loans*
125	X	√ Adjustable Rate Mortgage (ARM) Indicator	"A" implies Arm loan

126-127	XX	√ Program Code	ST = Non-Conforming "A" Paper
			Blank = Non-Conforming "A" Paper
			AA = Alternative A
			BC = B&C Shelf
			HE = Home Equity

128-129	XX	√ Credit Grade	Value as provided by underwriter
130-132	XXX	√ Channel Code	See Channel_code table

133	X	√ Relocation Indicator	Y = a relocation loan
			N = NOT a relocation loan

134	X	√ Balloon Indicator	"B" implies Balloon loan
135	X	√ Lien Status	1 = First Lien 2 = Second Lien Blank = Not Applicable

Figure 8: CTSLink shelf file layout

The shelf file layout tells us the location of each field in the file. One row in the file represents a single loan for a single month. The month itself and the name of the shelf is stored in the name of the file. Each file contains anywhere between 1,000 lines and 25,000,000 lines. It depends on how many loans there are for that month.

We extract the shelfname and month here. A shelf name is a string of arbitrary length that comes after the date entry. Shelf files use the following format in their name: MMYYshelfname. So, a January 2001 WFMBS shelf file would be 0101wfmbs.txt. Files are occasionally incorrectly named. The most common mistake is a three-character date code instead of a complete MMYY date. Some of these were manually renamed before processing; however, if such files are encountered, the software makes a note of it and ignores the file.

Thankfully, Python supports a built-in method of extracting substrings called slicing. Slicing allows us to do many things, but in this case, we are using it to extract a substring. Note that CTSLink's file uses an index that starts at 1, so the indices in our parser are adjusted to account for the fact that Python (and most modern languages) begin their indices at 0.

Since the format is arbitrary, we must manually encode the shelf file layout in some way. There is no programmatic way to determine the locations of the fields.

When we parse the files, we extract the file name. This presents several potential issues. Sometimes, file names have typos in them. We check for that using some simple string comparisons. Before dt is passed into process_single_line, it is checked for validity:

```
      name = Path(path).resolve().stem
# assuming YYMM(shelf)
print("File name is " + name)
try:
    year = int("20" + name[0:2])
except ValueError:
    print("Skipping invalid file " + name)
    return
if year > datetime.datetime.now().year:
    print("Shelf reports from the future indicate an invalid file name.
Skipping.")
    return
elif year < datetime.datetime.now().year and not ingest_old_files:
    print("Shelf reports from the past indicate that we may have already
ingested this file.")
    return
try:
    month = int(name[2:4])
```

```
except ValueError:
    print("Skipping invalid file " + name  + " because it appears to have an
invalid month value.")
    return
if month > 12 or month < 1:
    print("Skipping invalid file " + name + " because it appears to have an
invalid month value.")
    return
if month != datetime.datetime.now().month and not ingest_old_files:
    print("Skipping file from the past as this indicates that we may have
already ingested this file.")
    return
```

ingest_old_files is by default true. It can be disabled to prevent the function from processing any files whose shelf date is prior to the current month. This is useful as it prevents data duplication, but because shelf publication dates vary from shelf to shelf, we enable ingestion of old files by default. This feature can be disabled in case someone needs to modify the code in the future.

Once we have the data in a Python dictionary, it gets significantly easier to serialize it into a new format. We use the Python package fastparquet to perform this operation. First we compile a list of dictionary objects, each representing a line in the file. Then we simply use Pandas dataframes to build a single dataframe that can be written into a PARQ file.

Parquet files can be easily read by the Spark cluster, so they are an ideal choice for compressing large amounts of text. The compression ratio is roughly 10:1; that is to say, 400 GB of data will result in roughly 40 GB of Parquet archives. We mostly attribute this to the large amount of whitespace and repeated digits found in the files. The nature of compression is such that the less diverse the set of compressed entities is, the better the compression ratio.

Because of the significant file IO required to read in, compress, and write out the PARQ files, they are created once and then appended to throughout processing.

PARQ compression is a CPU bound operation. This is due to the algorithm used to create PARQ archives. Parquet is a columnar format; compression is performed column by column. Like many popular compression schemes, Parquet uses run length encoding (RLE). A value is only stored once, followed by the number of occurrences. Most of the files processed by ParquetFromPositional have large sections of intentionally omitted data, such as the address of the house being purchased, so Parquet compression significantly reduces the space consumed.

5.6. Multiprocessing PARQ compression

Parquet compression is a time-consuming operation, particularly when so much data is involved. To remedy this, we use a ProcessPoolExecutor to start 6 processes, which each take a single shelf folder. This prevents the need for an explicit mutex strategy as each process executes on a single archive at a time. This is why we execute SortByShelf, not only for organizational consistency, but also to distribute work.

```
    """
Distributes the PARQ process over max_worker threads. Setting the number too
high may lock up your machine.
"""
shelves = [(os.path.join(target, x[0]), output) for x in os.walk(target)]
#get all shelf files

with ProcessPoolExecutor(max_workers=max_workers) as executor:
    executor.map(walk_entire_directory, shelves)
```

Like any application making use of parallelization, it is important to be aware of the fact that the thread and process pools are limited. Based on some rough testing, we found that 6 processes allows a 400-500% performance increase, while still allowing the rest of the OS to function normally.

5.6.1. DateTime idiosyncrasies

One of the problems we encountered during development was issues with using POSIX timestamps with the Spark cluster. Python's default datetime class reports time in microseconds since January 1st, 1970. This is an exceptionally granular way of describing date, and it introduces many confusing issues. Since this measures milliseconds since a fixed point in time, (also known as an epoch), determining which day it is requires the time zone. While there are tools to do this, it adds an unnecessary level of complexity. Most importantly, however, is the incompatibility with Azure DataBricks. DataBricks cannot process such a large timestamp; this is a known bug in Parquet as well as DataBricks. To resolve this, we opted to store dates in a CCYYMMDD string format to guarantee compatibility across all versions of Spark.

5.7. Schema Scanner

We discovered that some of the Parquet files we generated ended up having mismatched schema as others. To combat this, we built a simple module that compares the schema of the first file in the directory with every other file. This is an acceptance test we implemented to ensure that no data with a mismatched schema ends up in Azure.

5.8. Azure Datalake Upload

The last step in the tech stack before the data is ready for analysis is to upload it to Azure. Microsoft provides the package adl_store for this purpose. To authenticate (not authorize) we use an Azure Data Lake (ADL) token to upload files. The function ask_token prompts the user for the token. For convenience, the default department Risk Data Lake token is used if no value is entered; however, it is omitted from this report for security reasons. ask_token returns a 3-tuple string with the values the user inputted, or the default values if none were entered. They are then immediately loaded into an adl_store.lib.auth object for use.

```
      def ask_token():
    """Asks the user for ADL token information or supplies hardcoded values
if none are provided. Used in ingestion.py as well."""
    client_id = input("Input client key; leave blank for default riskdatalake
token: ")
    if client_id == "":
        client_id = 'Intentionally Omitted'
    client_secret = input("Input client secret; leave blank for default
riskdatalake token: ")
    if client_secret == "":
        client_secret = 'Intentionally Omitted'
    tenant_id = input("Input tenant id; leave blank for default riskdatalake
token: ")
    if tenant_id == "":
        tenant_id = 'Intentionally Omitted'
    return (client_id, client_secret, tenant_id)
```

While hardcoded token values would normally be undesirable, a heavy emphasis was placed on a fast, painless install with minimal configuration over flexibility for customization. This function is also used in ingestion to ask the user what token they wish to use. From there, the upload to ADL is straightforward. The upload will always place files in a directory - never the root, and will not upload files when they already exist on the datalake. Since it is required by adl_store, all files are uploaded as binary files.

5.9. Ingestion

Ingestion is the composition root for the entire tech stack. It loads all of the modules and runs them sequentially. Prior to running, ingestion asks for the following information:

- Whether to use an explicit directory instead of a tempfile, and if so, which directory to use
- What directory to upload files to on the datalake
- Whether or not to run on a small subset of data for testing purposes
- Whether or not to delete transient files when done
- The Azure Data Lake token, defaulting to Risk's token

Ingestion then creates five folders, phase1 through phase5. This is to aid with debugging, because if a step fails, it can be resumed manually by executing the scripts individually. Ingestion merely runs the other modules in sequence; it has no business logic of its own.

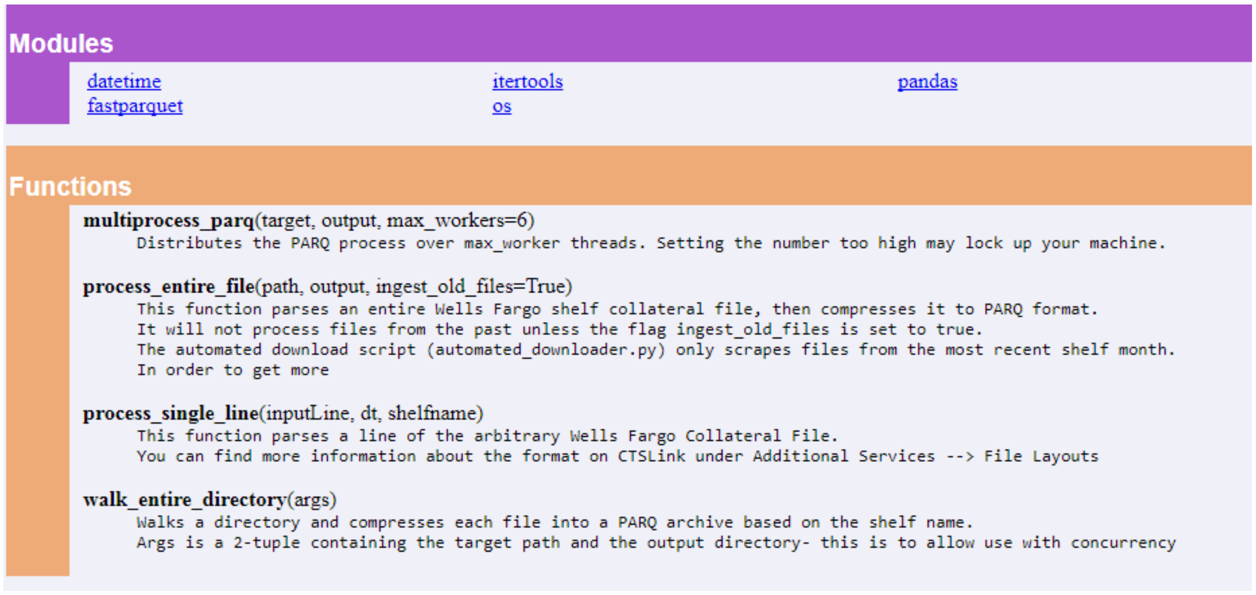
5.10. Miscellaneous tools

5.10.1. Installer

Installer was a request by AG to speed preparation for use. It simply calls pip repeatedly to download all of the required packages. While this task is often left to whoever is using the repository, this does reduce the effort required to get started. Running main will install the packages required.

5.10.2. DocGenerator

DocGenerator was created to help ensure that the automatically generated documentation is up to date. This calls pydoc on the modules in the project, as well as the other modules used in the software. This generates HTML files that show a neatly formatted page with the docstrings embedded within it. Each time ingestion is run, the user is prompted to generate the latest documentation.



Modules

datetime
fastparquet
itertools
os
pandas

Functions

multiprocess_parq(target, output, max_workers=6)
```
Distributes the PARQ process over max_worker threads. Setting the number too high may lock up your machine.
```

process_entire_file(path, output, ingest_old_files=True)
```
This function parses an entire Wells Fargo shelf collateral file, then compresses it to PARQ format.
It will not process files from the past unless the flag ingest_old_files is set to true.
The automated download script (automated_downloader.py) only scrapes files from the most recent shelf month.
In order to get more
```

process_single_line(inputLine, dt, shelfname)
```
This function parses a line of the arbitrary Wells Fargo Collateral File.
You can find more information about the format on CTSLink under Additional Services --> File Layouts
```

walk_entire_directory(args)
```
Walks a directory and compresses each file into a PARQ archive based on the shelf name.
Args is a 2-tuple containing the target path and the output directory- this is to allow use with concurrency
```

Figure 9 Screenshot of HTML with docstrings Embedded

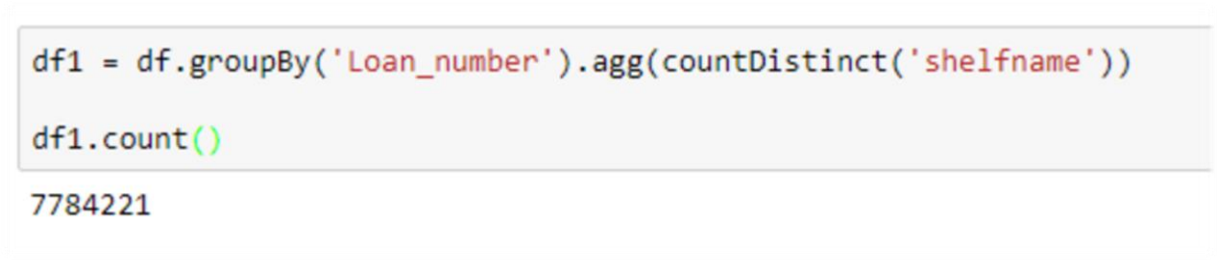
6. Data Wrangling and Analysis

6.1. Data Wrangling

After the data is uploaded to the data lake, the data is available to be used through Spark. However, before the data can be processed, certain operations need to be performed to prepare the data. These operations are not predefined but are specific to each data set. While exploring the data set, certain irregularities become apparent that need correction. A large portion of this project was spent on this stage. This stage ensured data integrity by finding and correcting irregularities.

6.1.1. Duplicate Loan Numbers

The risk management team had estimated the data set to account for 9.5 million loans. When a count function was initially performed, the total number of loans was 7,784,221. This clearly suggested inconsistencies within the data.

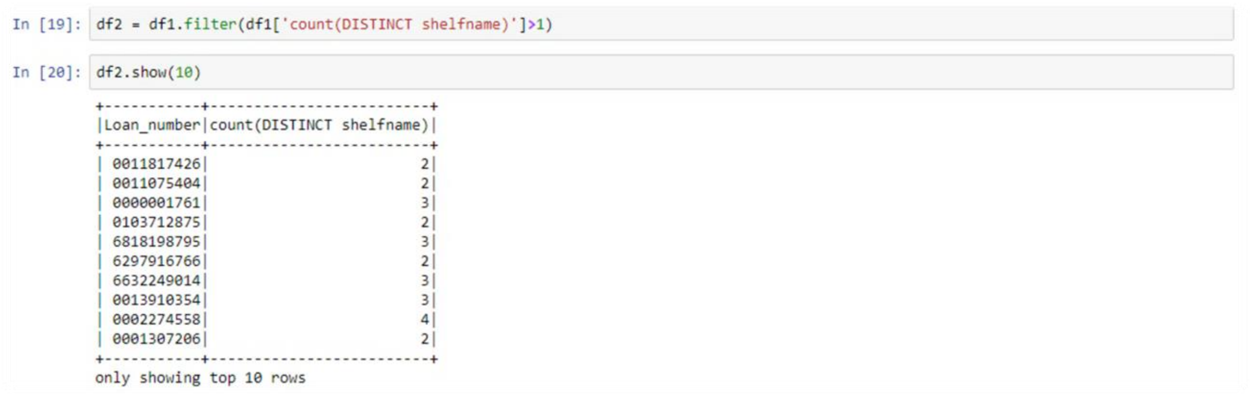


```
df1 = df.groupBy('Loan_number').agg(countDistinct('shelfname'))

df1.count()
```

```
7784221
```

Figure 10 Count Operation for Total Loans on Spark

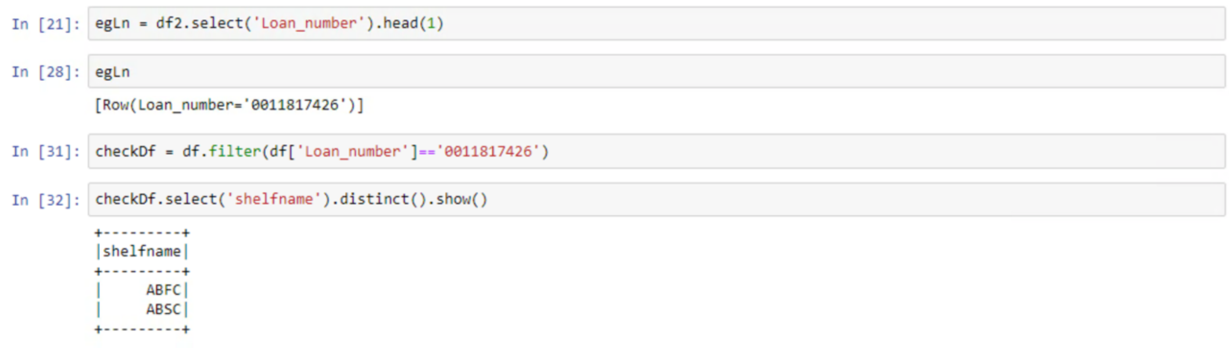
The first action was to check if the entire data set has been uploaded correctly. Once that was confirmed, there was suspicion that multiple shelves have the same loan numbers leading to a lesser count than total. The filter function was used to show all loan numbers that occurred more than once. In Figure 11, the first line executes the code and the second line shows a sample of the operation performed.



```
In [19]: df2 = df1.filter(df1['count(DISTINCT shelfname)']>1)

In [20]: df2.show(10)

+-----------+-------------------------+
|Loan_number|count(DISTINCT shelfname)|
+-----------+-------------------------+
| 0011817426|                        2|
| 0011075404|                        2|
| 0000001761|                        3|
| 0103712875|                        2|
| 6818198795|                        3|
| 6297916766|                        2|
| 6632249014|                        3|
| 0013910354|                        3|
| 0002274558|                        4|
| 0001307206|                        2|
+-----------+-------------------------+
only showing top 10 rows
```

Figure 11 Filter Operation and Output

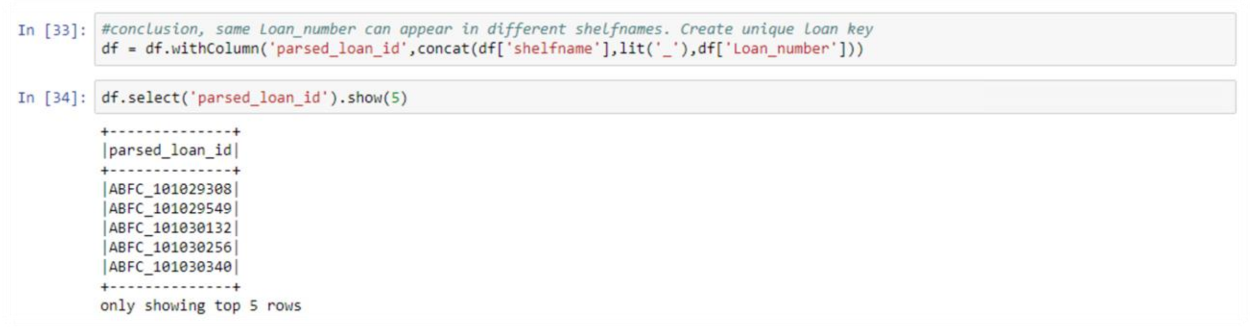
The table shows that the count of some loan numbers are more than 1 and sometimes even more. For example, if one were to search the data set for the loan number "0011817426", then it should appear twice in two different shelves. Figure 12 below does such an operation:



```
In [21]: egLn = df2.select('Loan_number').head(1)

In [28]: egLn

[Row(Loan_number='0011817426')]

In [31]: checkDf = df.filter(df['Loan_number']=='0011817426')

In [32]: checkDf.select('shelfname').distinct().show()

+---------+
|shelfname|
+---------+
|     ABFC|
|     ABSC|
+---------+
```

Figure 12 Operations to Check Loan Numbers on Spark

The particular loan number in question appears in shelf "Asset Backed Funding Corporation (ABFC)" and "Asset Backed Securities Corp (ABSC)". Although it is the same loan number, they are different loans entirely and should be treated separately for data processing purposes. To solve this issue, each loan number was assigned a new loan number with the new format of "Shelfname_LoanNumber".



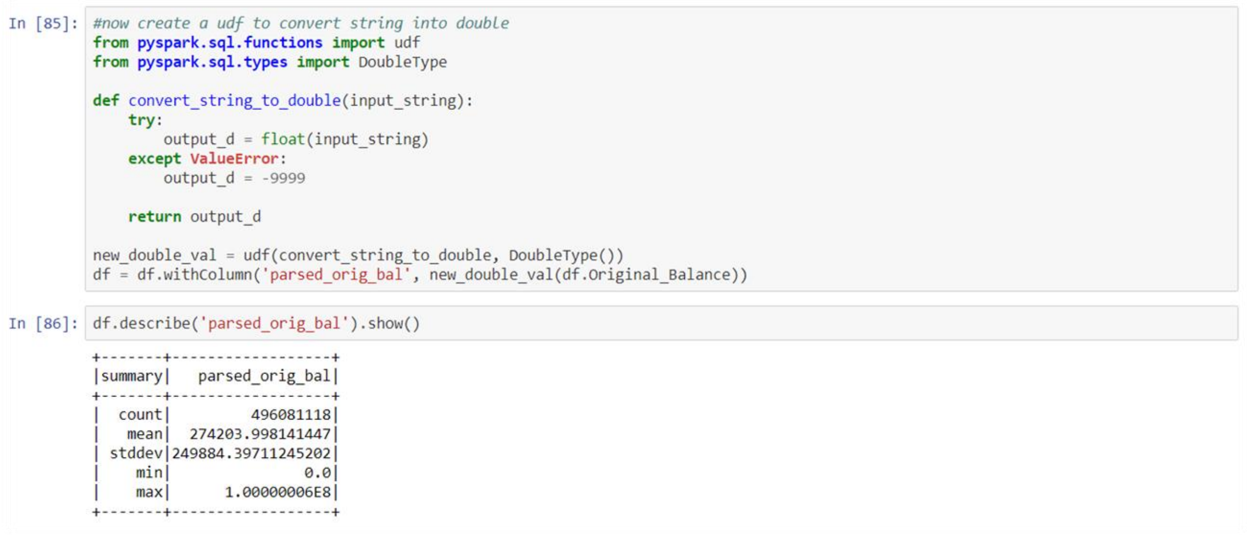
```
In [33]: #conclusion, same Loan_number can appear in different shelfnames. Create unique Loan key
         df = df.withColumn('parsed_loan_id',concat(df['shelfname'],lit('_'),df['Loan_number']))

In [34]: df.select('parsed_loan_id').show(5)

         +--------------+
         |parsed_loan_id|
         +--------------+
         |ABFC_101029308|
         |ABFC_101029549|
         |ABFC_101030132|
         |ABFC_101030256|
         |ABFC_101030340|
         +--------------+
         only showing top 5 rows
```

Figure 13 Operations to Create Unique Loan Numbers on Spark

The new total of loan numbers was raised to 9,852,826 which was much closer to the estimate amount of loan data. Had this irregularity been overlooked, approximately data of 2 million loans, 21% of the total data, would be accidentally excluded. Any analysis with respect to loan numbers would have also been erroneous.

6.1.2. Checking Integrity of Original Balance Values

A sanity check was performed for certain field values that were relevant to the data processing. Original balance (loan balance) was one relevant field that was problematic as spark was unable to perform operations on the field. When able, the value itself was incorrect. The problem was found in the formatting of the field. The field values were string types and needed to be converted to float values. A User Defined Function (udf) was written to perform this conversion. Figure 14 shows these operations.



```
In [85]: #now create a udf to convert string into double
         from pyspark.sql.functions import udf
         from pyspark.sql.types import DoubleType

         def convert_string_to_double(input_string):
             try:
                 output_d = float(input_string)
             except ValueError:
                 output_d = -9999

             return output_d

         new_double_val = udf(convert_string_to_double, DoubleType())
         df = df.withColumn('parsed_orig_bal', new_double_val(df.Original_Balance))

In [86]: df.describe('parsed_orig_bal').show()

         +-------+------------------+
         |summary|   parsed_orig_bal|
         +-------+------------------+
         |  count|         496081118|
         |   mean|  274203.998141447|
         | stddev|249884.39711245202|
         |    min|               0.0|
         |    max|      1.00000006E8|
         +-------+------------------+
```

Figure 14 Operations to Change Formatting on Spark

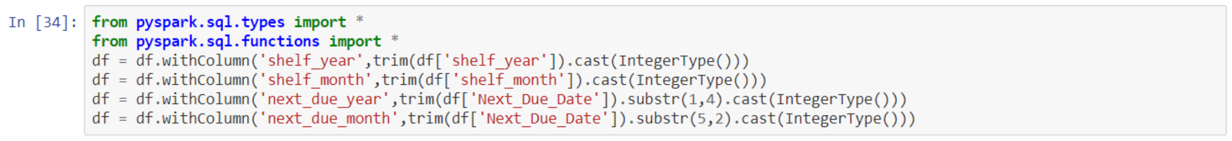
The second command generates a quick summary of the field values. The average loan balance for a Mortgage Backed Security with Alternative A credit in USA is estimated to be $250,000 (Appleyard, 2004). The new mean loan balance of $274,204 is close to the estimate confirming that the values are formatted correctly.

6.1.3. Delinquency Status of Loans

Delinquency status is the current status of a loan with respect to delay in monthly payments if any. The delinquency status deteriorates as the loaner defaults on more monthly payments. Delinquency status is an important indicator of loan performance and the Wells Fargo data set did not have this field built-in. This field was added as part of the data wrangling stage.

6.1.3.1. Standardizing Date Values

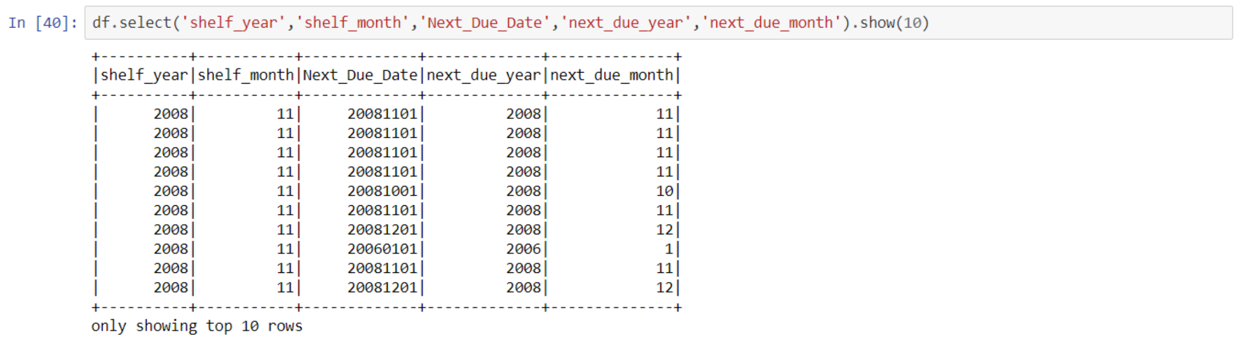
To be able to find difference in monthly due dates and monthly payments, the dates have to be in proper format. Since all fields are in string format including numbers, specific fields have to be assigned new formats such as integer type to numbers. The operations below remove whitespace (trim) and cast types to the fields (columns) current date and next due date. They also create two separate columns for the year and month of each dates to facilitate subtraction later to find the difference in months.



```
In [34]: from pyspark.sql.types import *
         from pyspark.sql.functions import *
         df = df.withColumn('shelf_year',trim(df['shelf_year']).cast(IntegerType()))
         df = df.withColumn('shelf_month',trim(df['shelf_month']).cast(IntegerType()))
         df = df.withColumn('next_due_year',trim(df['Next_Due_Date']).substr(1,4).cast(IntegerType()))
         df = df.withColumn('next_due_month',trim(df['Next_Due_Date']).substr(5,2).cast(IntegerType()))
```

Figure 15 Operations to Create Columns for Dates on Spark

Below is the output of these operations for the first 10 rows of the data set:

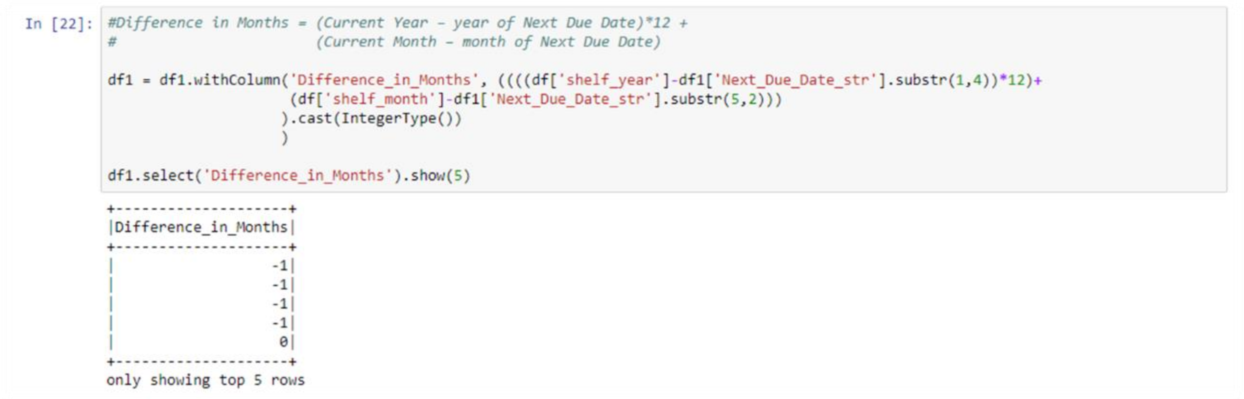


```
In [40]: df.select('shelf_year','shelf_month','Next_Due_Date','next_due_year','next_due_month').show(10)

         +----------+-----------+-------------+-------------+--------------+
         |shelf_year|shelf_month|Next_Due_Date|next_due_year|next_due_month|
         +----------+-----------+-------------+-------------+--------------+
         |      2008|         11|     20081101|         2008|            11|
         |      2008|         11|     20081101|         2008|            11|
         |      2008|         11|     20081101|         2008|            11|
         |      2008|         11|     20081101|         2008|            11|
         |      2008|         11|     20081001|         2008|            10|
         |      2008|         11|     20081101|         2008|            11|
         |      2008|         11|     20081201|         2008|            12|
         |      2008|         11|     20060101|         2006|             1|
         |      2008|         11|     20081101|         2008|            11|
         |      2008|         11|     20081201|         2008|            12|
         +----------+-----------+-------------+-------------+--------------+
         only showing top 10 rows
```

Figure 16 Output of Operations for First 10 Rows

6.1.3.2. Binning Loans by Delinquency Status

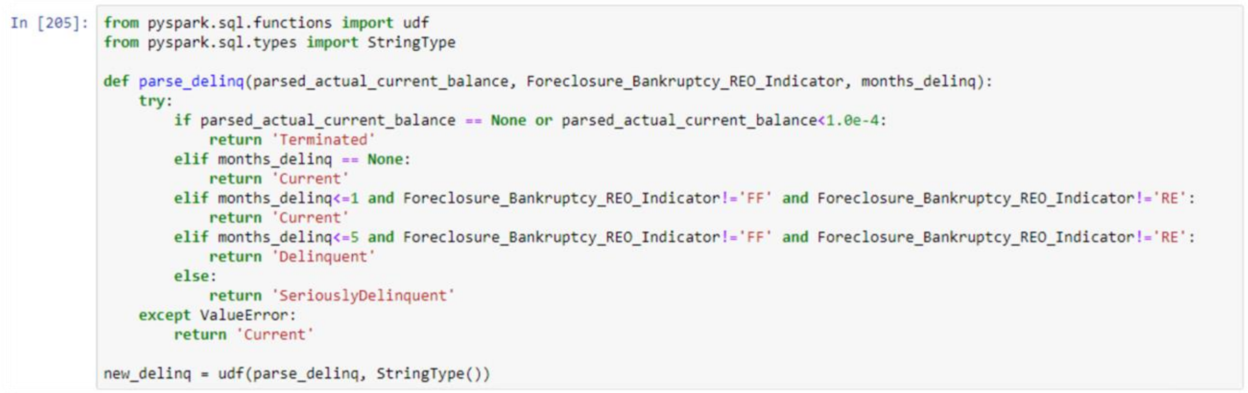
The difference between current shelf and the next due date field was calculated. This difference was calculated in months.



```
In [22]: #Difference in Months = (Current Year - year of Next Due Date)*12 +
         #                       (Current Month - month of Next Due Date)

         df1 = df1.withColumn('Difference_in_Months', ((((df['shelf_year']-df1['Next_Due_Date_str'].substr(1,4))*12)+
                               (df['shelf_month']-df1['Next_Due_Date_str'].substr(5,2)))
                              ).cast(IntegerType())
                              )

         df1.select('Difference_in_Months').show(5)

         +--------------------+
         |Difference_in_Months|
         +--------------------+
         |                  -1|
         |                  -1|
         |                  -1|
         |                  -1|
         |                   0|
         +--------------------+
         only showing top 5 rows
```

Figure 17 Operations to Find Difference in Months on Spark

Next, the loans were binned in four separate categories that serve as delinquency statuses. **Terminated** loans are loans that have been paid in full and have a zero loan balance remaining. Loans payments that are being paid, already paid, or are ahead of schedule are **Current**. If a loan payment is behind by 5 months or less, the loan is binned as **Delinquent**. If a loan payment is behind by more than 5 months, then the loan is in a **Seriously Delinquent** status. An important distinction is that loans of Delinquent or Seriously Delinquent status are only binned if they are marked for foreclosure in the future.



```
In [205]: from pyspark.sql.functions import udf
          from pyspark.sql.types import StringType

          def parse_delinq(parsed_actual_current_balance, Foreclosure_Bankruptcy_REO_Indicator, months_delinq):
              try:
                  if parsed_actual_current_balance == None or parsed_actual_current_balance<1.0e-4:
                      return 'Terminated'
                  elif months_delinq == None:
                      return 'Current'
                  elif months_delinq<=1 and Foreclosure_Bankruptcy_REO_Indicator!='FF' and Foreclosure_Bankruptcy_REO_Indicator!='RE':
                      return 'Current'
                  elif months_delinq<=5 and Foreclosure_Bankruptcy_REO_Indicator!='FF' and Foreclosure_Bankruptcy_REO_Indicator!='RE':
                      return 'Delinquent'
                  else:
                      return 'SeriouslyDelinquent'
              except ValueError:
                  return 'Current'

          new_delinq = udf(parse_delinq, StringType())
```

Figure 18 Operations to Bin Loans by Delinquency Status on Spark

After the binning of loans, the field or column is ready to be used in data processing.

6.2. Data Processing

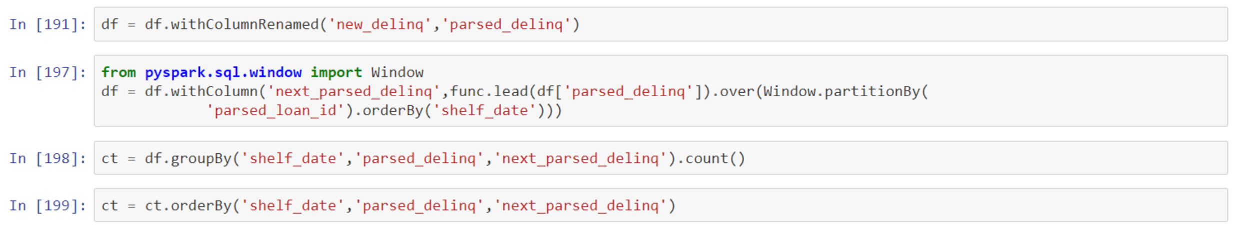
Once the data has been transformed into the desirable format with the necessary fields, it is ready to be explored for various insights. The Risk Team of Angelo Gordon eventually wants to use this data set amongst others to build a statistical model to predict the upcoming delinquency status of a loan, which in turn indicates if the loan is to improve or deteriorate. This is largely modeled on various transition (Markov) matrices. A change from someone paying monthly on time (Current) can pay off their loan (Terminated) or not pay for a couple of months (Delinquent). This change can be seen as a transition between two states. Hence, knowing the probability of going from one state to the other can be useful in determining which loans will perform well with consistent payments throughout. A matrix consisting of such transition probabilities would be a transition matrix.

In order to better understand such a model and how to start building it, certain sample analyses were performed. Section 6.2.1 explains how to build a transition matrix and shows two sample models. This procedure also serves as a sanity check to ensure all the data retrieval and manipulation performed until this stage is correct.

6.2.1. Delinquency Status Transition Matrix

A variety of indicators can be used to track the progression of loan repayment and/or the quality of loan. For our purposes, delinquency status will be used as an indicator to which other attributes can be measured against.

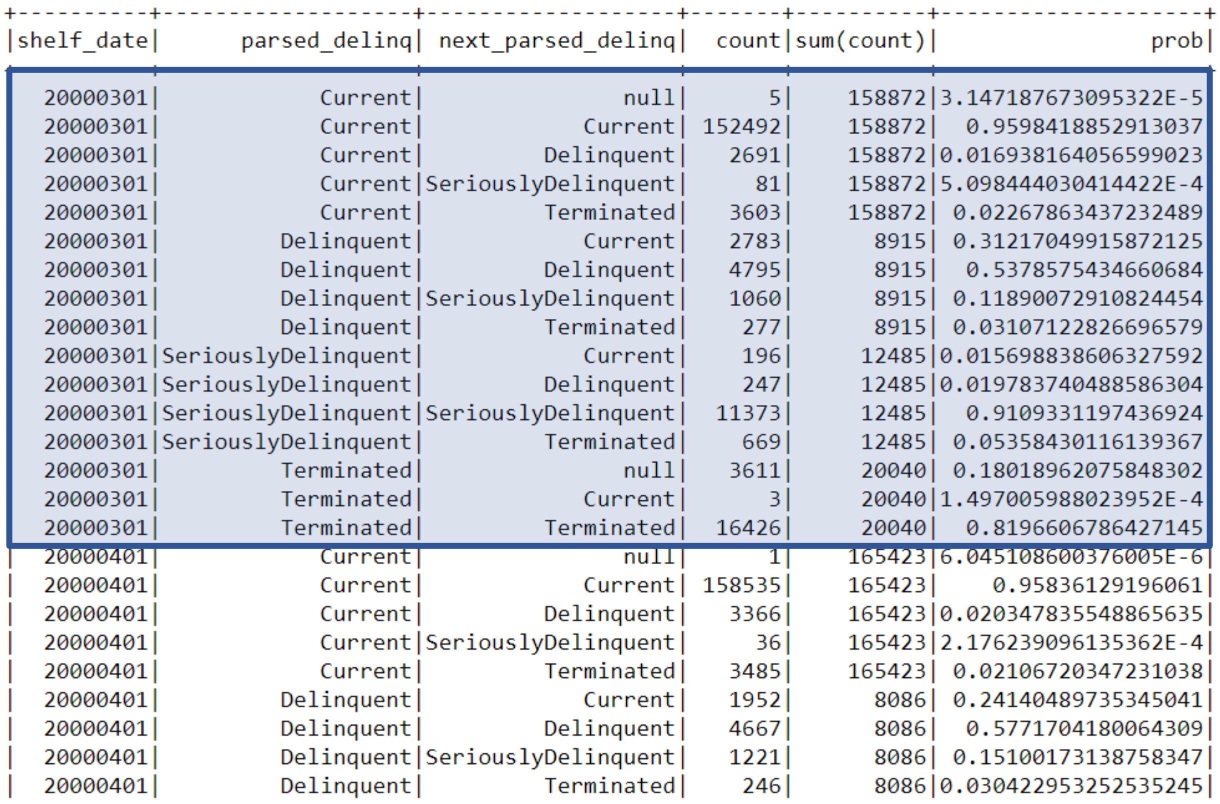
Firstly, the column of delinquency status is renamed to “parsed_delinq”. The second operation creates a new column that has the next delinquency status when the loan advances to the next month. The third operations creates an additional column that shows the number of loans that have made that particular change in delinquency status. Finally, the fourth operation orders them the table in a chronological manner. Figure 19 illustrates the said operations:



```
In [191]: df = df.withColumnRenamed('new_delinq','parsed_delinq')

In [197]: from pyspark.sql.window import Window
          df = df.withColumn('next_parsed_delinq',func.lead(df['parsed_delinq']).over(Window.partitionBy(
                        'parsed_loan_id').orderBy('shelf_date')))

In [198]: ct = df.groupBy('shelf_date','parsed_delinq','next_parsed_delinq').count()

In [199]: ct = ct.orderBy('shelf_date','parsed_delinq','next_parsed_delinq')
```

Figure 19 Operations to Create Column of 'next_parsed_delinq' on Spark

After finding the count, a sum column consisting of the total loans for that particular month is added. This later allows one to find the probability for each row as it is simply a function of the count over the sum. The output of these operations can be seen in Figure 20:



```
+----------+-----------------+-----------------+------+----------+--------------------+
|shelf_date|     parsed_delinq| next_parsed_delinq| count|sum(count)|                prob|
+----------+-----------------+-----------------+------+----------+--------------------+
|  20000301|          Current|             null|     5|    158872|3.147187673095322E-5|
|  20000301|          Current|          Current|152492|    158872|  0.9598418852913037|
|  20000301|          Current|       Delinquent|  2691|    158872|0.016938164056599023|
|  20000301|          Current|SeriouslyDelinquent|    81|    158872|5.098444030414422E-4|
|  20000301|          Current|       Terminated|  3603|    158872| 0.02267863437232489|
|  20000301|       Delinquent|          Current|  2783|      8915| 0.31217049915872125|
|  20000301|       Delinquent|       Delinquent|  4795|      8915|  0.5378575434660684|
|  20000301|       Delinquent|SeriouslyDelinquent|  1060|      8915| 0.11890072910824454|
|  20000301|       Delinquent|       Terminated|   277|      8915| 0.03107122826696579|
|  20000301|SeriouslyDelinquent|          Current|   196|     12485|0.015698838606327592|
|  20000301|SeriouslyDelinquent|       Delinquent|   247|     12485|0.019783740488586304|
|  20000301|SeriouslyDelinquent|SeriouslyDelinquent| 11373|     12485|  0.9109331197436924|
|  20000301|SeriouslyDelinquent|       Terminated|   669|     12485| 0.05358430116139367|
|  20000301|       Terminated|             null|  3611|     20040| 0.18018962075848302|
|  20000301|       Terminated|          Current|     3|     20040|1.497005988023952E-4|
|  20000301|       Terminated|       Terminated| 16426|     20040|  0.8196606786427145|
|  20000401|          Current|             null|     1|    165423|6.045108600376005E-6|
|  20000401|          Current|          Current|158535|    165423|    0.95836129196061|
|  20000401|          Current|       Delinquent|  3366|    165423|0.020347835548865635|
|  20000401|          Current|SeriouslyDelinquent|    36|    165423|2.176239096135362E-4|
|  20000401|          Current|       Terminated|  3485|    165423| 0.02106720347231038|
|  20000401|       Delinquent|          Current|  1952|      8086| 0.24140489735345041|
|  20000401|       Delinquent|       Delinquent|  4667|      8086|  0.5771704180064309|
|  20000401|       Delinquent|SeriouslyDelinquent|  1221|      8086| 0.15100173138758347|
|  20000401|       Delinquent|       Terminated|   246|      8086|0.030422953252535245|
```

Figure 20 Output of Operations to Build Transition Matrix on Spark

The last cell of the first row of Figure 20 shows a null value for the next delinquency status. This is likely because some loans are still on file when they need to be deleted. A transition matrix can be built for each month. As an example, a transition matrix is below for March 1st, 2000 (Marked Region).

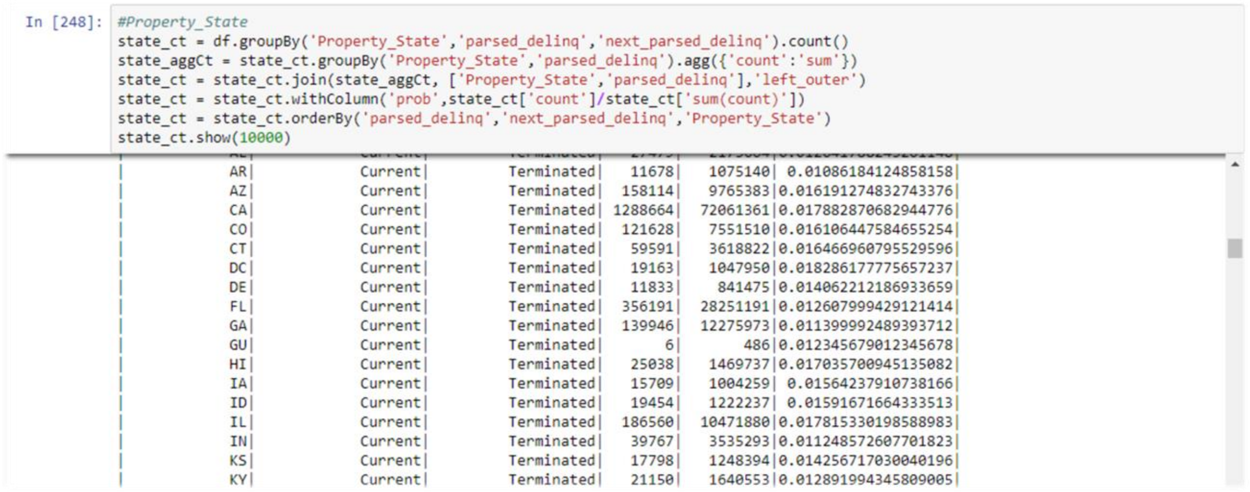
	Current	Delinquent	Seriously Delinquent	Terminated	Null
Current	0.960	0.017	0.000	0.023	0.000
Delinquent	0.312	0.538	0.119	0.031	
Seriously Delinquent	0.016	0.020	0.911	0.054	
Terminated	0.000			0.820	0.180

Figure 21 Delinquency Status Transition Matrix for month of March 1, 2000

Since each row in this matrix sums to 1, this is said to be a Right Stochastic Matrix. A group function passed on to Spark allowed to consolidate all transition matrices into one. This matrix can be also grouped along with other attributes such as borrower's FICO score or state of residence.

6.2.2. Pay Off Rate in Different States of USA

This section will explore the data to understand which state has the highest pay off rate with delinquency status as an indicator. A table was created using the consolidated transition matrix and grouping all loans by the state. The operations necessary are shown along with the table in Figure 22.



```
In [248]: #Property_State
state_ct = df.groupBy('Property_State','parsed_delinq','next_parsed_delinq').count()
state_aggCt = state_ct.groupBy('Property_State','parsed_delinq').agg({'count':'sum'})
state_ct = state_ct.join(state_aggCt, ['Property_State','parsed_delinq'],'left_outer')
state_ct = state_ct.withColumn('prob',state_ct['count']/state_ct['sum(count)'])
state_ct = state_ct.orderBy('parsed_delinq','next_parsed_delinq','Property_State')
state_ct.show(10000)
```

```
|             AR|       Current|       Terminated|  11678|   1075140| 0.01086184124858158|
|             AZ|       Current|       Terminated| 158114|   9765383|0.016191274832743376|
|             CA|       Current|       Terminated|1288664|  72061361|0.017882870682944776|
|             CO|       Current|       Terminated| 121628|   7551510|0.016106447584655254|
|             CT|       Current|       Terminated|  59591|   3618822|0.016466960795529596|
|             DC|       Current|       Terminated|  19163|   1047950|0.018286177775657237|
|             DE|       Current|       Terminated|  11833|    841475|0.014062212186933659|
|             FL|       Current|       Terminated| 356191|  28251191|0.012607999429121414|
|             GA|       Current|       Terminated| 139946|  12275973|0.011399992489393712|
|             GU|       Current|       Terminated|      6|       486|0.012345679012345678|
|             HI|       Current|       Terminated|  25038|   1469737|0.017035700945135082|
|             IA|       Current|       Terminated|  15709|   1004259| 0.01564237910738166|
|             ID|       Current|       Terminated|  19454|   1222237| 0.01591671664333513|
|             IL|       Current|       Terminated| 186560|  10471880|0.017815330198588983|
|             IN|       Current|       Terminated|  39767|   3535293|0.011248572607701823|
|             KS|       Current|       Terminated|  17798|   1248394|0.014256717030040196|
|             KY|       Current|       Terminated|  21150|   1640553|0.012891994345809005|
```

Figure 22 Operations to Group US States by Delinquency Status

This table is later exported to a local drive as a csv file. This csv file can be used with Data Visualization BI Tools to create powerful graphs. In this case, the csv file was used with excel to use inbuilt visualization tools. The state change from paying monthly to termination of the loan is isolated. This isolates loaners who have been paying consistently and in the end have paid off their loan which is different from loaners whose property has been foreclosed. A screenshot of the Excel file can be found in Appendix 10.3.1 and the resulting visual is in Figure 23. The warmer the color, the higher the transition probability.

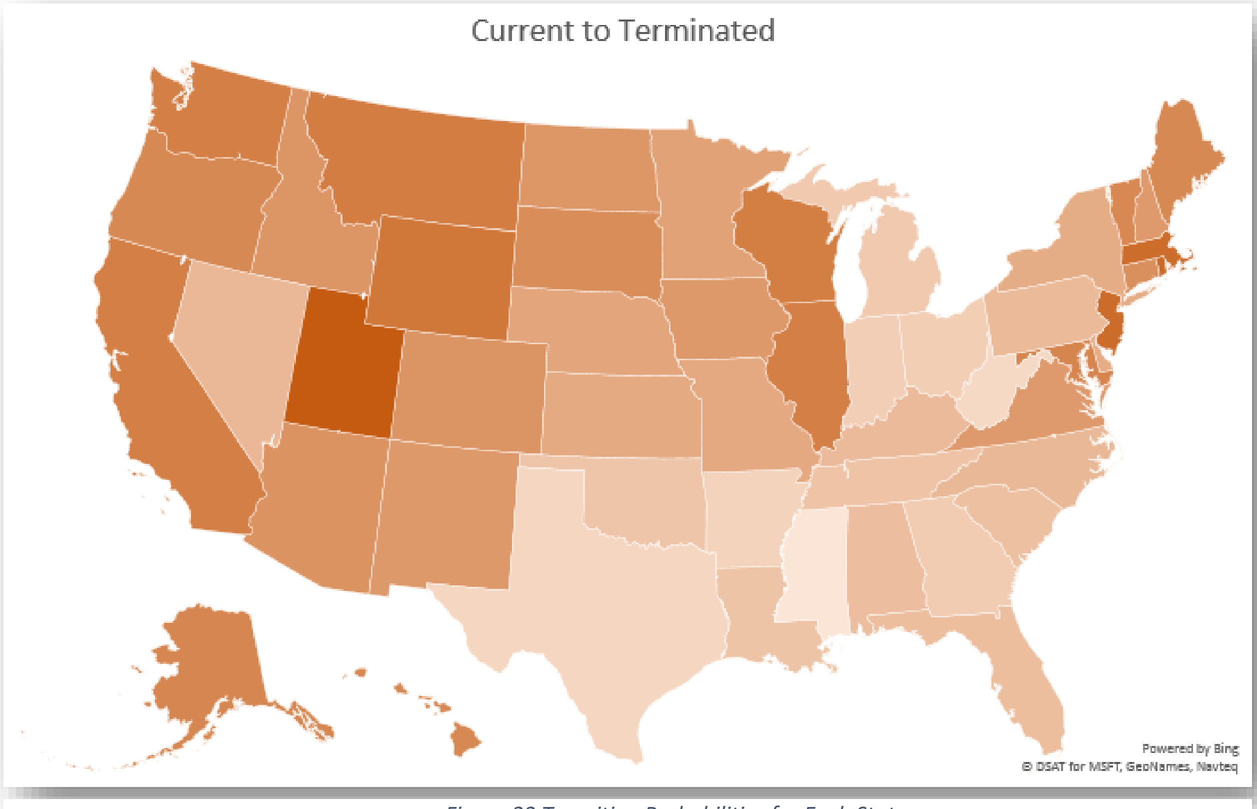


Figure 23 Transition Probabilities for Each State

Table 1 shows the top 10 states with the highest pay off rate.

Table 1 Highest Pay Off Rate in States Ranked Descending Order

Position	States
1	Utah
2	Rhode Island
3	New Jersey
4	Massachusetts
5	Wyoming
6	Washington DC
7	Montana
8	Wisconsin
9	California
10	Illinois

It is important to note at this point that this is just a simplification of a possible model. There are many other factors to be considered such as population, FICO scores, etc that would alter the above list.

6.2.3. Foreclosure in Various States

The chances of foreclosure in each state can be useful information to decide which loans are easily cleared of debt through foreclosure. This is shown by the transition from the state Seriously Delinquent to the state Terminated. This means that the loaner has moved from faulting for a long period of time to the sale of property. Figure 24 is a map of this transition probability for each state.

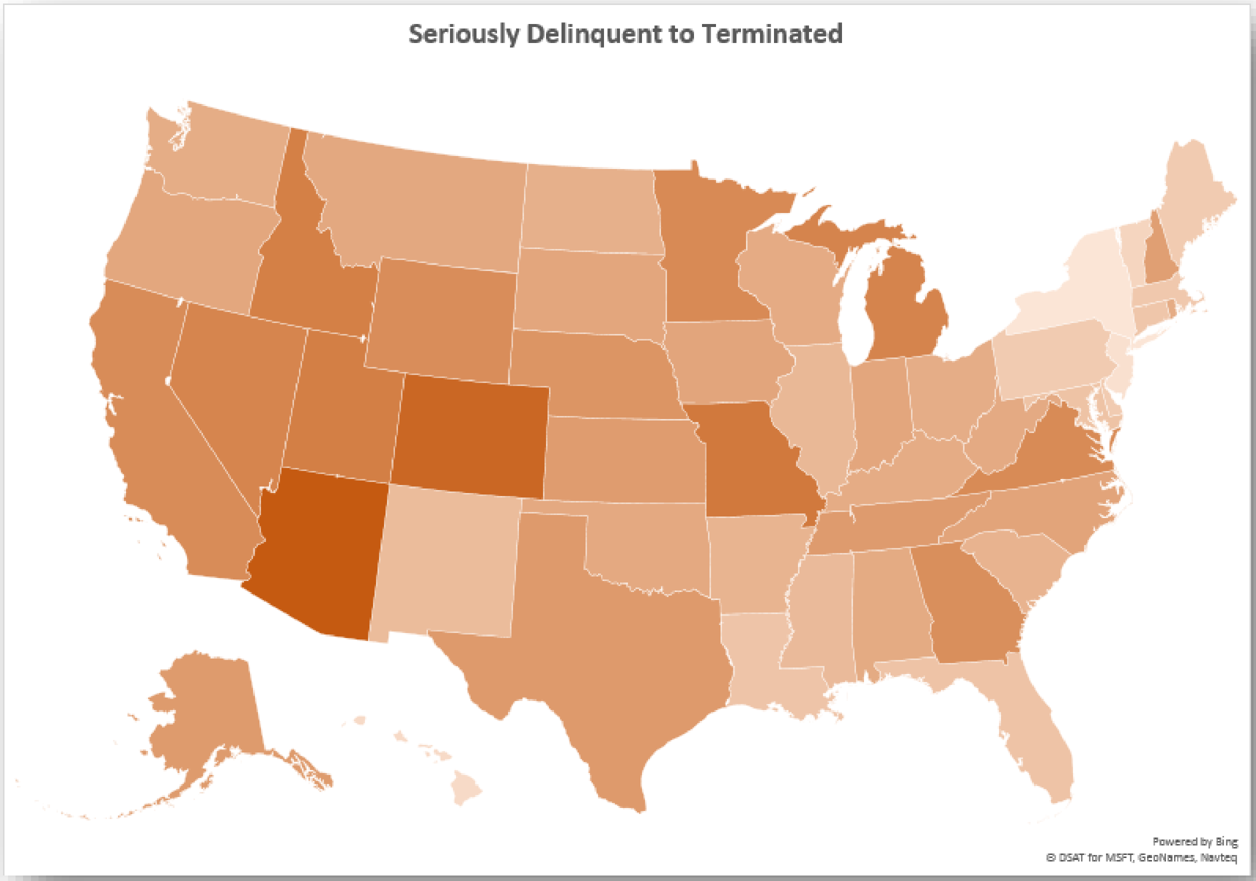

Seriously Delinquent to Terminated
Powered by Bing
© DSAT for MSFT, GeoNames, Navteq

Figure 24 Transition Probabilities for Each State

A judicial foreclosure state is a state that has court proceedings before a bank can foreclose a property. In non-judicial foreclosure states, foreclosures usually are processed without a court. Table 2 shows states ranking from least to highest probability for the first 15 states and their judicial status.

Table 2 Highest Probability of Foreclosures Ranked in Ascending Order

Position	States	Judicial
1	NY	Yes
2	NJ	Yes
3	HI	No
4	VT	Yes
5	DC	No
6	DE	Yes
7	PA	Yes
8	ME	Yes
9	MA	No
10	CT	Yes
11	LA	Yes
12	FL	Yes
13	MD	Yes
14	NM	Yes
15	MS	No

As suspected, 11 out of 15 are judicial states. These states have a significantly lower probability of foreclosure on a property. In fact, 21 out of the first 30 are judicial states. Considering there are only 27 judicial states in total, this finding is strongly supported.

7. Conclusions

One goal of the Risk Management team at Angelo Gordon is to build a sound statistical model to predict loan performance. Specifically, to predict if a loan will default and anticipate the progression of the loan to termination. Building the complete model was always outside the scope of this project. However, we were able build a basic statistical model to serve as a foundation for the model. The success in this project lies in that the sample analyses performed could serve as a check for all the accomplished work.

A formal meeting was arranged with all the key stakeholders at the end of this project. The business process tool (tech stack) was run in its entirety during the meeting. The meeting was fruitful as it allowed everyone to carefully check for any mistakes and formalized handing over the tech stack. All key stakeholders left the meeting with full confidence that they understood the tech stack. This ensured that the firm had complete process ownership which was previously lacking with third party vendors of data.

The data set provided was also a crucial deliverable that had to be correct and ready to use. This included running many sanity checks for various values such as average loan balances studying foreclosures. The data was also cleansed ensuring proper formatting for all field values. Additional fields were also added to the data set when necessary but was missing. Performing sample analyses allowed the risk team and others to understand how to use the data set for their benefit.

The firm was also new to many software tools that were being used for this project such as Data Lake or Spark. Operations such as format changes and export of data to local csv files were foreign to everyone. A Jupyter notebook with all the operations performed so far was delivered to the AG risk team as documentation. Not only will this be useful when building the comprehensive statistical model but the documentation will accelerate the learning curve for new stakeholders/employees if any.

8. Recommendations

The primary recommendation is to maintain the tech stack and the data set. The tech stack should be run every month to ensure that the most recent data is being fed to the model. Since the tech stack does not have a provision to download past months, running every month in a timely fashion is crucial.

The Selenium script of the tech stack also should be maintained carefully. The specific module is based on the CTSLink website. Since changes to the website cannot be predicted, one should be prepared to make changes to the script when necessary. However, no drastic change in the website is expected in the foreseeable future.

There is much potential for conducting more studies and further exploring the data for insights. Running the tech stack every month and adding more data from various open source data sets will allow the total data set to grow. A larger data set increases the reliability of predictions made and also accounts for a larger sample population. A total of 9.5 million loans may seem a lot but it is only a portion of the total loans. This means that Angelo Gordon has to pursue more data if it is to build a model to make investment decisions.

9. Reflection on the Project

9.1. Design Component

The Risk Management team at Angelo Gordon aims to build a comprehensive and precise statistical model to predict loan performance. The project's overall deliverable was to create a foundation for this model to begin building upon. This was achieved by designing a rudimentary statistical model using the Wells Fargo public data set. Elements of data analysis required knowledge of stochastic models, data manipulation in Excel, and strong understanding of data wrangling. When building the model, Markov matrices were used in understanding probability of various state changes. These state changes marked loan performance changes and were reflective of borrowers paying or defaulting their monthly payments. These state changes were later grouped by US States to understand performance varying across the country. Once completed, the data was exported to excel to be analyzed and gain insights such as most judicial states have the lowest probability for foreclosure on properties.

The Wells Fargo data set was used through Spark, an open source software that supports big data exploration. However, the data set had to be acquired from their website and processed before being able to begin data analysis. Angelo Gordon also required that the complete data set be updated with every month's loan information. This required designing and implementing a business process tool that could run every month by the risk team to update with new data. The project management aspect of implementing the process required that all key stakeholders have process ownership. Process ownership is key in implementing a sustainable project. This was achieved through weekly update meetings and delivering documentation about software package implementation, maintenance, and data analysis on spark.

9.2. Constraints and Alternatives

One of the most crucial limitations of this project was large wait times at each step of the process. The data accounted for 9.5 million loans which uses a total of 500 GB disk space. The download step of the process alone took 10 days. The download was I/O bound and no existing API meant that the download had to be done manually by clicking on a website. Later the data set had to be converted to .parq format which was CPU bound. Every step of the process took long wait times simply because of the size of the data set. This was overcome by various methods. For operations that were I/O bound, multiple operations were queued overnight and maintained remotely. For operations that were CPU bound, the risk team volunteered four of their high performing CPUs for computation. The operations were distributed amongst these CPUs to ensure effective allocation of time.

Another constraint was the sustainability of this project. The software package is designed to download the most recent month's data set for each shelf or loan issuer. Hence, the download needs to be run every month to update the data set. If not, the download needs to be done manually. The modularity of the software package allows for the rest of the process to run after the manual download. However, if the website were to be updated differently in any way, the Selenium script would need changes. To ensure sustainability, the risk team has been trained on the use of the business process tool. Documentation also has been provided containing instructions on how to make changes to the software package and the data analysis.

9.3. Need for Life-Long Learning

The project in the end was both challenging and fulfilling. Despite studying how to analyze data through lab work and class work, analysis of a data set of this size was challenging. The real-life application of the project was at first tough to grasp. However, the firm provided us with guidance and support throughout which facilitated understanding the financial context of this project. The project was unique in that it required an implementation of a process that simultaneously allowed for continual improvement of the precision of the data set. Increasing the data set monthly will allow more data points for the statistical model, improving precision of studies conducted.

One of the challenges was to be able to use big data analytics software Spark. Not having any experience in big data from coursework, the learning curve was very steep. Although Apache Spark is a great open source software, using it was quite difficult initially. The coding was done in PySpark which is an IPython based frontend API. Learning to use Spark was a fruitful challenge as the software is a very powerful tool.

The project showed the application of our majors in a light that we had not given much thought to previously. Many Industrial Engineering skills were transferrable to big data analysis that surprised me. The skills learned at Angelo Gordon may or may not be useful in whatever professional paths we take but the mentorship and guidance received was invaluable.

10. References

Angelo, Gordon & Co. "ANGELO, GORDON." Home : Angelo, Gordon & Co. January 22, 2006. Accessed January 22, 2018. https://www.angelogordon.com/.

Appleyard, Randy. "Overview of the Mortgage-Backed Securities Markets." October 1, 2004. Accessed January 20, 2018. https://warrington.ufl.edu/graduate/academics/msf/docs/speakers/presentation_Appleyard0405.pdf.

Beattie, Andrew. "Mortgage-Backed Security (MBS)." Investopedia. January 17, 2018. Accessed January 21, 2018. https://www.investopedia.com/terms/m/mbs.asp.

Fannie Mae. n.d. *Loan Performance Data.* Accessed 1 11, 2018. https://loanperformancedata.fanniemae.com/lppub/.

FINRA. "Mortgage-Backed Securities." Mortgage-Backed Securities | FINRA.org. January 18, 2018. Accessed January 21, 2018. http://www.finra.org/investors/mortgage-backed-securities.

Giesbrecht, Nathan. "Javascript Check all Checkboxes in Google Chrome or Firefox." Nathan Giesbrecht Winnipeg Web Design & Development. January 27, 2015. Accessed January 22, 2018. http://nathangiesbrecht.com/check-all-checkboxes-chrome-javascript.

IBM. 2002. *Clustering: A basic 101 tutorial.* April 03. Accessed 12 10, 2017. https://www.ibm.com/developerworks/aix/tutorials/clustering/clustering-pdf.pdf .

Intex. n.d. *Cashflow Analytics for Global Structured Finance.* Accessed 1 11, 2018. http://www.intex.com/main/.

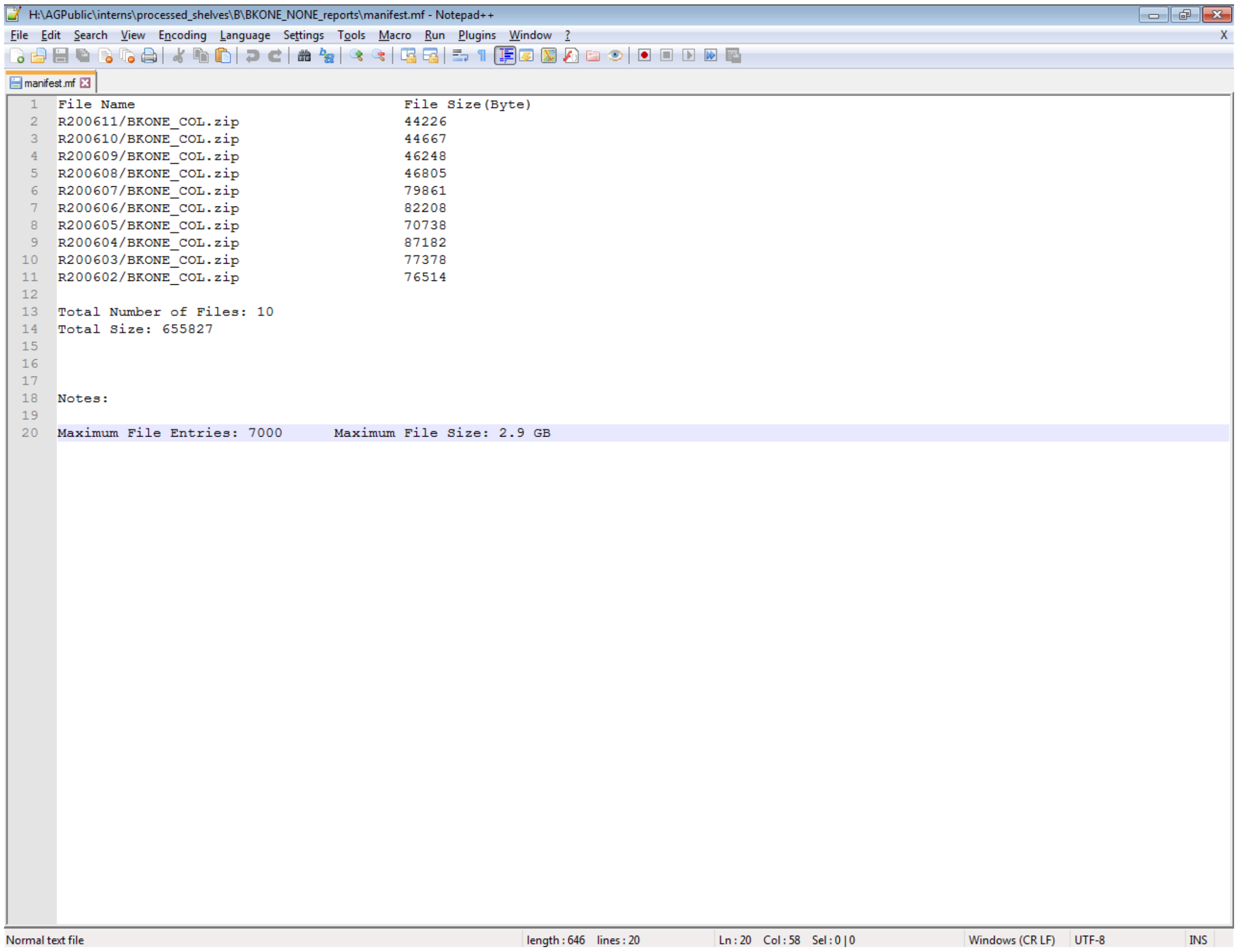
Staff, Investopedia. "Hedge Fund." Investopedia. August 23, 2017. Accessed January 22, 2018. https://www.investopedia.com/terms/h/hedgefund.asp.

Zaharia, Matei. "HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing." *USENIX*, June 22, 2010, 10. Accessed January 20, 2018. https://dl.acm.org/citation.cfm?id=1863103.1863113#.

11. Appendix

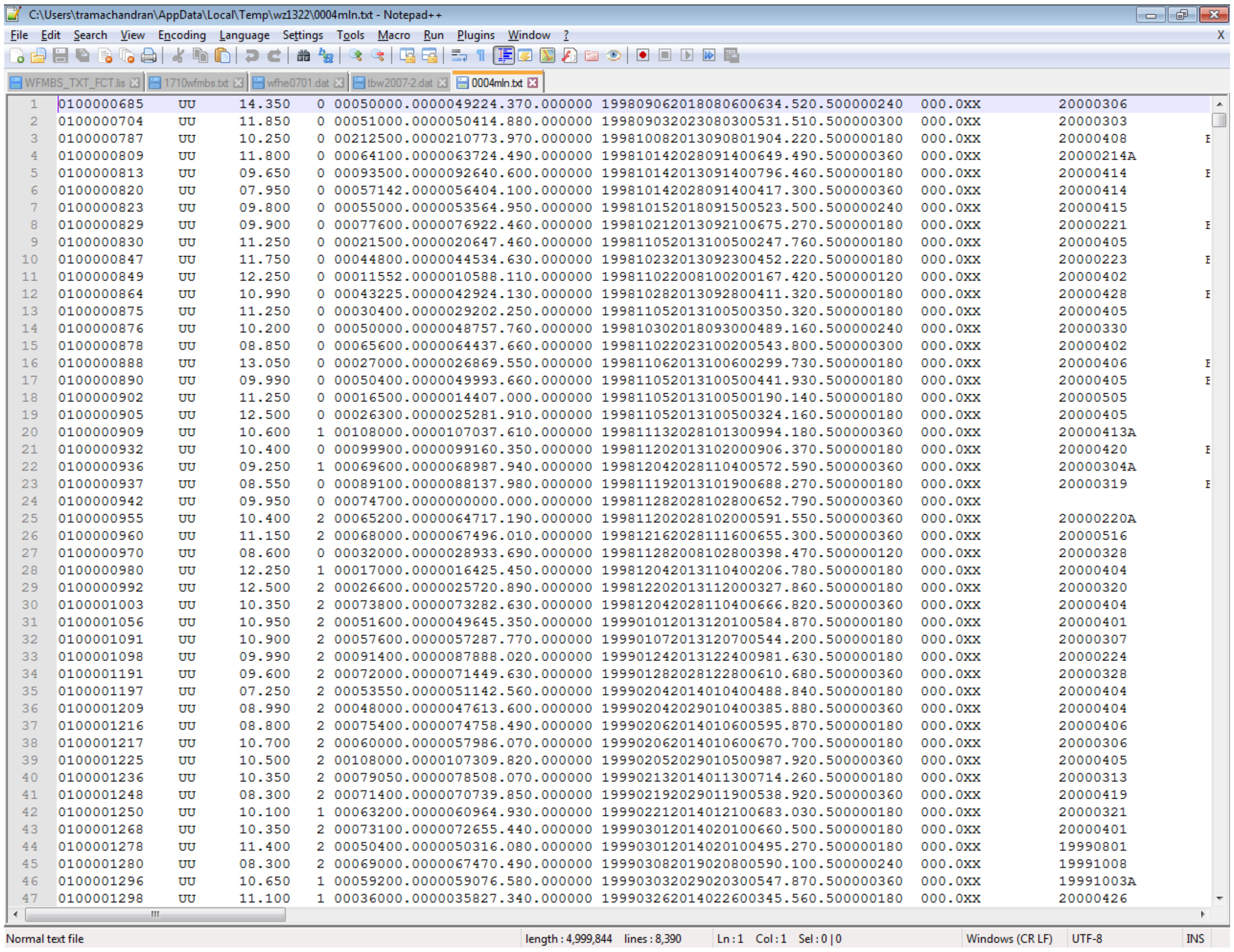
11.1. Sample Data

11.1.1. Manifest Files


H:\AGPublic\interns\processed_shelves\B\BKONE_NONE_reports\manifest.mf - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
manifest.mf
Normal text file
length : 646 lines : 20
Ln : 20 Col : 58 Sel : 0 | 0
Windows (CR LF) UTF-8 INS

```
File Name                                File Size(Byte)
R200611/BKONE_COL.zip                    44226
R200610/BKONE_COL.zip                    44667
R200609/BKONE_COL.zip                    46248
R200608/BKONE_COL.zip                    46805
R200607/BKONE_COL.zip                    79861
R200606/BKONE_COL.zip                    82208
R200605/BKONE_COL.zip                    70738
R200604/BKONE_COL.zip                    87182
R200603/BKONE_COL.zip                    77378
R200602/BKONE_COL.zip                    76514

Total Number of Files: 10
Total Size: 655827



Notes:

Maximum File Entries: 7000      Maximum File Size: 2.9 GB
```

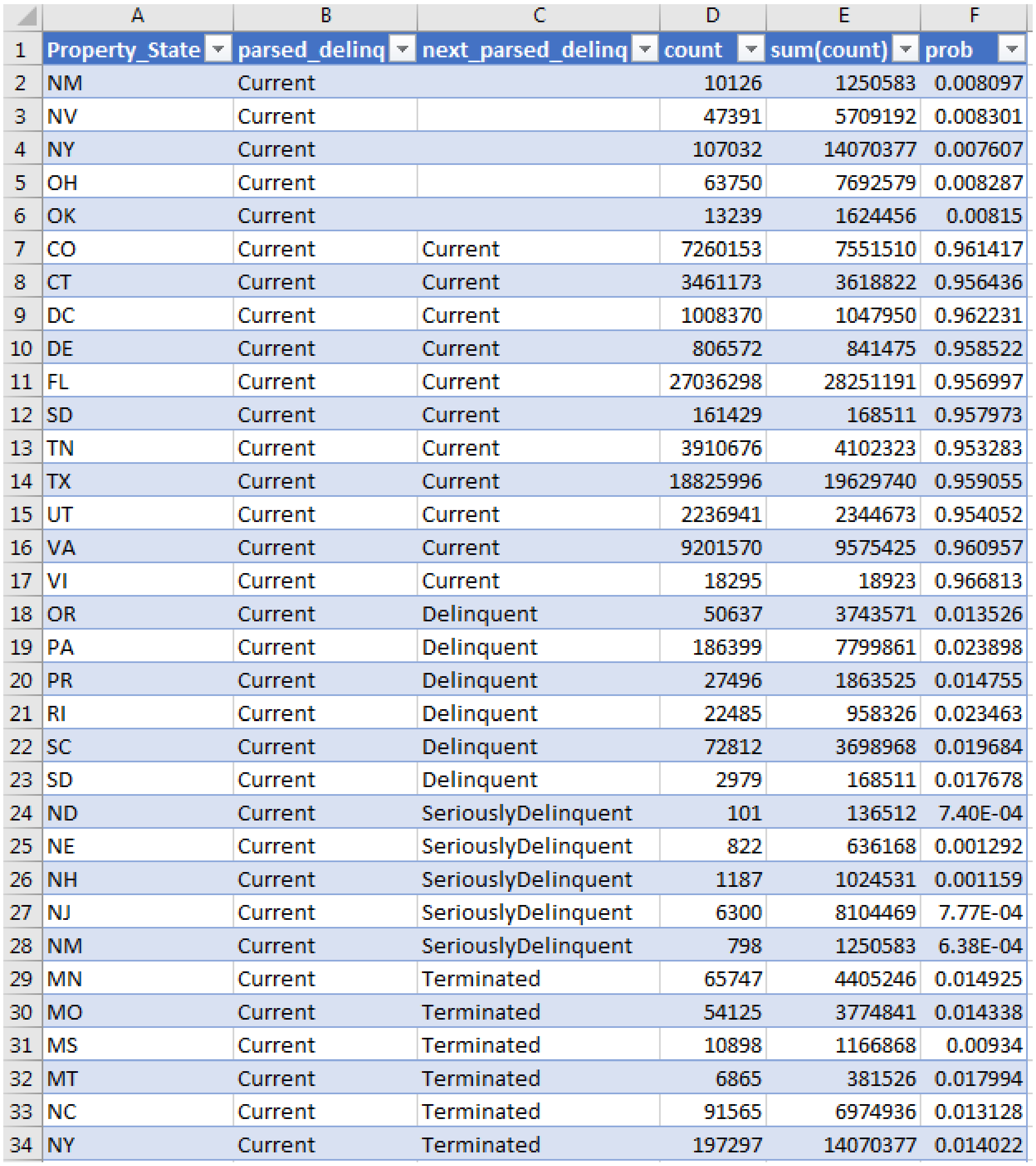
11.1.2. Sample Data file (.txt)


C:\Users\tramachandran\AppData\Local\Temp\wz1322\0004mln.txt - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
WFMBS_TXT_FCT.lis
1710wfmbs.txt
wfhe0701.dat
tbw2007-2.dat
0004mln.txt

```
0100000685    UU      14.350   0 00050000.0000049224.370.000000 19980906201808060063 4.520.500000240  000.0XX          20000306
0100000704    UU      11.850   0 00051000.0000050414.880.000000 19980903202308030053 1.510.500000300  000.0XX          20000303
0100000787    UU      10.250   0 00212500.0000210773.970.000000 19981008201309080190 4.220.500000180  000.0XX          20000408          E
0100000809    UU      11.800   0 00064100.0000063724.490.000000 19981014202809140064 9.490.500000360  000.0XX          20000214A
0100000813    UU      09.650   0 00093500.0000092640.600.000000 19981014201309140079 6.460.500000180  000.0XX          20000414          E
0100000820    UU      07.950   0 00057142.0000056404.100.000000 19981014202809140041 7.300.500000360  000.0XX          20000414
0100000823    UU      09.800   0 00055000.0000053564.950.000000 19981015201809150052 3.500.500000240  000.0XX          20000415
0100000829    UU      09.900   0 00077600.0000076922.460.000000 19981021201309210067 5.270.500000180  000.0XX          20000221          E
0100000830    UU      11.250   0 00021500.0000020647.460.000000 19981105201310050024 7.760.500000180  000.0XX          20000405
0100000847    UU      11.750   0 00044800.0000044534.630.000000 19981023201309230045 2.220.500000180  000.0XX          20000223          E
0100000849    UU      12.250   0 00011552.0000010588.110.000000 19981102200810020016 7.420.500000120  000.0XX          20000402
0100000864    UU      10.990   0 00043225.0000042924.130.000000 19981028201309280041 1.320.500000180  000.0XX          20000428          E
0100000875    UU      11.250   0 00030400.0000029202.250.000000 19981105201310050035 0.320.500000180  000.0XX          20000405
0100000876    UU      10.200   0 00050000.0000048757.760.000000 19981030201809300048 9.160.500000240  000.0XX          20000330
0100000878    UU      08.850   0 00065600.0000064437.660.000000 19981102202310020054 3.800.500000300  000.0XX          20000402
0100000888    UU      13.050   0 00027000.0000026869.550.000000 19981106201310060029 9.730.500000180  000.0XX          20000406          E
0100000890    UU      09.990   0 00050400.0000049993.660.000000 19981105201310050044 1.930.500000180  000.0XX          20000405          E
0100000902    UU      11.250   0 00016500.0000014407.000.000000 19981105201310050019 0.140.500000180  000.0XX          20000505
0100000905    UU      12.500   0 00026300.0000025281.910.000000 19981105201310050032 4.160.500000180  000.0XX          20000405
0100000909    UU      10.600   1 00108000.0000107037.610.000000 19981113202810130099 4.180.500000360  000.0XX          20000413A
0100000932    UU      10.400   0 00099900.0000099160.350.000000 19981120201310200090 6.370.500000180  000.0XX          20000420          E
0100000936    UU      09.250   1 00069600.0000068987.940.000000 19981204202811040057 2.590.500000360  000.0XX          20000304A
0100000937    UU      08.550   0 00089100.0000088137.980.000000 19981119201310190068 8.270.500000180  000.0XX          20000319          E
0100000942    UU      09.950   0 00074700.0000000000.000.000000 19981128202810280065 2.790.500000360  000.0XX
0100000955    UU      10.400   2 00065200.0000064717.190.000000 19981120202810200059 1.550.500000360  000.0XX          20000220A
0100000960    UU      11.150   2 00068000.0000067496.010.000000 19981216202811160065 5.300.500000360  000.0XX          20000516
0100000970    UU      08.600   0 00032000.0000028933.690.000000 19981128200810280039 8.470.500000120  000.0XX          20000328
0100000980    UU      12.250   1 00017000.0000016425.450.000000 19981204201311040020 6.780.500000180  000.0XX          20000404
0100000992    UU      12.500   2 00026600.0000025720.890.000000 19981220201311200032 7.860.500000180  000.0XX          20000320
0100001003    UU      10.350   2 00073800.0000073282.630.000000 19981204202811040066 6.820.500000360  000.0XX          20000404
0100001056    UU      10.950   2 00051600.0000049645.350.000000 19990101201312010058 4.870.500000180  000.0XX          20000401
0100001091    UU      10.900   2 00057600.0000057287.770.000000 19990107201312070054 4.200.500000180  000.0XX          20000307
0100001098    UU      09.990   2 00091400.0000087888.020.000000 19990124201312240098 1.630.500000180  000.0XX          20000224
0100001191    UU      09.600   2 00072000.0000071449.630.000000 19990128202812280061 0.680.500000360  000.0XX          20000328
0100001197    UU      07.250   2 00053550.0000051142.560.000000 19990204201401040048 8.840.500000180  000.0XX          20000404
0100001209    UU      08.990   2 00048000.0000047613.600.000000 19990204202901040038 5.880.500000360  000.0XX          20000404
0100001216    UU      08.800   2 00075400.0000074758.490.000000 19990206201401060059 5.870.500000180  000.0XX          20000406
0100001217    UU      10.700   2 00060000.0000057986.070.000000 19990206201401060067 0.700.500000180  000.0XX          20000306
0100001225    UU      10.500   2 00108000.0000107309.820.000000 19990205202901050098 7.920.500000360  000.0XX          20000405
0100001236    UU      10.350   2 00079050.0000078508.070.000000 19990213201401130071 4.260.500000180  000.0XX          20000313
0100001248    UU      08.300   2 00071400.0000070739.850.000000 19990219202901190053 8.920.500000360  000.0XX          20000419
0100001250    UU      10.100   1 00063200.0000060964.930.000000 19990221201401210068 3.030.500000180  000.0XX          20000321
0100001268    UU      10.350   2 00073100.0000072655.440.000000 19990301201402010066 0.500.500000180  000.0XX          20000401
0100001278    UU      11.400   2 00050400.0000050316.080.000000 19990301201402010049 5.270.500000180  000.0XX          19990801
0100001280    UU      08.300   2 00069000.0000067470.490.000000 19990308201902080059 0.100.500000240  000.0XX          19991008
0100001296    UU      10.650   1 00059200.0000059076.580.000000 19990303202902030054 7.870.500000360  000.0XX          19991003A
0100001298    UU      11.100   1 00036000.0000035827.340.000000 19990326201402260034 5.560.500000180  000.0XX          20000426
```

Normal text file   length : 4,999,844   lines : 8,390   Ln : 1   Col : 1   Sel : 0 | 0   Windows (CR LF)   UTF-8   INS

11.1.3. Sample Excel File



	A	B	C	D	E	F
1	Property_State	parsed_delinq	next_parsed_delinq	count	sum(count)	prob
2	NM	Current		10126	1250583	0.008097
3	NV	Current		47391	5709192	0.008301
4	NY	Current		107032	14070377	0.007607
5	OH	Current		63750	7692579	0.008287
6	OK	Current		13239	1624456	0.00815
7	CO	Current	Current	7260153	7551510	0.961417
8	CT	Current	Current	3461173	3618822	0.956436
9	DC	Current	Current	1008370	1047950	0.962231
10	DE	Current	Current	806572	841475	0.958522
11	FL	Current	Current	27036298	28251191	0.956997
12	SD	Current	Current	161429	168511	0.957973
13	TN	Current	Current	3910676	4102323	0.953283
14	TX	Current	Current	18825996	19629740	0.959055
15	UT	Current	Current	2236941	2344673	0.954052
16	VA	Current	Current	9201570	9575425	0.960957
17	VI	Current	Current	18295	18923	0.966813
18	OR	Current	Delinquent	50637	3743571	0.013526
19	PA	Current	Delinquent	186399	7799861	0.023898
20	PR	Current	Delinquent	27496	1863525	0.014755
21	RI	Current	Delinquent	22485	958326	0.023463
22	SC	Current	Delinquent	72812	3698968	0.019684
23	SD	Current	Delinquent	2979	168511	0.017678
24	ND	Current	SeriouslyDelinquent	101	136512	7.40E-04
25	NE	Current	SeriouslyDelinquent	822	636168	0.001292
26	NH	Current	SeriouslyDelinquent	1187	1024531	0.001159
27	NJ	Current	SeriouslyDelinquent	6300	8104469	7.77E-04
28	NM	Current	SeriouslyDelinquent	798	1250583	6.38E-04
29	MN	Current	Terminated	65747	4405246	0.014925
30	MO	Current	Terminated	54125	3774841	0.014338
31	MS	Current	Terminated	10898	1166868	0.00934
32	MT	Current	Terminated	6865	381526	0.017994
33	NC	Current	Terminated	91565	6974936	0.013128
34	NY	Current	Terminated	197297	14070377	0.014022

11.2. RecurShelfDownload

```
def RecurShelfDownload(browser, shelves_processed, download_limit):
    """
    This function exists because Selenium's DOM becomes stale as soon as
the page changes.
    In order to keep track of the running list of which shelves we've
visited, this function iterates recursively rather
    than through a normal loop. This allows us to refresh the DOM each
time the function is caused.
    It will also automatically dismiss any popups that it encounters.
    """
    shelves = browser.find_by_text("Shelf Documents")
    if len(shelves) == len(shelves_processed):
        return
    if download_limit is not None:
        #user has placed an arbitrary limit on the number of files that
can be downloaded
        if len(shelves_processed) >= download_limit:
            print("Stopping download as the script has reached the
download limit of " + str(download_limit))
            return
    for shelf in shelves:
        if shelf['href'] in shelves_processed:
            continue
        shelves_processed.append(shelf['href'])
        shelf.click()
        try:
            checkboxes = browser.find_by_id('documentChkBx')
            checkboxes.first.check()
        except:
#We use this catch-all exception here because some shelves are terminated
#This can cause errors in Splinter that manifest in a variety of
#unpredictable ways, so we catch all exceptions to reduce the likelihood
of #failed ingestion.
            print("Skipping suspected terminated shelf")
        try:
    # There is no common sense ordering of shelf files, so we will
download them all and filter them later.
            # Most of the files are negligible in size.
            browser.find_by_name('zip').first.click()
            browser.back()
        except UnexpectedAlertPresentException:
            alert = browser.get_alert()
            alert.dismiss()
            browser.back()
        except ElementDoesNotExist:
            print("ERROR: no zip button found!")
            browser.back()
        print("Shelves processed: " + str(shelves_processed))
        break
 #Selenium invalidates any objects that are in the DOM, so we really
 #only want the first entry that we have not already computed- hence break
after the first iteration.
    RecurShelfDownload(browser, shelves_processed, download_limit)
```

11.3. Unzipper

```
def execute_unzip(rootDirectory, outputDirectory):
    """
    Unzips all file in rootDirectory and places their output in
outputDirectory.
    """
    file_count = 0
    for root, dirs, files in os.walk(rootDirectory):
        for file in files:
            path = os.path.join(root, file)
            print("Now extracting: " + file)
            print('target path: ' + path)
            print("File count " + str(file_count))
            if zipfile.is_zipfile(path):
                zip_ref = zipfile.ZipFile(path, 'r')
                # ensure it does not exist already, if so, skip
                infolist = zip_ref.infolist()
                for info in infolist:
                    if os.path.exists(os.path.join(root, info.filename)):
                        print("File " + info.filename + " exists and
therefore will be skipped during processing.")
                        zip_ref.close()
                        continue
                zip_ref.extractall(outputDirectory)
                zip_ref.close()
            file_count += 1
    print("Extraction complete.")
```

11.4. Shelf file processor

```
def process_single_line(inputLine, dt, shelfname):
    """
    This function parses a line of the arbitrary Wells Fargo Collateral File.
    You can find more information about the format on CTSLink under
Additional Services --> File Layouts
    """
    line = "{:<1063}".format(inputLine)  # pad to max length with whitespace
    out = {}
    # metadata from filename
    out['shelfname'] = shelfname  # the shelfname from CMSLink
    out['shelf_date'] = dt.strftime("%Y%m%d")  # the datettime this loanmonth
is for
    out['shelf_month'] = dt.strftime("%m")
    out['shelf_year'] = dt.strftime("%Y")
    # Raw positional formatting
    out['Loan_Number'] = line[0:12]
    out['Property_Type_Code'] = line[12:14]
    out['Owner_Occupied_Code'] = line[14]
    out['Purpose_Code'] = line[15:17]
    out['Leasehold_ID'] = line[17]
    out['Account_or_Note_Type_Code'] = line[18:20]
    out['No_Ratio_ID'] = line[20]
    out['Current_Interest_Rate'] = line[21:27]
    out['Investor_ID'] = line[27:31]
    out['Pool_Number'] = line[30:32]
    out['Original_Balance'] = line[32:43]
    out['Ending_Scheduled_Balance'] = line[43:54]
    out['Fixed_Retained_Yield_Rate'] = line[54:62]
    out['Foreign_National_Code'] = line[62]
    out['First_Payment_Date'] = line[63:71]
    out['Maturity_Date'] = line[71:79]
    out['Current_PI_Constant'] = line[79:87]
    out['Servicing_Fee'] = line[87:95]
    out['Original_Term_in_months'] = line[95:98]
    out['Foreclosure_Bankruptcy_REO_Indicator'] = line[98:100]
    out['Original_LTV_Ratio'] = line[100:105]
    out['Property_State'] = line[105:107]
    out['ECS_Score_Version'] = line[107]
    out['ECS_Score_Raw'] = line[108:114]
    out['ECS_Score_Code'] = line[114:116]
    out['Next_Due_Date'] = line[116:124]
    out['Adjustable_Rate_Mortgage_ARM_Indicator'] = line[124]
    out['Program_Code'] = line[125:127]
    out['Credit_Grade'] = line[127:129]
    out['Channel_Code'] = line[129:132]
    out['Relocation_Indicator'] = line[132]
    out['Balloon_Indicator'] = line[133]
    out['Lien_Status'] = line[134]
    out['Original_Appraisal_Value'] = line[135:146]
    out['Prepayment_Penalty_Indicator'] = line[146:149]
    out['Interest_Collection_Code'] = line[149]
    out['Original_Interest_Rate'] = line[150:156]
    out['Index_Code'] = line[156:158]
    out['Margin'] = line[158:163]
```

```
out['Next_Interest_Rate_Change_Date'] = line[163:171]
out['Next_Payment_Change_Date'] = line[171:179]
out['Interest_Rate_Adjustment_Frequency'] = line[179:182]
out['Payment_Adjustment_Frequency'] = line[182:185]
out['Periodic_Rate_Cap'] = line[185:190]
out['Periodic_Payment_Cap_Percentage'] = line[190:196]
out['Life_Maximum_Interest_Rate_Ceiling'] = line[196:202]
out['Lifetime_Rate_Floor'] = line[202:208]
out['Issue_PI'] = line[208:216]
out['Original_Index_Value'] = line[216:222]
out['Negative_Amortization_Code'] = line[222]
out['Interest_Rate_at_Next_Reset_Date'] = line[223:229]
out['ARM_Convertibility_Code'] = line[229]
out['Property_City'] = line[287:298]
out['Property_Zip_Code'] = line[298:303]
out['PMI_Insurer_Code'] = line[311:313]
out['Origination_Date_or_Note_Date'] = line[313:321]
out['FICO_Raw_Score_Number'] = line[321:325]
out['Product_Type_Code'] = line[325:328]
out['CLTV'] = line[328:334]
out['Sale_Balance'] = line[334:345]
out['Document_Type_Code'] = line[345:347]
out['Issue_Year'] = line[347:351]
out['Series'] = line[351:354]
out['Loss_on_Liquidated_Property'] = line[354:365]
out['Actual_Current_Balance'] = line[365:376]
out['Prepayments_in_Full'] = line[376:387]
out['Partial_Prepayments'] = line[387:398]
out['Paid_in_Full_Effective_Date'] = line[398:406]
out['Issuer_Identification'] = line[406:414]
out['Servicer_Number'] = line[414:420]
out['Master_Servicing_Fee'] = line[420:428]
out['Trustee_Fee'] = line[428:436]
out['Pool_Insurance_Fee'] = line[436:444]
out['Special_Hazard_Fee'] = line[444:452]
out['Spread_1_Fee'] = line[452:460]
out['Spread_2_Fee'] = line[460:468]
out['Property_County'] = line[468:486]
out['Mortgagor_Employer_Name'] = line[486:511]
out['Subsidy_Code'] = line[511:517]
out['LEX_Lender_Identification'] = line[517:522]
out['Client_Code_Identification'] = line[522:527]
out['Senior_Lien'] = line[539:550]
out['Certificate_Administration_Fee'] = line[550:558]
out['Paid_Off_Remittance_Cycle'] = line[558:564]
out['Repurchase_Date'] = line[564:572]
out['Substituted_Loan_Number'] = line[572:584]
out['Remittance_Cycle'] = line[584:590]
out['Pledged_Asset_Mortgage_Indicator'] = line[590]
out['Group_ID'] = line[591:593]
out['Current_FICO_Score_Nbr'] = line[593:597]
out['Prepayment_Penalty_Amount'] = line[597:609]
out['Prepayment_Penalty_Waived_Amount'] = line[609:621]
out['Modification_Date'] = line[621:629]
out['Substitution_Date'] = line[629:637]
out['Issuance_Balance'] = line[637:649]
out['Losses_on_Previously_Liquidated_Loans'] = line[649:661]
```

```
    out['Servicer_Name'] = line[661:761]
    out['Arm_Rate_Life_Cap'] = line[761:768]
    out['Arm_Negative_Amortization_Cap'] = line[768:776]
    out['Arm_Round_Factor'] = line[776:783]
    out['Arm_Teaser_Period'] = line[783:786]
    out['Pay_Teaser_Period'] = line[786:789]
    out['Number_of_Arm_Look_Back_Days'] = line[789:792]
    out['Interest_Only_Original_Term'] = line[792:795]
    out['Originator_Name'] = line[795:895]
    out['Interest_Forgiveness_Amount'] = line[895:906]
    out['Expense_Forgiveness_Amount'] = line[906:917]
    out['Principal_Forgiveness_Amount'] = line[917:928]
    out['Total_Capitalized_Amount'] = line[928:939]
    out['Balloon_Date'] = line[939:947]
    out['Balloon_Payment_Amount'] = line[947:958]
    out['Modified_Next_Payment_Adjust_Date'] = line[958:966]
    out['Modified_Next_Interest_Rate_Adjust_Date'] = line[966:974]
    out['ARM_to_Fixed_Conversion'] = line[974]
    out['Fixed_to_ARM_Conversion'] = line[975]
    out['IO_to_Fully_Amortized_Conversion'] = line[976]
    out['Fully_Amortized_to_IO_Conversion'] = line[977]
    out['Segmentation'] = line[978:984]
    out['Temporary_Modification'] = line[984]
    out['Ending_Scheduled_Interest_Bearing_Balance'] = line[985:996]
    out['Ending_Actual_Interest_Bearing_Balance'] = line[996:1007]
    out['Ending_Scheduled_TDO_Balance'] = line[1007:1018]
    out['Ending_Actual_TDO_Balance'] = line[1018:1029]
    out['Non_Interest_Bearing_Deferred_Principal_Bal'] = line[1029:1040]
    out['Principal_Reduction_Alternative_Forbearance_Balance'] =
line[1040:1051]
    out['Non_Interest_Bearing_Treatment_Methodology'] = line[1051:1061]
```