Quantitative Research Modeling Library

Final Project Report

Team
sdMay19-06

Team Members
Josiah Anderson -- Meeting Facilitator
Doh Yun Kim -- Scribe
Gabriel Klein -- Report Manager
Drake Mossman -- Communication Manager
Jacob Richards -- Quality Assurance
Nathan Schaffer -- Overseer

Client
Joseph Byrum
(Principal Financial Group)

Advisor
Srikanta Tirthapura

Contact
sdmay19-06@iastate.edu
https://sdmay19-06.sd.ece.iastate.edu

Last Updated
29 April 2019

Table of Contents

Table of Contents **1**

List of Figures **3**

List of Tables **4**

Executive Summary **5**

1 Requirements Specification **6**
1.1 Functional Requirements 6
1.2 Non-Functional Requirements 6

2 Development Process **6**
2.1 Chosen Process 6
2.2 Rationale 7

3. Design Plan **8**
3.1 Use Cases 8
3.2 Design Process 8
3.3 Big-Picture Design 9
3.4 Design Objectives, Constraints, and Trade-offs 10
3.5 Implementation and Technologies Used 12
3.5.1 Implementation Diagram 12
3.5.2 Technologies and Software Used 12
3.5.3 Rationale for Technology and Software Choices 13
3.6 Modules 13
3.6.1 Aggregation 14
3.6.2 Predictions 14
3.6.3 Stock Scoring 15
3.6.4 Database 16
3.7 Applicable Standards and Best Practices 17

4. Test Plan **18**
4.1 Overall Plan 18
4.2 Unit Testing 18
4.3 Interface Testing 19

4.4 System Integration Testing 19
4.5 Use Case Testing 19
5 Testing Evaluation 19
5.1 Test Case Evaluation 19
5.2 Validation and Verification 20
6 Project Management 21
6.1 Roles and Responsibilities 21
6.2 Project Schedule 22
6.3 Risk and Mitigation 24
6.4 Lessons Learned 26
7 Conclusions 27
7.1 Closing Remarks 27
7.2 Future Work 27
References 29
Appendix A: Team Information 30
Josiah Anderson 30
Doh Yun Kim 30
Gabriel Klein 31
Drake Mossman 31
Jacob Richards 32
Nathan Schaffer 32

List of Figures

Figure 1: DRP 2.0 System Architecture Diagram
Figure 2: Implementation Diagram
Figure 3: Aggregation Module Diagram
Figure 4: Predictions Module Diagram
Figure 5: Stock Scoring Module Diagram
Figure 6: Database Schemas and Relationships

List of Tables

Table 1: Aggregation Benchmarking Results
Table 2: Project Schedule Planned Version
Table 3: Project Schedule Actual Outcome
Table 4: Risk Description Chart
Table 5: Risk Consequences Mapping
Table 6: Risk Likelihood Mapping
Table 7: Risk Mapping from Likelihood and Consequence to Severity
Table 8: Risk Severity Mapping

Executive Summary

The Global Equities Team at Principal Financial uses a process for developing stock performance prediction models called the Dynamic Risk Premium or DRP. Our team was tasked with evaluating the current state of the DRP from a software engineering perspective, and to develop potential solutions to the shortcomings that are present in the Dynamic Risk Premium. Through a thorough investigation into the process, and a series of interviews with users of the DRP at Principal, we have discovered three major areas that could be improved: consistency, efficiency, and transparency. Multiple projects are currently working on the DRP from across the country, which leads to inconsistencies in the interfaces between different modules of the process. Also, each time a potential user wants to implement a new model with the Dynamic Risk Premium they must start from scratch which leads to inefficiency and redundant coding. Finally, the way the current DRP process works is a black box to most observers which leads to confusion about what is really going on under the hood.

We worked with members of the Principal team to develop a solution to these problems that we will refer to as the DRP 2.0. The DRP 2.0 is a library of Python functions and classes that work together to give structure, standardization, and reusability to the quantitative research pipeline at Principal. Certain points of this pipeline have been intentionally exposed for extensibility using Abstract Base Classes in python. Once the user has implemented the necessary components, they can then generate a script that will import the DRP 2.0 library and call the functions for each module of the pipeline. Each of these functions then saves its output data in the appropriate table within a postgreSQL database on server managed by the team at Principal.

This design plan increases efficiency and consistency by providing an implemented function for factor portfolio aggregation to be used throughout Principal's research teams. Through the use of abstract base classes it also produces clarity in the interfaces between modules of the process. Since, the results from each stage of the pipeline are stored in the database, Power BI dashboards can be used to visualize what is being accomplished throughout the pipeline. This will address the need for transparency that currently exists in the Dynamic Risk Premium process.

1 Requirements Specification

1.1 Functional Requirements

1. The pipeline will consume raw stock level data from an AWS Postgres database
2. The pipeline will be able to aggregate data to factor portfolios
3. The pipeline will be able to build models on factor portfolios, generate predictions, and calculate model performance
4. The pipeline will be able to score stocks using a factor policy
5. The pipeline will be able to simulate portfolio returns
6. Users will be able to customize the stock universe, factor portfolio strategies, model algorithms, and factor policy of the DRP 2.0 pipeline
7. Each function of the pipeline shall be able to be called and run independently
8. The pipeline will be able to handle missing or invalid stock level data
9. It will be able to interface with the DRP 2.0 dashboards

1.2 Non-Functional Requirements

1. (*Performance*) The pipeline will be able to handle up to several GB of data
2. (*Performance*) The pipeline should be able to handle millions of observations and hundreds of factors
3. (*Maintainability*) The pipeline will have documented and standardized inputs and outputs for each of its functions and interfaces
4. (*Usability*) The pipeline interfaces will be intuitive to a novice data scientist
5. (*Accessibility*) The pipeline will be useable by novice programmers
6. (*Security*) The pipeline will not leak confidential data to outside sources
7. (*Compatibility*) The pipeline will be able to perform the same functionality as provided existing scripts
8. (*Transparency*) Individual runs of the pipeline will be able to be traced back to their inputs along with the modules used for that particular run

2 Development Process

2.1 Chosen Process

The process we used was closest to an agile development process. The key points of Agile we adopted into our development process were early and continuous feedback from the customer,

a constantly reprioritized Kanban board, face-to-face interactions, and a product vision statement to drive goal-oriented thinking.

Our vision statement grew in formality and precision as we understood the project domain more and were able to better articulate the end goal we were striving for, yet it kept the same core mentality through the entirety of the project: We want to provide Principal's Quant Research team with a tool that standardizes interfaces and helps avoid work duplication while anticipating future changes to aid in efficient and fast development of predictive equity models. While our ideas about the presentation and implementation of this project changed many times over the months we worked on it, this end-goal mentality helped keep us centered.

We used GitLab's Kanban-style issue board to manage tasks. This central location for documenting progress, future improvements, and even technical issues, which we added a section for, was vital to our success. Even before we were able to start writing working code, it helped us think of how we could take small steps toward our end goal and keep each other accountable to that. Early on, the tasks we made included interviews and other investigative efforts to help us grasp Principal's workflow and inefficiencies. Then as we started making our initial prototypes, we were able to break up tasks to assign to each of our members, putting the responsibility on them to update their own progress as they worked.

Within the first week of having a project assignment, we also began to set up weekly face-to-face amongst our team to review our progress as well as weekly meetings with our clients at Principal to get feedback and make sure we were on the right track. While it wasn't easy to coordinate so many busy and conflicting schedules to this extent, the result was worth it. These meetings were an essential time for us to realign our short-term goals with each other and with the team at Principal while also communicating about potential new risks or obstacles we foresaw. Because of the uncertain nature of our project, we anticipated the need to deliver multiple iterations of code to our client for feedback, which we planned for in our Gantt chart and followed through with throughout our process. The constant feedback we were able to get in return was what allowed us to have as successful a project as we did.

2.2 Rationale

The Waterfall method is sequential and structured. The development process is broken up into phases and development on the next phase doesn't begin until the current phase is finished. Because this method is very rigid and structured, the scope and requirements of the project don't change once development starts.

On the other hand, the Agile method is flexible. The development process for this method is iterative, this means that planning, development, testing, or other development phases may appear multiple times. Because phases can appear more than once, requirements are prone to change throughout the project lifecycle.

We chose the Agile process because of our awareness of the many unknowns and risks in our project. At the beginning of our first semester of senior design, none of our team had any experience with data science, the financial market, or the quantitative research methods Principal was using in this Dynamic Risk Premium 2.0 venture. This high level of uncertainty would have made a more linear process such as waterfall project management infeasible. Instead, with our more iterative project management approach, we came up with concrete, short-term tasks from the very beginning to help us as quickly as possible get up to speed with this entirely new domain and project.

3. Design Plan

3.1 Use Cases

1. A user wishes to aggregate a factor portfolio from a database of raw stock level data
2. A user wishes to train a predictive model based on a factor portfolio for predicting stock returns over various periods
3. A user wishes to score stocks' relative performance based on the predicted stock returns of a factor portfolio
4. A user wishes to run the entire pipeline to score the universe of stocks starting with a list of parameters for factor portfolios
5. A user wishes to visualize data from previous runs of the pipeline through an interface with the database
6. A user wishes to trace back a particular run of the pipeline to find out what factor portfolios, models, factor policies, and stock scorers were used.

3.2 Design Process

Our team began our design process by spending a significant amount of time researching our client's situation to establish their needs. We were all completely new to almost everything data science related, so we carried out several interviews with our client's employees to build our understanding of the circumstances we were designing a solution for. Once we'd aggregated the results of our interviews we discovered that our client has a workflow in serious need of automation for the less skilled employees, as well as for the many students and interns they work with. Specifically it would be helpful if we could automate any of the following tasks:

- Retrieving and formatting data
- Preprocessing data for the model
- Constructing accurate predictive models
- Postprocessing the model for validation
- Visualizing the result
- Storing data in a database for diagnostics

Along with automation, another feature employees desired was modularity. The employees we interviewed wanted to be able to separate the different stages of their process from each other as well as standardize the interfaces between them for ease with diagnostics and future development.

At this point we began to conceptualize possible solutions for our client's problems. We drafted a few proposals and communicated back and forth with our client several times through web conferences as well as in-person meetings to solidify our understanding of the necessary data science concepts as well as our concept of what the project should look like. We went through several prototype concepts before we landed on our current vision of what we now call the Dynamic Risk Premium (DRP) 2.0 pipeline.

3.3 Big-Picture Design

Our big-picture design for the pipeline is a 3-layer architecture split perpendicularly by 4 separate modules with different purposes. See Figure 1 where each horizontal row represents one layer of the architecture and each color represents a different module with its own purpose.

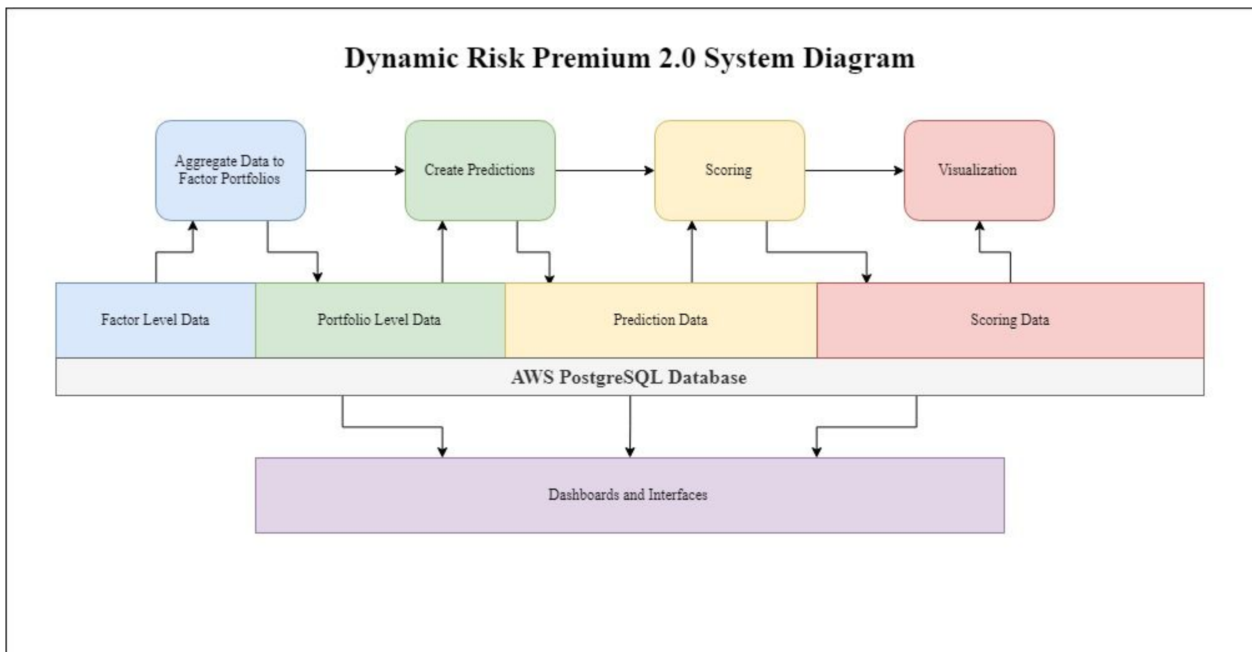


Figure 1: DRP 2.0 System Architecture Diagram

The processing layer of the architecture and the highest row in the diagram is a library that runs locally on the user's computer. This layer is where all of the data processing happens from aggregating factor portfolios to making models to predicting stock returns to scoring the universe of stocks. This layer can almost run entirely independently from the rest of the architecture, except for it needing the initial raw stock level data from the database (in the data layer) to get started.

The data layer of the architecture is the database along with all of its schemas and views. The database will contain all of the raw stock level data needed for the data processing to consume. This data is expected to come from FactSet, a service providing financial data for our client. Additionally it will provide a space to save and load intermediate results as represented by the several arrows back and forth between the two layers. This will allow for partial runs of the pipeline as well as the ability to inspect said intermediate results.

Finally, the visualization layer of the architecture consists of the dashboards and other interfaces our client would like to use to inspect and visualize the data in the database. This layer will not be implemented by us, but we will need to provide the interfaces through which this layer can interact with the data layer.

Now discussing the vertical slices of the pipeline, we split it up into three major sections: Aggregation, Predictions, and Scoring. Each of these will be discussed in more detail in the modules section below, but we give a brief description of each here as well.

The aggregation section takes care of aggregating raw stock level data to factor portfolios. The predictions section trains models based on the portfolios and makes predictions on future stock returns. Finally, the scoring section decides on a factor policy for the portfolios and then scores the universe of stocks according to that policy. Each module also stores the outputs of each operation in the data layer as they're calculated.

Since each of these three major modules are decoupled as much as possible, they will be available to run individually as well as as a group. The database will store the outputs of each stage so that any later stage can pick up where the others left off at any time and with any data. On the other hand, the modules will also return their results locally in code as well so that the pipeline doesn't need to waste time storing and then immediately retrieving results.

3.4 Design Objectives, Constraints, and Trade-offs

Our design objectives all stem from our vision statement: to provide Principal's Quant Research team with a tool that standardizes interfaces and helps avoid work duplication while anticipating future changes to aid in efficient and fast development of predictive equity models. We break down our more specific objectives for this tool into four primary categories: convenience, traceability, modularity, and adaptability.

One of the primary motivating factors for the DRP 2.0 project is that, previously, data scientists in Principal's Quantitative Research group often would end up using up a significant amount of time duplicating efforts for common tasks in their research of effective predictive equity models. For example, no matter what stage of that process they are working on, it is necessary to aggregate raw stock-level data into portfolio-level descriptive statistics. This often meant that each researcher would have to customize and tweak their own version of this same logic

because of a lack of standardization amongst the group. Our goal for the tool we provide them is that it would provide the convenience of code-reuse that only standardization can afford. Therefore, one of the most important tasks in our development was the standardization of inputs and outputs within the DRP 2.0 pipeline.

Additionally, our clients wanted a high level of traceability at each stage of the pipeline, allowing them to notice problems and have sufficient information to diagnose their cause. This would be helpful both in the developmental stage and in the production stage of the pipeline. In development, when predictive models and optimal scoring algorithms are first being created, researchers can use the stored inputs and outputs to test their code and verify its accuracy. Then, in production, when predictions on the very uncertain realm of equity data are inevitably less precise than desired, maintainers can verify that this is not due to an error in their algorithms.

Our third design objective, modularity, is important because it can ensure that researchers are able to focus on development of their specific modules without spending time worrying about the other stages of the pipeline. Having an effectively-modular pipeline means having low coupling between modules, ensuring the breaks between modules are at logical points, and thoroughly documenting the interfaces between them. Together, these three work to ensure that researchers can easily use components created by other members of their team, swapping them out with ease to ultimately achieve the convenience in development which we described previously.

Finally, our design must be adaptable so that it can continue use as future changes occur. This is essential, because the Quant Research team that is acting as our client is a very new team, and their processes and methods are certain to change over time. If our work becomes obsolete within a year of completion, it would have been a waste of our time and a waste of our client's time. Therefore, all parameters to our tool have to be made as flexible as possible to anticipate future diversification in methodologies. Furthermore, since we can't fully anticipate all changes that could occur, we have to hand off the code to Principal in a way that they are able to read, understand, and modify it as needed.

The only rigid constraints on our software were in terms of the operation environment and data. Principal provided us with a Postgres database of raw stock data that had been pre-populated from FactSet, an subscription-based equity data provider. Our program has to consume that data in the form it was given to us and interface with that database as appropriate. Furthermore, since that database resides on an AWS server which we were given access to by Principal, our tool also needed to be able to run in that environment. This meant that any additional packages that our tool depended on would need to be set up in that context. Though these constraints defined the initial operation environment for our project, ideally our project should be easily portable to other similar settings.

One of the trade-offs we had to make during development was the balance between intensively-thorough documentation and feature-dense software. This tradeoff primarily came in the last months of our project. At that point we knew of a plethora of changes which could improve our package but knew that the greater value to our client ultimately was to provide them sufficiently-detailed documentation for what had already been accomplished. Additionally, in making this decision, we were able to document future changes we foresaw so that they would be able to pick up development and make those changes themselves if they desired.

A second tradeoff we considered in our project was the level of abstraction of our function calls and classes. Early on, we had thoughts of creating a library of more, smaller functions for typical data-manipulation tasks. As our project came into a more concrete form, we settled on a higher level of abstraction. This abstraction can have a downside of causing more difficulty for a user of our library in understanding what our code is doing and how to integrate with it. It also introduces more rigidity in the flow of data that could require a larger amount of rework of current code in order to realize usage of our package. Nonetheless, this higher abstraction level afforded us the ability to not get bogged down by the details of data science to which our team was not very familiar, but rather focus on structuring the flow of data logically from a high level.

3.5 Implementation and Technologies Used

3.5.1 Implementation Diagram

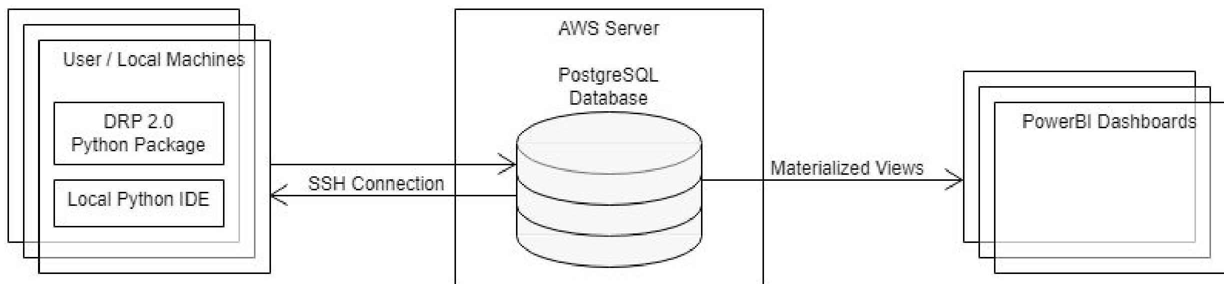


Figure 2: Implementation Diagram

3.5.2 Technologies and Software Used

We chose to implement the processing layer as a Python package. This package then represents the DRP 2.0 pipeline and is imported into scripts as one uses its many classes and functions. We connected from local machines to the data layer over SSH connections.

We were given access to an AWS server by our client with a PostgreSQL database already set up, so we chose to use it as our data layer.

Finally, our visualization layer exists only as a concept as the moment, but we know it will consist of at least PowerBI dashboards. We used materialized views in our data layer to provide an interface for any visualization tools our client chooses.

3.5.3 Rationale for Technology and Software Choices

For our processing layer, our main choice was between the two languages Python and R. We ended up going with Python for the following reasons:

- Our client's employees we spoke with were familiar with Python.
- We were more comfortable with it of the two.
- It's one of the easier programming languages to learn and work with for new programmers.
- It has existing data science tools and packages we can make use of that our client already uses.

The last point was really what helped motivate us to use Python. Utilizing Python gave us the ability to work with packages like Numpy and Pandas which our client already relies heavily upon in their current scripts. While R also has plenty of support for data science uses, by using the same language and packages as our client, we were able to more easily communicate ideas and translate between their scripts and ours with little effort. Python is also just a great general purpose language. It is lightweight, requires very little overhead for the code, and it is also used in the data science field. All this combined makes using Python as our language of choice for our project perfect.

As for our data layer, because we were given a server and database already set up and ready to go with all the necessary data, we had little reason to switch to any other alternative. Additionally our client makes use of AWS servers and PostgreSQL for their current projects, so it made perfect sense to stick with that choice.

While we didn't need to make any implementation choices for the visualization layer, we did decide on using materialized views to give the dashboards access to the data in the data layer. We made this decision as opposed to regular views so that the calculations necessary to visualize the data would only have to be run once for the dashboards to use them. A regular view would have required the data layer to reassemble the data each time.

3.6 Modules

What follows is a brief description of each of the modules that make up the entirety of the DRP 2.0 pipeline.

3.6.1 Aggregation

The aggregation module is in charge of aggregating raw stock level data into factor portfolios that are both saved in the database and consumed by other modules of the pipeline. Because these portfolios are being saved in the database, it also checks to see if the requested portfolio has already been made before so it can reload it instead of re-aggregating it.

The module expects a description of what factor portfolios are to be aggregated in the form of a Factor Portfolio or Factor Portfolio Factory object as well as implementations of the abstract base classes for preprocessing or postprocessing if desired.

Once aggregation is complete, the module saves the results in the database for later reuse and then outputs a set of aggregated factor portfolios as requested in the form of a list of Factor Portfolio objects.

There are few constraints in this module other than that any requested data must actually exist in the given database to be aggregated. The module will provide detailed errors should a user make an invalid request.

The following figure shows the main flow of data internal to this module.

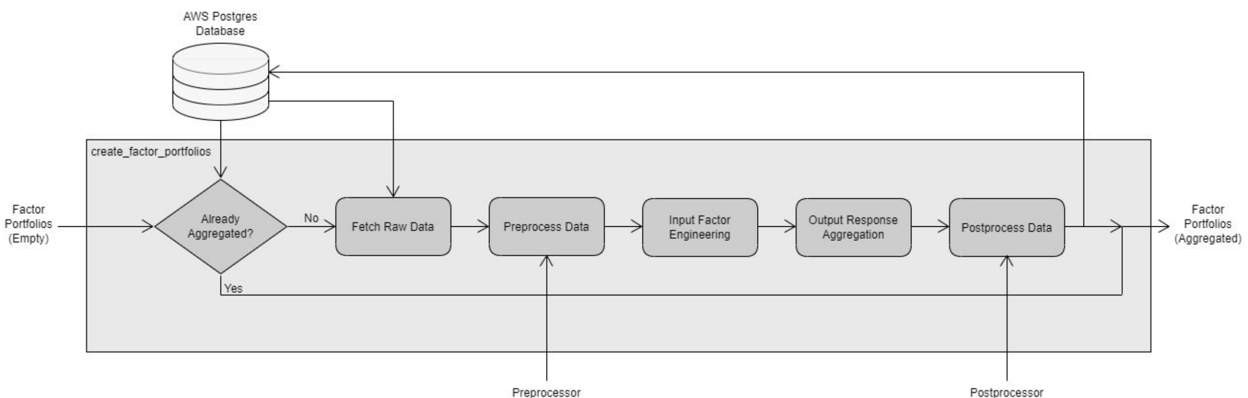


Figure 3: Aggregation Module Diagram

3.6.2 Predictions

The below Figure 4 is the design for the predictions step of the project. In this module, there are are a number of models that are created by the user (matching the abstract class). The factor portfolios are provided and the models will run with user provided inputs to output the predictions for the factor portfolios.

The predictions modules works by accepting first the aggregated factor portfolios from the factor aggregation step. From the accepted aggregated factor portfolio, the user then will provide the

necessary model inputs (different from model to model). With the inputs and factor portfolio, the predicted data will be generated.

The constraints in this step are that we have used an abstract based class for the model classes. Any model that is to be implemented here in this step must conform to this abstract class. Depending on how the original model is implemented, there may or may not be troubles in implementing the model to fit the abstract class. Another constraint here is currently the abstract model class is assuming that the dates we are predicting, we are not outputting past the end date. So the predictions will start at the start date of the given factor portfolio and end at the provided date.

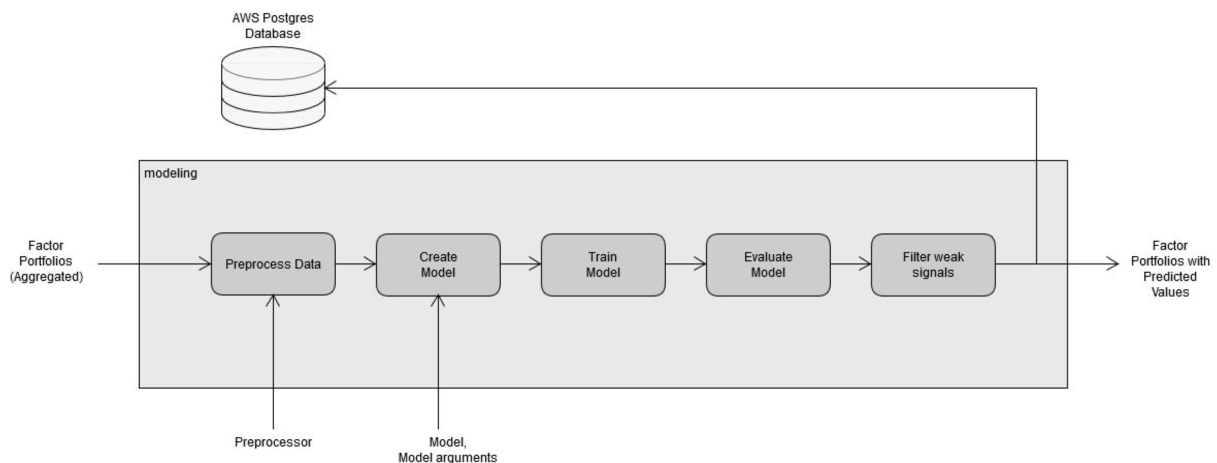


Figure 4: Predictions Module Diagram

3.6.3 Stock Scoring

The scoring stage is broken up into two parts: factor policy generation and stock scoring. Figure 5 below shows the inputs and outputs of the entire scoring stage. The scoring stage expects aggregated factor portfolios as well as predicted values from the predictions step and outputs a table of relative scores for each company at each time point.

The factor policy generation step essentially scores whole factors based on the predicted data for their factor portfolios. It takes as input aggregated factor portfolios containing predicted returns and uses those to create a factor policies table. This table contains relative weights for each factor at each time point that sum to 1 for any given row. Higher weights are indicative of greater predictive power for the corresponding factor at that time point. This stage is to be implemented by data scientists at Principal and, in practice, is typically further segmented into two internal stages as shown in Figure 5.

The second scoring step, called stock scoring, uses a factor policy and more raw stock data to score individual stocks at each time point. This also is broken into two segments. First, the database is queried to produce a table containing percentile rankings of each company for each factor at each time point. This information is then provided, along with a factor policies table, to

the primary abstract function for stock scoring. This can be used, like all of the other abstract classes we created, as an extension point for customizing functionality of the pipeline. However, since there is a fairly standard and straightforward way this step is typically done, we also provided a concrete implementation that can be used out of the box.

The generated factor policies and stock scores are all stored back in the database with links to the factor portfolios and predicted data that served as their inputs. This provides full traceability of the scoring stage.

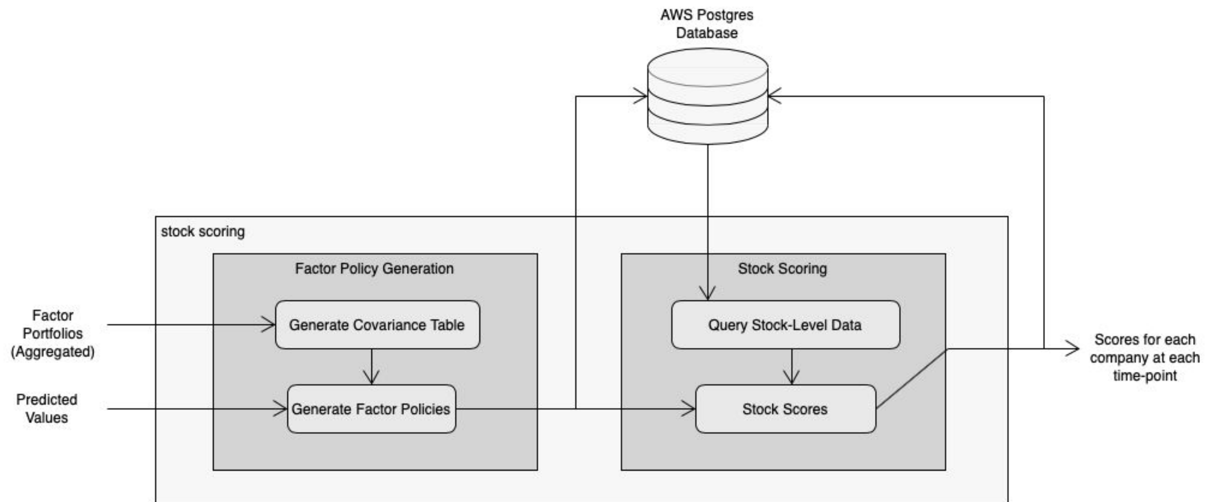


Figure 5: Stock Scoring Module Diagram

3.6.4 Database

The database stores the information being passed along the pipeline from every step. This allows for the user to easily diagnose issues since they are able to see the data at every step of the pipeline. See figure 6 below.

Along with the multiple tables that are in the database, materialized views are also within the database to help organize the data for easier implementation with Power BI dashboards. Because the materialized views are able to store information, past data is easily accessible for the user.

There are some constraints with the database. The materialized views can only give information to the Power BI dashboards that is already in the database. Since the materialized views do not currently have a trigger function, they must be manually refreshed with data whenever the user wants to look at the most recent data. Also, anytime the schema is changed, the code must also be changed so it can correctly insert the information in the new schema.

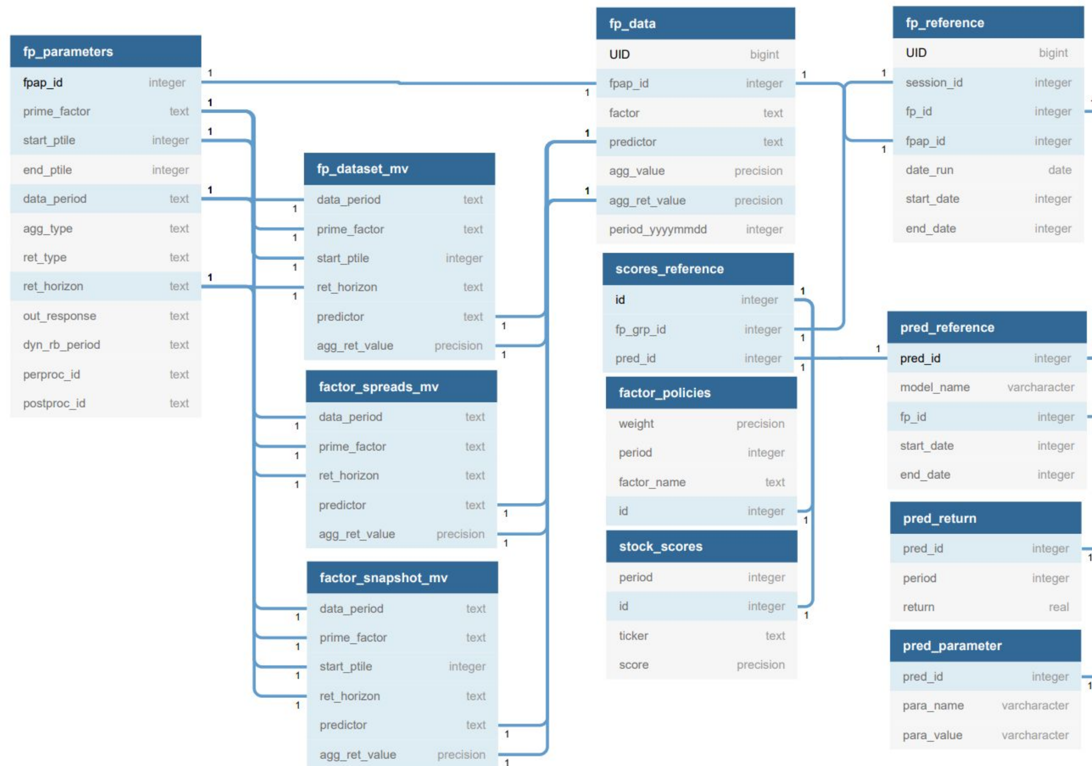


Figure 6: Database Schemas and Relationships

3.7 Applicable Standards and Best Practices

Our project used three standards from the start of the project. A standard related to the software life cycle, a standard for test techniques, and a standard for validation. [1] - [3]

While we tried to use all these different standards for our project, due to time constraints and the nature of our project, we could not completely follow these standards. Nevertheless, our group made great strides in trying to use all of these standards.

The first standard we used was the IEEE Systems and software engineering--Software life cycle processes. This standard was about the software life cycle process for the entire project. When we started the project, we believed that this was a good standard to use because we were creating a prototype for our client. Thus we thought that having a good grasp of the entire software life cycle processes would be important to have a successful project.
Our project followed closely what this standard has stated. During our initial stage, we have made great efforts to define all of our modules and processes. We had many Organizational project-enabling processes to determine how certain parts should work in relation to what our client wanted. Through constant communication we have accomplished this. Since our project was focused on prototyping the DRP 2.0, the other stages of the software life cycle did not matter as much to us. [1]

The second standard Software and systems engineering--Software testing--Part 4: Test techniques, was useful to learn what test techniques we should use for our testing phase. However, due to some changes to how we were doing testing, and changes to how we had to take some testing out due to them not being available, we could not fully use all these standards. We couldn't efficiently use all the test techniques mentioned in this standard (that would've unrealistic too), but for the testing we could do, we tried our best to make use of the techniques mentioned in this standard. State Transition testing, syntax testing, data flow testing were some of the techniques we successfully used. [2]

IEEE Standard for System, Software, and Hardware Verification and Validation, we used the vv table as a guideline to create which processes, deliverables and modules were important to the project. Through communication with our client and our own meetings, what we should focus on was successfully made, so our verification step went well. In our validation step, we took steps to validate the system analysis V&V process, Validation process, and Design Definition V&V process. Due to the size of our team, we could not validate everything according to the standards, our team made great efforts to do the verification and validation our team thought was crucial to the success of our project. [3]

4. Test Plan

4.1 Overall Plan

Our testing plan for this project revolves mainly around our continuous integration setup. As detailed in the following sections, we wrote unit tests, interface tests, and integration tests as we developed the pipeline and placed them all in a testing suite that gets run by our continuous integration server. This helps pinpoint where exactly some functionality was broken, as well as protects the master branch of the project from ever being broken, because it won't let you merge unless all the tests are passing. We wrote the tests as we developed the features so that we'd always know whenever something had broken.

For aggregation tests we made sure to get full option coverage over the important choices that can be made such as aggregation type (Static/Dynamic) and return type (Absolute/Excess).

4.2 Unit Testing

For unit testing we wrote up a suite of tests for each of the 3 software based modules of the project: Aggregation, Predictions, and Scoring. These tests mainly ensured the outputs of any internal or intermediate functions were correct. We also added these tests to our continuous integration suite of tests so that we would know immediately if we accidentally broke any piece of functionality.

4.3 Interface Testing

For interface testing, we ensured that the outputs of each module were formatted correctly and as expected. Most of the data we pass around in the pipeline is in the form of Pandas dataframes, so we checked to make sure that the dataframe getting passed from module to module always had the expected index and columns. Again, these tests were added to our continuous integration system so that we'd know if someone ever violated the interface contracts immediately.

4.4 System Integration Testing

Once each of the individual components of the pipeline were finished and passing their own unit tests, integration tests were written to test their functionality as a cooperating system. In these tests the entire pipeline was validated for correctness by running each of the high-level modules sequentially and verifying key qualities about the data frames that were output by each. These tests didn't focus on specific data in any of the results, but rather invariant properties that should be true regardless of raw data or starting parameters. Certain invariants like the relationships between the input parameters and the column titles and row labels of the resulting dataframe were tested at every stage. Other more specific tests were also used for individual stages, such as the sum of the values in a row of the factor policies table summing to one.

4.5 Use Case Testing

For use case testing we packaged up our code at various stages of the process and let our client attempt to use it. Then they would report back with information such as how easy or difficult it was to use for their purposes as well as what kinds of changes they'd like to see that would make it more useful for them. We went through this process both over emails and in person at different times. In this way we discovered whether we'd successfully covered each use case or if we needed to revisit certain ones and make it more clear how to use our code in that way. We also did our best to reproduce realistic operating conditions when testing our code ourselves in order to catch issues even before reaching this stage of testing.

5 Testing Evaluation

5.1 Test Case Evaluation

At of the end of our project, 100% of our various tests passed, from unit tests, to integration tests, to interface tests, meaning that each module of the pipeline alone and integrated together are functioning as intended.

On the other hand we benchmarked the aggregation module in specific since it's the only module we were to implement in its entirety, and we found that it is unfortunately slower than their current method of aggregation. In addition, loading portfolios from the database takes significantly longer than recalculating and aggregating manually. The issue stems from us not having the time to optimize some sections of the pipeline in favor of adding more of the architecture and features our client desired. Our client was more interested in the overall architecture decisions and designs we as Software and Computer Engineers could bring to the table instead of having us work on small data transformation optimizations. So while the benchmarking results are somewhat disappointing, they are at least not too unexpected. The results of running the aggregation module on 180 various requested factor portfolios are as follows.

Interaction Level	Time Taken (s)	Time Taken (min)	Avg. Time/Portfolio (s)
No DB Interaction	1735.47	28.9245	9.6415
DB Interaction, Portfolios don't exist	2041.70	34.0283	11.3428
DB Interaction, Portfolios already exist	3401.00	56.6833	18.8944

Table 1: Aggregation Benchmarking Results

5.2 Validation and Verification

The results of our use case testing and otherwise validation of our end product were mostly positive although not entirely so. We found that our final product did indeed cover the necessary use cases, and our client was able to utilize the package effectively to accomplish their goals. This we can consider a success.

On the other hand, our client brought up several valid concerns about the project throughout its lifetime that were never able to be addressed for lack of time. Issues we didn't consider mandatory or severe such as reorganizing class members for clarity or switching to a preferred format of dates and times were overlooked in favor of completing the project on time. Another month or two of work would have been enough to implement many of these quality of life changes, but as it stands, the project is only able to measure up to a minimum viable product standard. As such, we can consider the project mostly a success despite some unfortunate loose ends.

6 Project Management

6.1 Roles and Responsibilities

Josiah Anderson: Meeting Facilitator

- Created presentation material for weekly meetings with our client Principal Financial
- Led team, client and advisor meetings
- Set up the database schema
- Worked to establish the tables used for storing aggregated portfolio data
- Established a documentation protocol to be used throughout the project

Doh Yun Kim: Scribe

- Documented team meetings and meetings with Principal
- Led investigative interviews with Principal team members to establish areas of need
- Designed the schedule of work to be used by our team
- Worked closely with a member of the Principal team to create the predictions module of the DRP 2.0 pipeline

Gabriel Klein: Report Manager

- Organized weekly reports
- Oversaw the completion of all the documentation required by the class
- Maintained the Gitlab issues board
- Designed the factor aggregation module of the DRP 2.0 pipeline
- Worked with Jacob to establish integration tests on our Continuous Integration server

Drake Mossman: Communications Manager

- Maintained constant communication with our client and advisor
- Developed the stock scoring module of the DRP 2.0 pipeline
- Created a schema and corresponding tables for storing stock scoring data
- Worked to integrate the individual pipeline modules
- Developed tests for integration

Jacob Richards: Quality Assurance

- Set up the Continuous Integration server
- Helped with knowledge interviews
- Developed materialized views to be consumed by external power BI dashboards
- Provided input on testing procedures

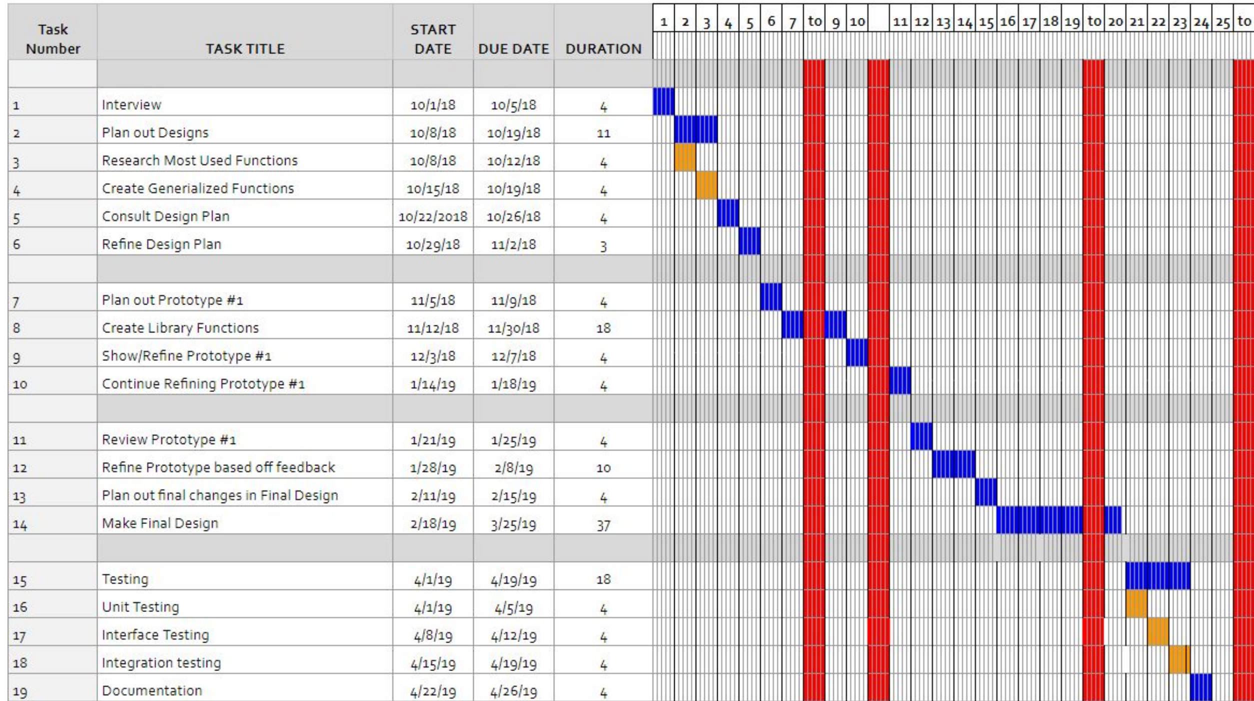
Nathan Schaffer: Overseer of Work

- Set up the Gitlab issues board to be used for managing tasks
- Worked with Drake to develop the stock scoring module
- Packaged the pipeline into a single Python package that can be easily imported

6.2 Project Schedule

Task Number	TASK TITLE	START DATE	DUE DATE	DURATION
1	Interview	10/1/18	10/5/18	4
2	Plan out Designs	10/8/18	10/19/18	11
3	Research Most Used Functions	10/8/18	10/12/18	4
4	Create Generialized Functions	10/15/18	10/19/18	4
5	Consult Design Plan	10/22/2018	10/26/18	4
6	Refine Design Plan	10/29/18	11/2/18	3
7	Plan out Prototype #1	11/5/18	11/9/18	4
8	Create Library Functions	11/12/18	11/30/18	18
9	Show/Refine Prototype #1	12/3/18	12/7/18	4
10	Continue Refining Prototype #1	1/14/19	1/18/19	4
11	Plan out Prototype #2 (based off #1)	1/21/19	1/25/19	4
12	Create Prototype #2	1/28/19	2/8/19	10
13	Refine Prototype #2	2/11/19	2/15/19	4
14	Create Final Design	2/18/19	3/8/19	20
15	Testing	3/11/19	4/5/19	24
16	Functional Testing	3/11/19	3/15/19	4
17	Non-functional Testing	3/25/19	3/29/19	4
18	Usability Testing	4/1/19	4/5/19	4
19	Documentation	4/8/19	4/12/19	4
	Risk Buffer			0

Table 2: Project Schedule Planned Version



Task Number	TASK TITLE	START DATE	DUE DATE	DURATION
1	Interview	10/1/18	10/5/18	4
2	Plan out Designs	10/8/18	10/19/18	11
3	Research Most Used Functions	10/8/18	10/12/18	4
4	Create Generialized Functions	10/15/18	10/19/18	4
5	Consult Design Plan	10/22/2018	10/26/18	4
6	Refine Design Plan	10/29/18	11/2/18	3
7	Plan out Prototype #1	11/5/18	11/9/18	4
8	Create Library Functions	11/12/18	11/30/18	18
9	Show/Refine Prototype #1	12/3/18	12/7/18	4
10	Continue Refining Prototype #1	1/14/19	1/18/19	4
11	Review Prototype #1	1/21/19	1/25/19	4
12	Refine Prototype based off feedback	1/28/19	2/8/19	10
13	Plan out final changes in Final Design	2/11/19	2/15/19	4
14	Make Final Design	2/18/19	3/25/19	37
15	Testing	4/1/19	4/19/19	18
16	Unit Testing	4/1/19	4/5/19	4
17	Interface Testing	4/8/19	4/12/19	4
18	Integration testing	4/15/19	4/19/19	4
19	Documentation	4/22/19	4/26/19	4

Table 3: Project Schedule Actual Outcome

During the first semester, our actual schedule and our projected schedule matched. During our first semester, we were focused on figuring out what our client wanted and needed. We met our goals in regards of keeping with the schedule. However starting the second semester our schedule changed from the projected one.

When we were presenting our first prototype, the reviewing and refining phase of our first prototype took longer than expected. Due to the fact our first prototype stage took so long, our team forgone the second prototype phase and decided to start making changes to our first prototype and focus on our final design. Our final design phase took longer than expected to create. Multiple troubles in regards with trying to make our final design line up with the current design we had. Due to the final design step taking a week longer than expected, our expected testing stage took was delayed back to the start of April.

The types of testing we did also changed. Back when we first created gantt chart, we did not have a clear plan for testing. After we got our actual design ironed out, we managed to get our testing phase cleared out, and the actual testing we did was different from the ones we planned. While we did have usability testing as one of the few tests we wanted to do for our project, do to schedule conflicts with our client, we could not fit in usability testing into our project.

In our original schedule, we had a three weeks as a risk buffer. This risk buffer we allocated for ourselves became useful when our schedule didn’t confirm exactly to our original one. We

ended up using all three weeks of our risk buffer when our final design work took longer than expected.

6.3 Risk and Mitigation

Risk Map	Consequence	Likelihood	Risk Description	Project Impact	Risk Area	Symptoms	Risk Response	Response Strategy
H	5	D	Significant conceptual misunderstanding of pipeline processes	Most, if not all of the work done up to this point will be scrapped, significant setback in schedule	Scope, Schedule	We present our work to Principal and they tell us the functions aren't what they wanted	Mitigation	Continuous feedback from Principal at every stage of development will significantly decrease the likelihood
S	4	D	One of the PowerBI dashboards doesn't correspond well to any of the interfaces or functions we've defined	Extend scope of project to rework or add to function library	Schedule	Code for dashboard is not easily reworked to use DRP 2.0 library	Mitigation	Early interactions with the dashboards will allow us to identify incompatibilities when we still have time to correct for them
S	4	C	Don't get to the end of the development by our deadline	Have to present unfinished project	Scope	Semester finished, project not complete	Monitor and Prepare	Follow schedule; Include sufficient buffer
S	4	D	The first prototype is not received well	Considerable time from the risk buffer must be taken out to make changes to the prototype	Organizational, Schedule	During the presentation of the first prototype Principal does not like it	Mitigation	Make rapid changes to the prototype with all team members thriving to finish the redesign as quickly as possible
S	3	A	A team member gets busy with other course work	A team member who is busy with other work might not be able to finish their tasks on time or hand in substandard work	Organizational, Schedule	A team member is unavailable at work at designated times or are missing deadlines	Mitigation	Redistribute the work the team member's work to the other people
S	4	C	Procedural Risk: Improper process implementation, conflicting priorities, or lack of clarity in responsibilities	Team members code being incompatible with each other, lack of understanding of goals	Implementation	Incompatible code	Mitigation	As a group discuss the problems and identify the solutions
M	3	C	Code written in Python can't be easily ported to R package	Reduce the scope of the project so that our library is only in Python	Scope, Schedule	When attempting to port to R, we find large parts of the code that would need significant rework to function in R	Mitigation	Investigate what specific aspects of Python code might be difficult to port and try to work around them
M	2	B	A team member gets stuck on a part of their task and can't get work progressed	The person who is stuck cannot get work progressed, potentially risking delays	Schedule, Organizational	A team member is not getting any sufficient work done a task even after working on it for a (set amount of time)	Monitor and Prepare	Consistent communication amongst team about blockers; Assistance from team members early when necessary
L	2	D	Changes submitted for one task breaks code that was previously finished	The code will not function right and will need to be fixed before moving on, taking additional time and effort	Schedule	Previously tested code not working as expected following push to git repository	Mitigation	Ensure problem is caught early if it occurs through automated regression tests in a continuous integration framework

Table 4: Risk Description Chart

Level	Descriptor
1	Insignificant
2	Minor
3	Moderate
4	Major
5	Catastrophic

Table 5: Risk Consequences Mapping

Level	Descriptor	Description
A	Almost certain	Almost certain Expected to occur in most circumstances
B	Likely	Will probably occur in most circumstances
C	Moderate	Should occur at some time
D	Unlikely	Could occur at some time
E	Rare	may occur only in exceptional circumstances

Table 6: Risk Likelihood Mapping

	Consequence				
Likelihood	1	2	3	4	5
A	M	S	S	H	H
B	L	M	S	H	H
C	L	M	M	S	H
D	L	L	M	S	H
E	L	L	L	M	S

Table 7: Risk Mapping from Likelihood and Consequence to Severity

Key	Risk Map
H	High Risk
S	Significant risk
M	Moderate risk
L	Low risk

Table 8: Risk Severity Mapping

The above shows some of the risks we have planned out during our initial project plan of our project. There were some risks that never came up. The risk regarding the PowerBI dashboards never came up. The project on our client's side never came far enough for our project to worry about integrating with the PowerBI dashboards. Our first prototype was also well received. This was important because due to our changed schedule, we did not have time to implement major changes into a second prototype, and instead directly proceed into our final design based off our first prototype. The R conversion risk never came up either because at the start of the spring semester, our team has decided that converting our project to R was outside the scope of our project, so we scaled back on that. So that risk became irrelevant.

For the risks that did came up, we feel that in our project, we did a decent job of dealing with all the problems that came up in our way. The first risk, and the one we were always worrying about during the beginning of the project was the misunderstanding of the pipeline process. During the start of our project, our team did not understand the DRP process well, and we were struggling with understanding it. This put us in trouble when designing some of the initial modules. Our plan was to mitigate this risk through constant communication with our client, and this worked out well. We had weekly meetings with Principal for the entirety of the project. Through this constant communication, any misunderstandings we had were resolved early instead of later.

When a team member got busy with other course work, or got stuck with a certain module or process in our project, our team handled this by reshuffling work. The module nature of our project made it so that having a team member being stuck or being busy will not impact the entire project negatively. We have considered when starting our project, and this came to be useful when we did get stuck and become busy. So we handled this risk well. Thanks to the constant communication we had within our team throughout the entire project, our integration step was not as worrisome as it was.

6.4 Lessons Learned

Over the course of this project, our team has a learned a few valuable lessons. The first lesson we learned was the importance of establishing clear expectations and project guidelines early on in the project. The first couple months of our project were very frustrating because we did not have a lot of clarity as to what we were supposed to be developing as a team. To address this issue, we interviewed members of the team at Principal to find areas where our software engineering knowledge could be used to improve their workflow.

Another lesson we learned was the importance of constant communication. By nature of our project being a prototype, the design plan changed a lot more than most projects. Because of our constant communication with the team at Principal though, we were able to address these changes as they came up instead of wasting time on a feature that was no longer important or relevant to the project.

One more important lesson we learned was perseverance. In the beginning, our project seemed was pretty overwhelming because of the lack of experience our team had with data science. This led to a long process of learning what data science looks like at Principal, and how as software engineers, we can contribute to the data science team. By the end of our project, our knowledge of data science, although still far from complete, was more than sufficient for us to complete this project.

7 Conclusions

7.1 Closing Remarks

In summary, our team was mostly successful in architecting a software solution for our client's needs in quantitative data science. The Python package and database design we created allows for them to quickly and easily create and evaluate stock portfolios. The architecture is also modular such that they are able to swap components in and out as abstract classes without modifying the now standardized interfaces between stages. Finally the database keeps track of the outputs of each portion of the pipeline and provides traceability for how and when different stages were executed.

Our testing shows that our final product is reliable from a module level to a system level, and our client was generally happy with the outcome of the project. Benchmarking shows that our solution as it currently stands is somewhat slower than was expected, but there is still plenty of room for optimizations to be made. As the first stage of their longer term project, our package is only the first prototype, and it will help lay the foundation for other teams to iterate on in the months to come.

7.2 Future Work

As a rough prototype of our client's investigation into automating their quantitative data processes, our project could certainly be taken in several directions for future work. We will discuss a few of these options.

One big improvement that could be made is related to the efficiency of the aggregation portion of the pipeline. Our current implementation was focused on extensibility, customizability, and maintainability, so unfortunately the raw efficiency of the aggregation suffered a lack of attention. This prototype pipeline could be made much more useful if some time was spent finding optimizations in the aggregation of which we know several do exist. For example, many portfolios share the same aggregated input factor data, but our pipeline processes each portfolio individually, repeating the same work for each one. Exploiting these similarities is likely the easiest speed boost to work towards from this point.

Another improvement that could be made is adding support for more of the input factors and input predictors our client is currently using in their own code. One example is to add support for using macro data as an input factor which should be as simple as querying the relevant database table and appending the data alongside the rest of the inputs. There are also several input predictors such as spread decile mean, spread decile standard deviation, spread universal mean, and spread universal standard deviation that the pipeline doesn't currently support. These will require a little more doing to get working, but the necessary architecture exists to

perform a full query of all relevant raw stock level data and do whatever data transformations desired on it.

Finally as with any database heavy project, support for parallelized calculations and asynchronous server calls would greatly increase the speed of the pipeline. Currently all database interactions are blocking, which wastes significant processing time. We suggest finding ways to parallelize each module of the pipeline so that the all processor cores can be kept busy at all times as well as finding ways to make database calls asynchronous so that the processor is never just waiting for data to come back.

References

[1] *Systems and software engineering -- Software life cycle processes,* ISO/IEC/IEEE 12207:2017, 2017

[2] *Software and systems engineering--Software testing--Part 4: Test techniques,* ISO/IEC/IEEE 29119-4:2015(E), 2015

[3] *IEEE Standard for System, Software, and Hardware Verification and Validation,* IEEE 1012:2016, 2017

Appendix A: Team Information

Josiah Anderson

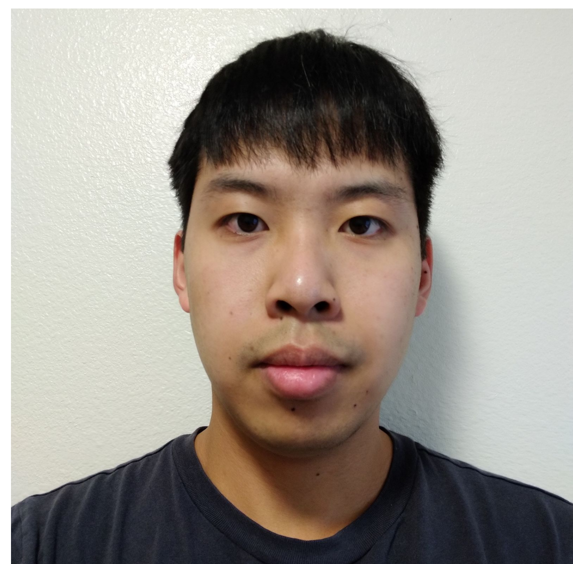
Josiah is a Computer Engineering major and our team's Meeting Facilitator. He's in charge of setting up our meeting times and locations. He also leads our meetings and puts together great presentations when necessary.

His main contribution to the project was working on our database, schemas, and views along with Jacob Richards.

josiande@iastate.edu

Doh Yun Kim

DK is a Software Engineering major and our team's Scribe. He's in charge of taking notes during both our internal meetings and our meetings with our client. He formats them all nicely and posts them online for us to look back over.



His main contribution to the project was working on the modeling and predicting portion of the pipeline.

dohyunk@iastate.com

Gabriel Klein

Gabe is a Software Engineering major and our team's Report Manager. He's in charge of notifying the team of report deadlines and making sure they get done on time. He compiles and formats all the work once it's done and then submits it or posts it on the website.

His main contribution to the project was working on the data aggregation of raw stock level data to factor portfolios part of the pipeline.

gabrielk@iastate.edu

Drake Mossman

Drake is a Software Engineering major and our team's Communication Manager. He's in charge of the communication with our client. He writes up emails to our client and our advisor as necessary and makes sure replies are sent in a timely manner.

His main contribution to the project was working on the factor policy and stock scoring portion of the pipeline along with Nathan Schaffer.

dmossman@iastate.edu

Jacob Richards

Jacob is a Software Engineering major and our team's Quality Assurance Manager. He's in charge of the testing side of our project. He makes sure tests are written for new code and maintains the continuous integration systems.

His main contribution to the project was working on our database, schemas, and views along with Josiah Anderson.

jacobr17@iastate.edu

Nathan Schaffer

Nathan is a Computer Engineering major and our team's Overseer. He's in charge of assigning tasks to team members to get work done efficiently. He prioritizes them to ensure we avoid bottlenecks and to get client feedback as soon as we can.

His main contribution to the project was working on the factor policy and stock scoring portion of the pipeline along with Drake Mossman.

nathans@iastate.edu