Merck-Data Mine Documentation

Merck and Data Mine Corporate Partnership Team

2021-04-19

Contents

Chapter 1

Introduction

This book will serve as tutorial based documentation for the Merck - Data Mine Coporate Partnership team for the 2020-2021 academic year.

Chapter 2

Tips for Writing Great Documentation and Walkthroughs

2.1 Find a Good Topic

While writing documentation for a project, it would nearly impossible to include every piece work completed or every line of code written. This makes it important to pick important topics to write about. The goal is to include as much specificty as possible while also remembering the project as its entirety.

Here are a few helpful concepts to consider when choosing a topic:

1. Step by step guides – perfect for readers to learn quickly and implement in their own projects
2. In depth discussions of a specific topic – great for readers who are looking for deeper knowledge in a topic
3. Numbered lists of useful facts about a common topic – lightweight readings that readers can consume in bits and pieces.

2.2 Make Goals and Audience Clear

For these writings, our team will have a dual purpose of writing them.

1. Record the work we have completed

2. Create a centralized location for tutorial based learning

The audience to consider is future team members of the Merck-Data Mine Corporate Partnership and Merck scientists looking to read and learn from our work.

Remember to keep the audience and goal of our documentation in mind as writing your entries.This will help the book keep continuiuty and give readers the best chance to get exposed to the work that we have completed.

2.3 Have a Begining, Middle, and End

It is important to have an introduction, body, and conclusion while writing the documentation. This helps with fluidity within each document and allows for easier comprehension.

2.3.1 Introduction

The introduction should encourage the reader to continue reading. Start with information about what will be covered in the read and how it applies to the project. Try to keep the introduction less technical so readers aren't discouraged by the complexity of the document.

2.3.2 Body

The body is where you elaborate on all that you discussed in the introduction. Provide depth and instruction while still relating to the project as a whole. Use headings, photos, numbered lists, bullet points, and formatting to help provide small bits of information at a time. This is where you can facilitate a technical discussion with code.

2.3.3 Conclusion

Always finish the read with a conclusion, providing assurance of what was just learned and include possible resources for more information (i.e. academic papers, blog posts, youtube videos). It is also appropriate to give the reader a domain in which to use the skills they have learned from your documentation.

2.4 Getting Feedback and Iterate

Everyone on the team is encouraged to follow the documents that are added to this book. Read through them and provide helpful feedback to your teammate on how to improve.

Some common things to look out for:

1. Formatting – Does the document flow properly? Are there enough images, code, headers, etc…?
2. Formality – Is the document written well? Is the language approiate?
3. Goal and Audience – Does the document relate to the goals and target audiences of this book?
4. Attention – Was the reading interesting? Is there opportunity to learn from the read?

2.5 Practice, Practice, Practice

Writing the entries in this book will undoubtably get better over time. You are welcome to write as many entries as you'd like. They can be simple or deep, long or short, imformational or technical, etc. As long as the information in this book is informative and relevant to the projects we hope to complete, it is encouraged for all members of the team to write what they want!

Chapter 3

Documentation Example - The Basics of R Markdown

3.1 Overview of R Markdown

R Markdown is a file format for making dynamic documents with R. An R Markdown document is written in markdown (an easy-to-write plain text format) and contains chunks of embedded R code, text, images, headers, and more.

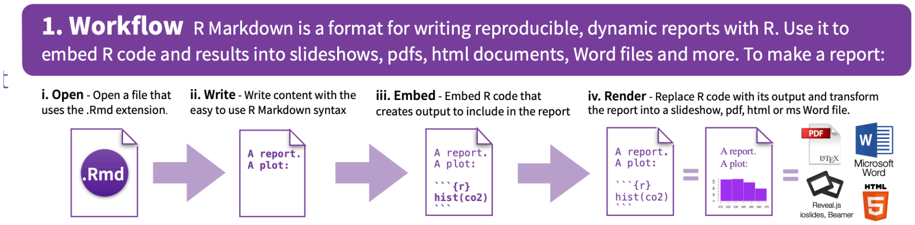
In this walkthrough, we'll discuss some of the functionality of R Markdown Documents and how to add images, code, and other features to the document.

Throughout this tutorial, we'll be reviewing the contents of the R Markdown cheat sheet that has most important information for writing in Markdown. The cheat sheet can be found at https://rstudio.com/wp-content/uploads/2015/02/rmarkdown-cheatsheet.pdf.

3.2 Workflow

One of the great features of Markdown files is that they can be rendered to PDF files, Word documents, HTML content, and more. This allows the writer to easily write in R and export the document how they see best.

Take a look at a common lifecylce of R Markdown documents in th following picture:



1. Workflow R Markdown is a format for writing reproducible, dynamic reports with R. Use it to embed R code and results into slideshows, pdfs, html documents, Word files and more. To make a report:

i. Open - Open a file that uses the .Rmd extension.

ii. Write - Write content with the easy to use R Markdown syntax

iii. Embed - Embed R code that creates output to include in the report

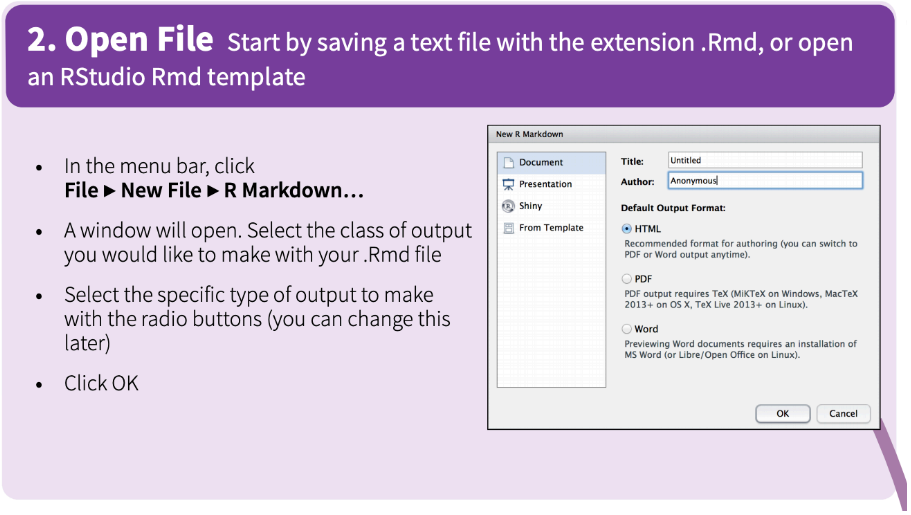
iv. Render - Replace R code with its output and transform the report into a slideshow, pdf, html or ms Word file.

.Rmd
A report.
A plot:
```{r}
hist(co2)
```
=
PDF
LaTeX
Microsoft Word
HTML
Reveal.js
ioslides, Beamer

Figure 3.1:

3.3 Opening a New File

Writing R Markdown files is easiest within R Studio. Navigate to File > New File > R Markdown to create a new file.



2. Open File Start by saving a text file with the extension .Rmd, or open an RStudio Rmd template

- In the menu bar, click **File ▸ New File ▸ R Markdown…**
- A window will open. Select the class of output you would like to make with your .Rmd file
- Select the specific type of output to make with the radio buttons (you can change this later)
- Click OK

New R Markdown

Title: Untitled

Author: Anonymous

Default Output Format:

● HTML
Recommended format for authoring (you can switch to PDF or Word output anytime).

○ PDF
PDF output requires TeX (MiKTeX on Windows, MacTeX 2013+ on OS X, TeX Live 2013+ on Linux).

○ Word
Previewing Word documents requires an installation of MS Word (or Libre/Open Office on Linux).

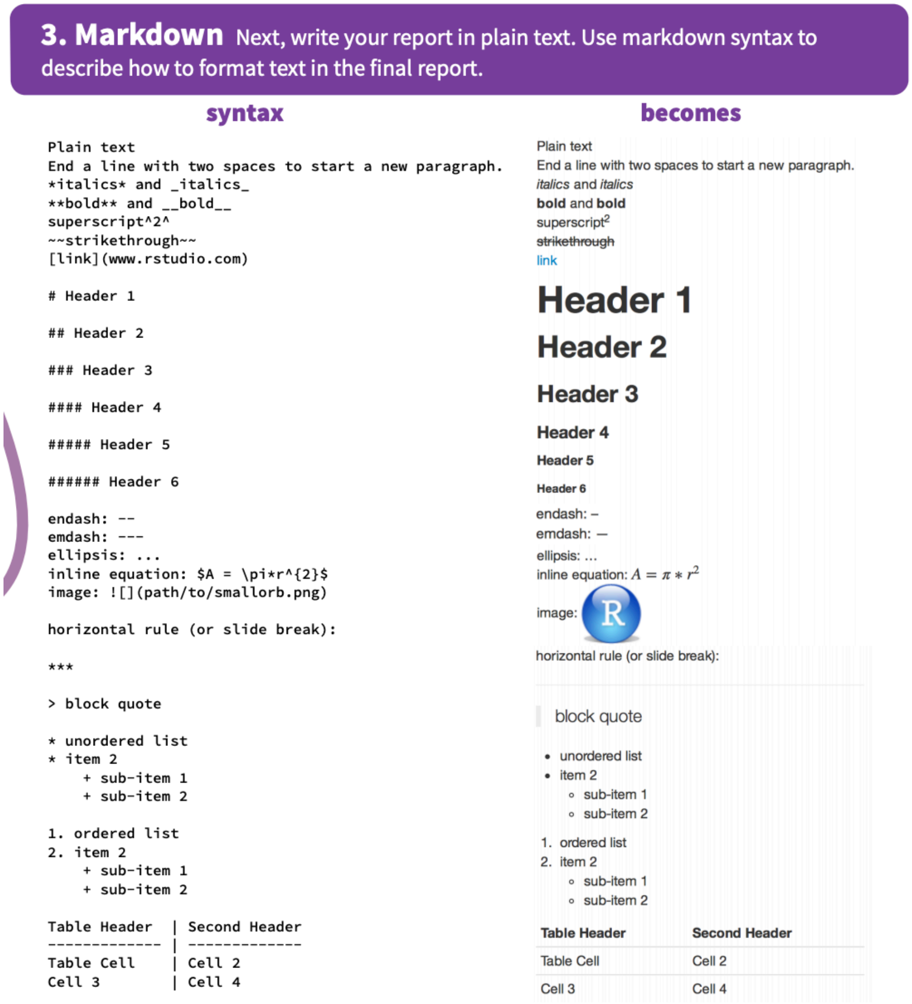
Figure 3.2:

3.4 Helpful Syntax

Take a look through helpful sytax in this photo:

Here are some examples to vie. Take a look at the raw R Markdown to view the syntax in pratice.

italics **bold** ~~strikethrough~~



3. Markdown Next, write your report in plain text. Use markdown syntax to describe how to format text in the final report.

syntax

```
Plain text
End a line with two spaces to start a new paragraph.
*italics* and _italics_
**bold** and __bold__
superscript^2^
~~strikethrough~~
[link](www.rstudio.com)

# Header 1

## Header 2

### Header 3

#### Header 4

##### Header 5

###### Header 6

endash: --
emdash: ---
ellipsis: ...
inline equation: $A = \pi*r^{2}$
image: ![](path/to/smallorb.png)

horizontal rule (or slide break):

***

> block quote

* unordered list
* item 2
    + sub-item 1
    + sub-item 2

1. ordered list
2. item 2
    + sub-item 1
    + sub-item 2

Table Header  | Second Header
------------- | -------------
Table Cell    | Cell 2
Cell 3        | Cell 4
```

becomes

Plain text
End a line with two spaces to start a new paragraph.
italics and *italics*
bold and **bold**
superscript^2
~~strikethrough~~
link

Header 1

Header 2

Header 3

Header 4

Header 5

Header 6

endash: –
emdash: —
ellipsis: ...
inline equation: $A = \pi * r^2$

image: R

horizontal rule (or slide break):

> block quote

- unordered list
- item 2
    - sub-item 1
    - sub-item 2

1. ordered list
2. item 2
    - sub-item 1
    - sub-item 2

Table Header	Second Header
Table Cell	Cell 2
Cell 3	Cell 4

Figure 3.3:

> Block Quote

- Bullets
- Bullets
- subitem

1. Lists
2. Lists

- subitem

3.5 Embed Code

One of the best features of R Markdown is the ability to add code to your document. To add a code snippet, click on the green *insert* button in your R Studio tool bar and choose which language you would like to use!

```
print("This is an R code snippet")
```

```
## [1] "This is an R code snippet"
```

```
print("This is a python code snippet with eval=FALSE")
```

```
echo this a bash code snippet
```

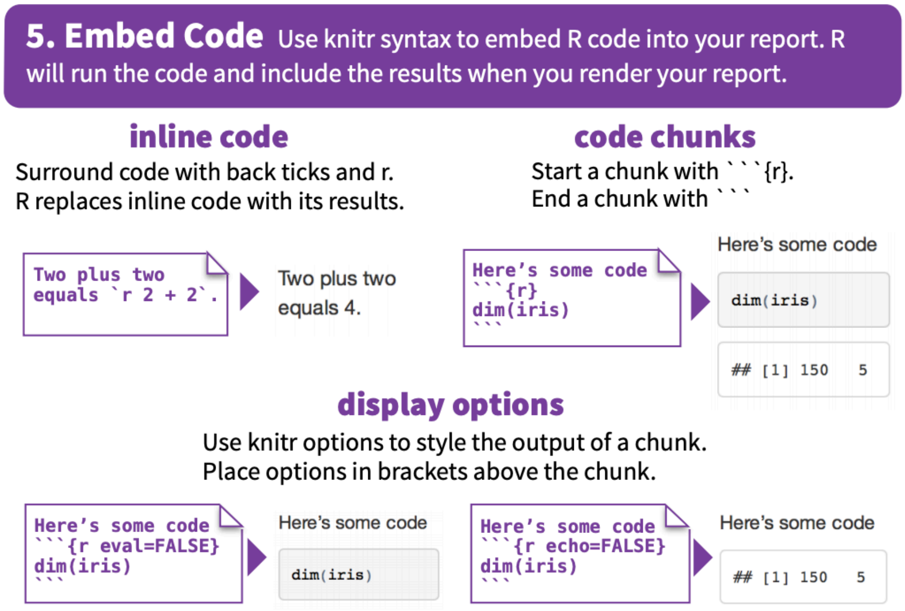
```
## this a bash code snippet
```

There are also many options for your code snippets. Take a look:

3.6 Wrapping (or knitting) it Up

To knit your Markdown file you click the blue *knit* button in your R Studio toolbar. You can choose which file format you would like to knit to as well!

For the purposes of our Merck-Data Mine documentation book, however, we will not have to knit anything because we are placing the individual documents in one bookdown book. To learn more about bookdown take a look at https://bookdown.org/yihui/bookdown/introduction.html for more information.



5. Embed Code Use knitr syntax to embed R code into your report. R will run the code and include the results when you render your report.

inline code

Surround code with back ticks and r. R replaces inline code with its results.

```
Two plus two
equals `r 2 + 2`.
```

Two plus two equals 4.

code chunks

Start a chunk with ```{r}. End a chunk with ```

~~~
Here's some code
```{r}
dim(iris)
```
~~~

Here's some code

```
dim(iris)
```

```
## [1] 150    5
```

display options

Use knitr options to style the output of a chunk. Place options in brackets above the chunk.

~~~
Here's some code
```{r eval=FALSE}
dim(iris)
```
~~~

Here's some code

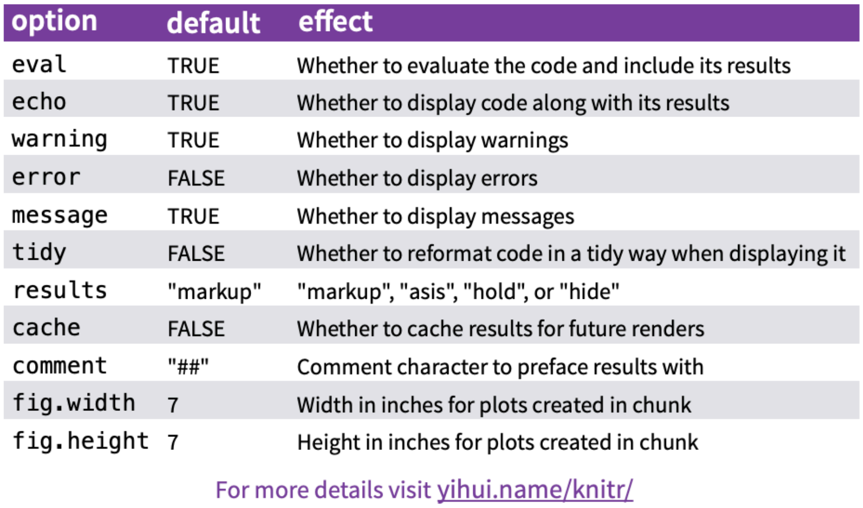
```
dim(iris)
```

~~~
Here's some code
```{r echo=FALSE}
dim(iris)
```
~~~

Here's some code

```
## [1] 150    5
```

Figure 3.4:



option	default	effect
eval	TRUE	Whether to evaluate the code and include its results
echo	TRUE	Whether to display code along with its results
warning	TRUE	Whether to display warnings
error	FALSE	Whether to display errors
message	TRUE	Whether to display messages
tidy	FALSE	Whether to reformat code in a tidy way when displaying it
results	"markup"	"markup", "asis", "hold", or "hide"
cache	FALSE	Whether to cache results for future renders
comment	"##"	Comment character to preface results with
fig.width	7	Width in inches for plots created in chunk
fig.height	7	Height in inches for plots created in chunk

For more details visit yihui.name/knitr/

Figure 3.5:

Chapter 4

Documentation Template

4.1 Introduction

This section will give an overview of what was accomplished during the previous sprint.

4.2 Code

This section is where important code can be documented and commented on. Some teams do not need to include all their code here as this would get excessive. However, important functions or classes should be documented and discussed here.

4.3 Flow Diagrams / Visualizations

This is where teams should showcase flow diagrams of functions, data pipeline, etc. This might also be a good place for visualization/screenshots.

4.4 White Paper

When important decision are made, such as software specifications or product developments, they should be documented in this section. Teams should explain how they came to the conclusion and key takeaways about the decision.

4.5 Technical Report

This section is where teams can highlight the processes of their work. This is a more in depth look into what was accomplised during the sprint where teams should describe step by step development.

Chapter 5

Data Engineering Fall 2020 Documentation

5.1 Introduction to the Team

Our data engineering team is comprised of 6 team members all with unique skills and backgrounds. The team worked hard to utilize each member's strengths to help accomplish the semester goals for this project.

Include a brief sentence including your name, major, year, and relevant experience to the data mine.

Jennifer Leising: I am a 3rd year graduate student in Industrial Engineering. My research is focused on healthcare and data analytics, and I have experience working in Clinical trials for a large Pharma company. Eric Yap: I am sophomore majoring in data science and finance. Most of my classes have helped developed my Python and R skills, which are very transferable to the work I do in the data mine. Allison Hill: I am a senior majoring in electrical engineering. I have coding experience in various languages, including some Python and R which were used within this project. Joshua Kosnoff: I am a junior majoring in biomedical engineering. I perform research with the cancer research center, where I have previously learned Python for data management and basic analysis. Karthik Ravishankar: I am a freshman majoring in data science. I have learned Java in highschool and at Purdue and some Python which was self taught to contribute to the project. Pranav Anandarao: I am a sophomore majoring in computer engineering. I have experience in a few different programming languages, including Python. I also have previous experience working on clinical applications.

5.2 Semester Goals

The overall aim of this project was to create a method to collect continuous data from patients in clinical trials. The biometrics team decided to do this by collecting data from Fitbit devices and developing an application to collect supplementary data from the users.

The data engineering team's task was to collect the data and put it into a usable format for storage with the data architecture team. This semester, our goal was to add functionality to collect data from multiple users and handle data coming in from different devices. Furthermore, we would like to be able to schedule the program to collect the user data periodically.

5.3 Data Overview

5.3.1 Fitbit Devices vs. Apple Watch

The 2019-2020 Merck Biometric team used the Fitbit to gather data for the project. The Merck stakeholder's had also mentioend wanting to explore the Apple Watch to reach additional populations. Our team wanted to better understand what data was available from each company's devices. As a note, we listed all the features that would be available from any device. We created the chart below to list the features available on each device. We found very similar data to be available on both devices.

```
          |                                       |Fitbit          |Apple Watch
```

——————|————————————————|——————|—————— Activity |Steps |X |X |Workouts |X |X |Sit/Stand |X* |NA |Flight Climbed |X |X |Elevation |X |X |Time spent in different activity levels|X |X |Calories burned |X |X Sleep |Asleep vs Awake |X |X |Stages |X |NA Heart Rate |Time spent in different heart ranges |X |3rd party apps |Resting heart rate |X |X |Walking Heartrate |NA |X |Heart rate during activity |X |X |Intraday Tracking |5 sec |10 min Nutrition |Food log* |X |X |Water log* |X |X Body/Weight |BMI* |X |X |Weight* |X |X Additional |ECG |Coming soon |X |Blood Oxygen |X |X |Fall Detection |X |X |Skin Thermometer |X |NA |Ear Health |NA |X |Stress Response |X |NA ———————————————————————————— *Entered manually NA = Not Available

With the Apple Watch, most data tracking is through the Health app and the files are exported as a .xml, which is difficult to read. However, you can also export your data through the Health app and have it opened in Numbers/Excel for easy viewing. For the FitBit, activity is recorded through the watch and outputs a .csv file when exporting, which is easy to read and open in applications like Excel and Numbers.

Additionally, it appears we are able to use Python in order to gather/sort data on both the Apple watch and Fitbit devices. Data is able to be read through pandas. Since Apple Watch returns a .xml file, we have to parse it first and use xmltodict module within Python before we can start applying functions to our data.

Based on the scope for this semester, we did not do further research into the Apple Watch data collection, but this is one of our goal's for the coming Spring semester.

5.3.2 Data Collected from Fitbit Devices

Our team collected data that can be grouped into several major categories:

```
Device: Device Name

Activites: totalDistance, veryActiveDistance, moderatleyActiveDistance, lightlyActiveDistance, ve
lightlyActiveMinutes, sedentaryMinutes, floorsClimbed, daySteps, intraday data

Sleep: sleepEfficiency, minutesAsleep

Heart Rate: HRrange30to100, HRrange100to140, HRrange140to170, HRrange170to220, avgRestingHR, intr

Weight: weight, BMI
```

5.4 Scaling the Framework

The framework for collecting biometric data from Fitbit's API was previously established and can be found in the Merck Wearables Book section 8. However, this framework could only be used to collect one user's data at a time, and needed to be manually monitored to switch between browser and python windows. Further, the framework assumed that the user's Fitbit device would be the Fitbit Ionic. In order to make this project upscalable, the framework needed to be modified to support multiple users in an automizable way and to account for multiple device types.

5.4.1 Multiple Users

5.4.1.1 Supporting Multiple Users Via Accessing Credentials

To support multiple users, a database of multiple user login credentials needed to be created. There is an ongoing collaboration with the Front End team to create a more secure storage system, but for now credentials are stored in a csv file with the format “username,password”. With the usernames and passwords stored in this way, a login array can be created that will be used to plug into the automated process.

```
# Initialize Emails and Passwords lists
Emails = []
Passwords = []
# Create Emails and Passwords lists from Fitbit_Credentials.csv
with open("Fitbit_Credentials.csv") as File1:
    IDs = csv.DictReader(File1)
    for row in IDs:
        Emails.append(row['Username'])
        Passwords.append(row['Password'])
```

As previously mentioned, the previous framework was not set up for running through multiple accounts. To solve this issue, it was reformatted into iterative-friendly components. The first of which is the PythonBot.py file, which both establishes the login arrays outlined above and is responsible for coordinating the other file and function calls to make this process run smoothly.

```
# import other codes
import BiometricPrevious
import gather_keys_oauth2 as Oauth2
class FitbitBot:
    def __init__(self, EMAIL, PASSWORD, DATE):
        #Both the Client ID and Client Secret come from when Fitbit site after registe
        CLIENT_ID = '22BH28'
        CLIENT_SECRET = '78a4838804c1ff0983591e69196b1c46'

        #Authorization Process
        server = Oauth2.OAuth2Server(CLIENT_ID, CLIENT_SECRET)
        server.browser_authorize(EMAIL, PASSWORD)
        ACCESS_TOKEN = str(server.fitbit.client.session.token['access_token'])
        REFRESH_TOKEN = str(server.fitbit.client.session.token['refresh_token'])
        auth2_client = fitbit.Fitbit(CLIENT_ID, CLIENT_SECRET, Oauth2=True, access_toke
        refresh_token=REFRESH_TOKEN)
        BiometricPrev = BiometricPrevious.FitbitModel1(auth2_client)

        biometricDF = BiometricPrev.getBiometricData(DATE) #append to data frame
        biometricDF.to_csv('./user' + str(i) + '_' + DATE + '.csv')
        print("Python Script Executed")
# Run data extraction
for i in range(len(Emails)):
    for j in range(1): # Call the previous 1 days worth of info <-- will be replaced w
        today = str((datetime.datetime.now() - datetime.timedelta(j)).strftime("%Y-%m-
        FitbitBot(Emails[i], Passwords[i], today)
```

The *BiometricPrevious* file contains the remainder of the previous framework with data collection functions, but it was also altered to allow for multiple devices. More information about his can be found in the **Multiple Devices** section of this report. The line *biometric.to_csv* will store the collected data in a csv format with a title of *user#_DATE, where user#* corresponds to the order at which the user's login credentials are stored in the credentials csv database.

5.4.1.2 Supporting Multiple Users Via Allowing for Automation

In order to create an automation bot to regularly call the data collection code (see **Scheduling Our Data Collection** for more information on this), the code had to be adjusted so that it did not open browser windows, as the opening of browser windows was found to stall the bot. In order to accomplish this, the Selenium module was used. Official Selenium documentation can be found here.

Since Selenium was not already downloaded on the device, it needed to be installed. To do this, the following was typed into a Terminal window:

```
pip install selenium
```

To use Selenium in a code, the libraries and packages will need to be imported. Since *gather_keys_oauth2.py* is the only code package responsible for accessing websites, Selenium only needs to be imported into that document. To do this, the following code was added to *gather_keys_oauth2.py*.

```
from selenium import webdriver
from selenium.webdriver import Firefox
from selenium.webdriver.firefox.options import Options
from selenium.webdriver.common.by import By
```

Strictly speaking, the only explicitly needed code would be *import Selenium*. However, by importing specific functions from the Selenium library upfront, notation later on was simplified. This can be seen below when firefox options were set. Instead of calling *Selenium.webdriver.firefox.options* every line, *Options* was able to be called instead.

The appropriate Selenium firefox browser driver was downloaded here.

Note it does not matter where the driver is installed, but its installation path needs to be plugged into the gather_keys_oath2.py code. For those following along with the guide, make a note of the installation path.

The following block of code was written to set the Selenium webdriver functions.

```
firefox_options = Options()
firefox_options.add_argument("window-size=1920,1080")
firefox_options.add_argument("--headless")
firefox_options.add_argument("start-maximized")
firefox_options.add_argument("--disable-infobars")
firefox_options.add_argument("--disable-extensions")
firefox_options.add_argument("--no-sandbox")
firefox_options.add_argument("--disable-dev-shm-usage")
firefox_options.binary_location = '/class/datamine/apps/firefox/firefox'
```

Adjust the path in the final option, *firefox_options.binary_location*, accordingly. The other main option of interest is *firefox_options.add_argument("–headless")*. This *headerless* option is what allows the code to access the Fitbit website without actually opening a browser – effectively, Python becomes a browser simulator. This is the main reason Selenium was used. The other options are for optimal performance, but not strictly necessary.

The first time an account is added, manual permissions will need to be granted to Fitbit's authorization website. To do this, comment out this headerless option (place a "#" before the line), run the *PythonBot.py* code, and check the permissions boxes for each new account as it pops up. After doing this once, uncomment this headerless line. The system will then be ready for full automation.

The official Selenium documentation also supports a Google Chrome driver. If Chrome is preferred, it can be substituted in for FireFox without issue. However, make sure to change all instances of *firefox* in the above codes to *Chrome.*

With Selenium downloaded and imported, the last step was to call and use Selenium. The following line in *browser_authorize* function in *gather_keys_oauth2.py* was responsible for opening the web browser, so it was targeted for edits.

```
    threading.Timer(1, webbrowser.open, args=(url,)).start()
```

As its name might suggest, the *webbrowser.open* function causes a web browser to open. The other passed arguments *1* and *args=(url,)* are parameters that can be left unchanged. This line was replaced with the following code:

```
    driver = webdriver.Firefox(executable_path = '/class/datamine/apps/geckodriver', op
    driver.get("https://accounts.fitbit.com/login?targetUrl=https%3A%2F%2Fwww.fitbit.co
    sleep(5)
    driver.find_element(By.XPATH, "//input[@type='email']").send_keys(email)
    sleep(2)
    driver.find_element(By.XPATH, "//input[@type='password']").send_keys(password)
    sleep(2)
    driver.find_element(By.XPATH, "/html/body/div[2]/div/div[2]/div/div/div[3]/form/div[4]/div/bu
    sleep(10)

    threading.Timer(1, driver.get, args=(url,)).start()
```

The first line of the following code initializes the Selenium webdriver. Edit the *executable_path* as needed based on where the firefox driver was installed. The next line, *driver.get*, tells the Selenium web browser what website to go to. The *find_elements* commands are what allow for the automatization of plugging in usernames and emails. The sleep functions are input time delays to help make sure that the website has time to properly load before attempting to log in. The final line should look familiar. It is the same as the original line, except replacing the *webbrowser.open*, which opens a normal web browser, with *driver.get*, which calls Selenium's browser simulator. The rest of *gather_keys_oauth2* can remain as is.

5.4.1.3 Moving to Scholar

When multiple user logins were pushed and data extractions were performed in relatively quick succession on personal Wi-Fi, Fitbit's cybersecurity software flagged and temporarily banned the IP address from accessing their site. While these bans can be overturned by contacting Fitbit's customer support page on Twitter, the "unbanning" is temporary. This leads to a cycle of getting banned and asking to be unbanned, which is both inconvenient and detrimental to any long term data collection plans. The hope with Scholar is that, as an educational IP address, it can be whitelisted and the automated process can function without getting users banned.

5.4.1.3.1 Using Scholar

To access Scholar, go to *https://scholar-fe03.rcac.purdue.edu:300/main/* and login with Purdue 2 Factor Credentials.

The codes used for this part of the project are written in Python. Unfortunately, Scholar does not currently support Python IDEs (Spyder, VS code, etc). This means that the code either needs to be called through a Jupyter Notebook Kernel or through a terminal window. For now, the code will be run via Terminal commands.

To open Terminal, click on **Terminal Emulator** in Scholar's applications bar. Once Terminal is open, navigate to the */class* directory and then to the appropriate folder. To do this, type the following commands into Terminal:

```
cd ..
cd ..
cd /class/datamine/corporate/merck/DataEngineers/Fitbit
```

Needed files and libraries have been installed on Scholar. Pathways within the code have been adjusted to Scholar's directory. As a result, there is only one more step that needs to be done before the code can be run. This step is to direct the python3 program where to find installed libraries. Type the following command into the Terminal:

```
source /class/datamine/apps/python.sh
```

Note You will need to rerun this command everytime you close out of Terminal and reopen it.

Now, Scholar is ready to run PythonBot.py! To run the code, type the following into Terminal:

```
python3 PythonBot.py
```

Note if the previous Terminal window was closed, navigate to */class/datamine/corporate/merck/DataEng* again before running *python3 PythonBot.py*

Extracted data is formatted into CSV files and accessible at */class/datamine/corporate/merck/DataEngin*

5.4.2 Multiple Devices

Not all fitbit devices have the same data available to them. This is most noticeable in older models, in which functionality such as the number of floors climbed or sleep cycles was not yet implemented. Using the get_Devices function available in the Fitbit API, the code returns a string corresponding to the Fitbit version. For example, a Fitbit Ionic would return the string "Ionic". Using this information, if-else statements were implemented to check the version and assign corresponding data values. Data that is not available would be input as "None" within the data table. In some cases, the function can simply check if the data is NULL and automatically do this such as with the sleep data. However for data such as activity levels, the data gets input as a "0" instead and thus needs a check. Checks for some of the newer models have been implemented, along with a generic case. The table below shows the implemented models and their available data: | |Ionic | Versa Lite |Inspire |Charge 3 | ————|———— ————————|————|————-|————-|————-| Activity |Steps |X |X |X |X | |Workouts |X |X | | | |Sit/Stand |X |X | | | |Flight Climbed |X | | | | |Elevation |X | | | | |Time spent in different activity levels|X |X | | | |Calories burned |X |X | | | Sleep |Asleep vs Awake |X |X |X |X | |Stages |X |X | | |

Heart Rate |Time spent in different heart ranges |X |X | |X | |Resting heart rate |X |X | |X | |Walking Heartrate |X |X | |X | |Heart rate during activity |X |X | |X |

The following function is what is used to obtain the device model string:

```
def getDevice():
    devices = auth2_client.get_devices()
    if(len(devices) == 0):
        deviceVersion = 'None'
    else:
        deviceVersion = devices[0]['deviceVersion']
    return deviceVersion
```

The function first calls the built-in Fitbit API get_devices(). Next it checks the length of the output to determine if it is valid. If it is not valid, the user must not have any device registered. At the moment, the code only obtains the first device if a user has multiple registered.

5.5 Scheduling the Data Collection

Through using the CronTab module within Python, we are able to schedule and run our FitBit functions on a routinely basis. The data we collect is stored in a dataframe and new data is continuously appended to this dataframe. The scheduled time is set to every 9 AM on weekdays.

5.6 Collaborating with other Teams

Our team had three main interactions with the other teams. The first one was with the Data Architects. When reviewing last year's code, we noticed that sending a CSV file everytime would be inneficient. We discovered a function, dataFrame.to_sql(), which would allow the data to be sent straight into the SQL server. Although a solution was found, we decided to leave the code how it is and save this idea for another day. The second and third interactions were with the Front End team. The first time, we wanted to see if there was a way to easily collect the user's fitbit username and password. The Front End team was able to create a way to have the login information collected from mobile app and saved on the SQL server with the help of the Back End team. This was a very important problem to solve as the API needs to log into the user's fitbit account to pull their data. Our last collaboration was regarding a issue that would result in complications during the clinical trial. Regardless of whether the user is wearing their device or not, any time the fitbit wasn't in an active motion, time is accrued on the user's sedentary minutes. This would be a problem as there is no way to differentiate whether the user was sedentary, or if the fitbit wasn't on the user. After conversing with the Front End team, we decided that an alert system or question during their survey would help solve this issue. This problem is a work in progress and doesn't have a solution as of yet.

5.7 Future Work

Our future work for the Spring 2021 semester seeks to build upon our work completed during this Fall 2020 semester. Our main goals are listed include: (1) streamlining our work with the other Merck biometric teams to ensure the entire process works well from start to finish, (2) exploring alternative ways to access the user's fitbit information that might better align with the way fitbit intends multiple user data collection, and (3) investigating the collection of data from the Apple Watch.

(1) Streamlining work with the other Merck Biometric teams
(2) Exploring alternative ways to access the user's fitbit information
(3) Investigating the collection of data from the Apple Watch

Chapter 6

Data Engineering Spring 2021 Documentation

6.1 Completing the Pipeline

In order to connect data collection process to the rest of the pipeline, it had to be able to read patient information and login credentials from the SQL database, as well as automatically append the collected biometric data to a database. In order to accomplish this, the PythonBot.py code had to be able to connect to SQL. This was done by first importing the necessary sqlachemy pacakges and then establishing a connection to our specific database.

To import packages:

```
# using sqlalchemy
import sqlalchemy as sal
from sqlalchemy import create_engine
import pandas as pd
```

To establish connection to SQL database:

```
# establish connecttion URL
conn = "mysql+pymysql://{0}:{1}@{2}:{3}/{4}".format(
    'cb3i17t0aqn6a4ff', 'e2l4k9zn24shcj42', 'rnr56s6e2uk326pj.cbetxkdyhwsb.us-east-1.rds.amazonaw

# create engine
engine = sal.create_engine(conn)
```

If another SQL database is used in the future, please note that 'cb3i17t0aqn6a4ff' corresponds to 'username', 'e2l4k9zn24shcj42' corresponds to 'password', 'rnr56s6e2uk326pj.cbetxkdyhwsb.us-east-1.rds.amazonaws.com' corresponds to 'address', '3306' corresponds to 'port', and 'lfry112yqr3k2dfr' corresponds to 'DB'. Change these values as appropriate in the code.

6.1.1 Reading in user data from patient table

Patient data was imported as a series of arrays. The SQL patient table was read in as df1, which was used to form lists of data corresponding to individual patient's emails, passwords, IDs, and usernames.

```
df1 = pd.read_sql_query("SELECT * FROM patient", engine)
emails = []
passwords = []
IDs = []
usernames = []
for i in range(len(df1)):
    emails.append(df1.email[i])
    passwords.append(df1.fitbit_password[i])
    IDs.append(df1.patient_id[i])
    usernames.append(df1.username[i])
```

As was previously the case, this data was iterated through and plugged into the FitbitBot function.

6.1.2 Appending user data to SQL table

Once data was collected (collection functions can be found in *BiometricPrevious_getDevice_v2.py*), it needed to be appended to the SQL database. A new function was made to reformat the data to have the same column names as the SQL table.

```
def appendDataBase(DATA,ENGINE, USER, ID):
    obj = pd.DataFrame()

    obj['patient_id'] = [ID]
    obj['fbusername'] = [USER]
    obj['collection_date'] = [DATA.get("Date")]
    obj['steps'] = [DATA.get("Steps")]
    obj['floors_climbed'] = [DATA.get("Floors Climbed")]
    obj['total_miles'] = [DATA.get("Total Miles")]
    obj['lightly_active_miles'] = [DATA.get("Lightly Active Miles")]
    obj['moderately_active_miles'] = [DATA.get("Moderately Active Miles")]
    obj['very_active_miles'] = [DATA.get("Very Active Miles")]
    obj['sedentary_minutes'] = [DATA.get("Sedentary Minutes")]
    obj['lightly_active_minutes'] = [DATA.get("Lightly Active Minutes")]
    obj['fairly_active_minutes'] = [DATA.get("Fairly Active Minutes")]
    obj['very_active_minutes'] = [DATA.get("Very Active Minutes")]
    obj['hr30_100_minutes'] = [DATA.get("HR 30-100 Minutes")]
    obj['hr100_140_minutes'] = [DATA.get("HR 100-140 Minutes")]
    obj['hr140_170_minutes'] = [DATA.get("HR 140-170 Minutes")]
    obj['hr170_220_minutes'] = [DATA.get("HR 170-220 Minutes")]
    obj['average_resting_hr'] = [DATA.get("Average Resting HR")]
    obj['bmi'] = DATA.get("BMI")
    obj['sleep_efficiency'] = DATA.get("Sleep Efficiency")
    obj['weight'] = DATA.get("Weight")
    obj["minutes_asleep"] = todaysData.get("Minutes Alseep")

    obj.to_sql("fitbit_data", con=ENGINE, if_exists='append', index= False)

    return(None)
```

In this function, the argument *DATA* is a dataframe returned from *biometric_previous_getDevice_v2.py*, *ENGINE* is the connection engine to the SQL database, and *USER* and *ID* correspond to the user's Fitbit username and Merck trial userID. The *to_sql* function at the end is responsible for exporting the reformatted data to the *"fitbit_data"* table in the SQL database. The *if_exists='append'* argument is responsible for appending to the database instead of overwriting it, and the *index=False* argument stops the code from creating the original indexing as a column in SQL.

More documentation on .to_sql can be found here: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_sql.html

In order to pass the userID and fitbit username to the SQL database, the FitbitBot function and function call had to be slightly altered.

```
class FitbitBot:
    def __init__(self, EMAIL, PASSWORD, DATE, usernames, ID):

        #Both the Client ID and Client Secret come from when Fitbit site after registering an app
        CLIENT_ID = '22BH28' #Mine:'22BKP3'
        CLIENT_SECRET = '78a4838804c1ff0983591e69196b1c46' #Mine:'1a42e97b6b4cc640572ae5cf10a7d0b
        #Authorization Process
        # opens website
        server = Oauth2.OAuth2Server(CLIENT_ID, CLIENT_SECRET)
        # opens website
        server.browser_authorize(EMAIL, PASSWORD)

        ACCESS_TOKEN = str(server.fitbit.client.session.token['access_token'])
        REFRESH_TOKEN = str(server.fitbit.client.session.token['refresh_token'])
        auth2_client = fitbit.Fitbit(CLIENT_ID, CLIENT_SECRET, Oauth2=True, access_toke
        refresh_token=REFRESH_TOKEN)
        BiometricPrev = BiometricPrevious.FitbitModel1(auth2_client)
        bioDict, biometricDF = BiometricPrev.getBiometricData(DATE) #append to data fr
        title = './CSV_Files/user' + str(i) + '_' + DATE + '.csv'

        appendDataBase(bioDict,engine,usernames,ID)
        print("Python Script Executed")
```

usernames and ID arguments were added to the function. The appendDataBase function call was also added into the function, and BiometricPrev.getBiometricData(DATA) was slightly altered to return both a dictionary and a database.

Since the function arguments were exanded, the function call also had to be adjusted to pass usernames[i] and IDs[i].

```
today = str((datetime.datetime.now() - datetime.timedelta(1)).strftime("%Y-%m-%d"))
# Run data extraction
for i in range(len(emails)):
    FitbitBot(emails[i], passwords[i], today, usernames[i], IDs[i])
```

6.2 Thank you & Acknowledgements

We would like to thank our Merck Corporate Partners, TA, and all the Data Mine staff that have helped us throughout this semester.

Chapter 7

Data Architects

7.1 Background

What is a database?

A database is an organized collection of data stored electronically. It is typically controlled by a database management system (DBMS). There are two main categories a database can fall into: relational and non-relational. A relational database is one that stores data in rows and columns, otherwise known as a table. While a simple database may include only one table, a typical functional database contains multiple tables. These tables are linked to each other using keys. A primary key is a unique identifier and is referenced by other tables. When in another table, it is considered a foreign key. While it is not necessary, it is common practice to include a primary key in every table. Nearly all relational databases rely on the programming language SQL (Structured Query Language) to query, manipulate, and define data, as well as proved access control. The term “non-relational database” is a blanket term for all other databases that do not rely on SQL, hence their other name, NoSQL. These databases can take on many forms: key-value store, document store, column-oriented databases, and graph databases, to name a few.

7.2 Database Design

Designing a relational database is straightforward. First, the data that will go into the database must be known, along with the datatypes. Next, an entity relationship diagram (ERD) should be constructed. This will help keep the design of the database organized. There are plenty of excellent free tools on the internet that help with design and construction. Lucidchart is one of these tools. Lucidchart allows one to design an ERD and then export it as an SQL script! This can be handy, especially when first starting out with SQL. Below is the ERD for the current (at the time of writing this) working database. PK stands for primary key, and FK stands for foreign key. This ERD includes the datatypes as well, which is necessary for properly creating a database.

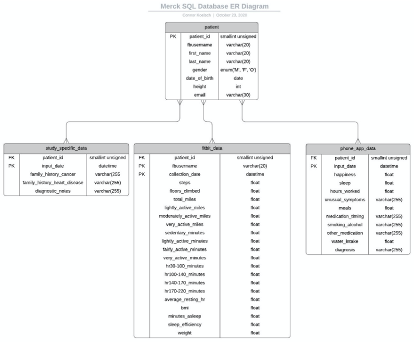
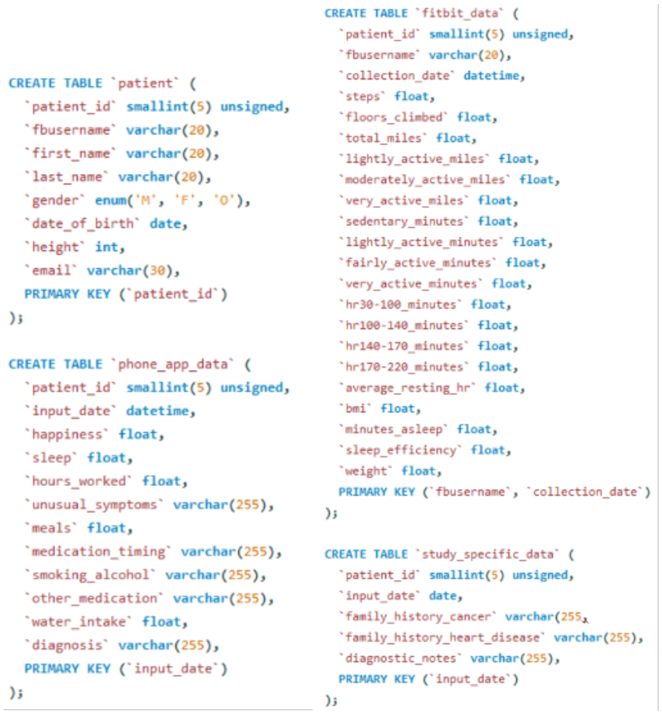

Merck SQL Database ER Diagram
Connor Koelsch | October 23, 2020
patient
PK patient_id smallint unsigned
fbusername varchar(20)
first_name varchar(20)
last_name varchar(20)
gender enum('M', 'F', 'O')
date_of_birth date
height int
email varchar(30)
study_specific_data
FK patient_id smallint unsigned
PK input_date datetime
family_history_cancer varchar(255
family_history_heart_disease varchar(255)
diagnostic_notes varchar(255)
fitbit_data
FK patient_id smallint unsigned
PK fbusername varchar(20)
PK collection_date datetime
steps float
floors_climbed float
total_miles float
lightly_active_miles float
moderately_active_miles float
very_active_miles float
sedentary_minutes float
lightly_active_minutes float
fairly_active_minutes float
very_active_minutes float
hr30-100_minutes float
hr100-140_minutes float
hr140-170_minutes float
hr170-220_minutes float
average_resting_hr float
bmi float
minutes_asleep float
sleep_efficiency float
weight float
phone_app_data
FK patient_id smallint unsigned
PK input_date datetime
happiness float
sleep float
hours_worked float
unusual_symptoms varchar(255)
meals float
medication_timing varchar(255)
smoking_alcohol varchar(255)
other_medication varchar(255)
water_intake float
diagnosis varchar(255)

Figure 7.1:

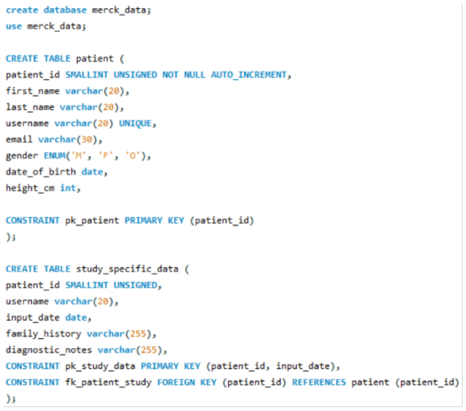
Database Creation with SQL Exporting the ERD above results in the following SQL code:



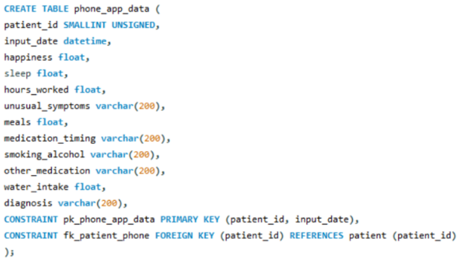
```
CREATE TABLE `patient` (
  `patient_id` smallint(5) unsigned,
  `fbusername` varchar(20),
  `first_name` varchar(20),
  `last_name` varchar(20),
  `gender` enum('M', 'F', 'O'),
  `date_of_birth` date,
  `height` int,
  `email` varchar(30),
  PRIMARY KEY (`patient_id`)
);

CREATE TABLE `phone_app_data` (
  `patient_id` smallint(5) unsigned,
  `input_date` datetime,
  `happiness` float,
  `sleep` float,
  `hours_worked` float,
  `unusual_symptoms` varchar(255),
  `meals` float,
  `medication_timing` varchar(255),
  `smoking_alcohol` varchar(255),
  `other_medication` varchar(255),
  `water_intake` float,
  `diagnosis` varchar(255),
  PRIMARY KEY (`input_date`)
);
```

```
CREATE TABLE `fitbit_data` (
  `patient_id` smallint(5) unsigned,
  `fbusername` varchar(20),
  `collection_date` datetime,
  `steps` float,
  `floors_climbed` float,
  `total_miles` float,
  `lightly_active_miles` float,
  `moderately_active_miles` float,
  `very_active_miles` float,
  `sedentary_minutes` float,
  `lightly_active_minutes` float,
  `fairly_active_minutes` float,
  `very_active_minutes` float,
  `hr30-100_minutes` float,
  `hr100-140_minutes` float,
  `hr140-170_minutes` float,
  `hr170-220_minutes` float,
  `average_resting_hr` float,
  `bmi` float,
  `minutes_asleep` float,
  `sleep_efficiency` float,
  `weight` float,
  PRIMARY KEY (`fbusername`, `collection_date`)
);

CREATE TABLE `study_specific_data` (
  `patient_id` smallint(5) unsigned,
  `input_date` date,
  `family_history_cancer` varchar(255,
  `family_history_heart_disease` varchar(255),
  `diagnostic_notes` varchar(255),
  PRIMARY KEY (`input_date`)
);
```

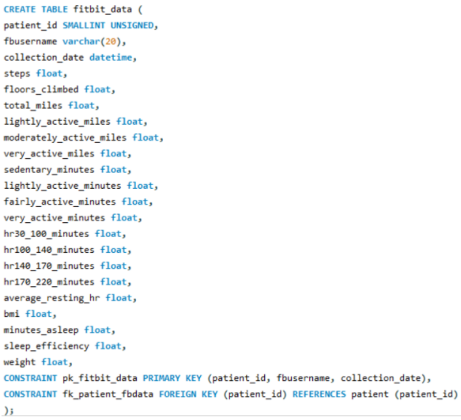
However, this code lacks a few required pieces. The foreign keys did not populate, and the primary keys do not reference the other tables. We also have not "created" a database yet, nor told our program which database to use. These issues are fixed in the following code snippet, along with some syntax changes. These syntax changes were necessary because the server our database is hosted on has some requirements that are not very important (at least not right now).



```
create database merck_data;
use merck_data;

CREATE TABLE patient (
patient_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
first_name varchar(20),
last_name varchar(20),
username varchar(20) UNIQUE,
email varchar(30),
gender ENUM('M', 'F', 'O'),
date_of_birth date,
height_cm int,

CONSTRAINT pk_patient PRIMARY KEY (patient_id)
);

CREATE TABLE study_specific_data (
patient_id SMALLINT UNSIGNED,
username varchar(20),
input_date date,
family_history varchar(255),
diagnostic_notes varchar(255),
CONSTRAINT pk_study_data PRIMARY KEY (patient_id, input_date),
CONSTRAINT fk_patient_study FOREIGN KEY (patient_id) REFERENCES patient (patient_id)
);
```



```
CREATE TABLE phone_app_data (
patient_id SMALLINT UNSIGNED,
input_date datetime,
happiness float,
sleep float,
hours_worked float,
unusual_symptoms varchar(200),
meals float,
medication_timing varchar(200),
smoking_alcohol varchar(200),
other_medication varchar(200),
water_intake float,
diagnosis varchar(200),
CONSTRAINT pk_phone_app_data PRIMARY KEY (patient_id, input_date),
CONSTRAINT fk_patient_phone FOREIGN KEY (patient_id) REFERENCES patient (patient_id)
);
```



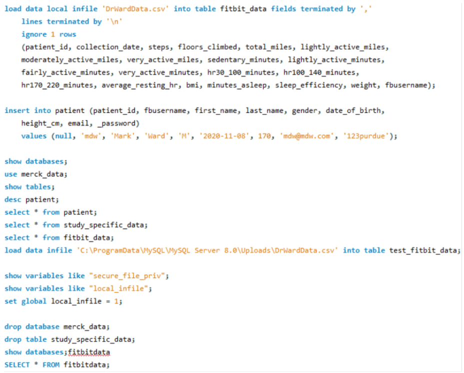
```
CREATE TABLE fitbit_data (
patient_id SMALLINT UNSIGNED,
fbusername varchar(20),
collection_date datetime,
steps float,
floors_climbed float,
total_miles float,
lightly_active_miles float,
moderately_active_miles float,
very_active_miles float,
sedentary_minutes float,
lightly_active_minutes float,
fairly_active_minutes float,
very_active_minutes float,
hr30_100_minutes float,
hr100_140_minutes float,
hr140_170_minutes float,
hr170_220_minutes float,
average_resting_hr float,
bmi float,
minutes_asleep float,
sleep_efficiency float,
weight float,
CONSTRAINT pk_fitbit_data PRIMARY KEY (patient_id, fbusername, collection_date),
CONSTRAINT fk_patient_fbdata FOREIGN KEY (patient_id) REFERENCES patient (patient_id)
);
```

The following snippets are useful commands for handling the database.

7.3 Database Creation with Neo4j

7.3.1 Neo4j Background

Neo4j is a graph database created by Neo4j inc. Neo4j is implemented in Java; however, it can also be written with any other cypher query language.



```
load data local infile 'DrWardData.csv' into table fitbit_data fields terminated by ','
    lines terminated by '\n'
    ignore 1 rows
    (patient_id, collection_date, steps, floors_climbed, total_miles, lightly_active_miles,
    moderately_active_miles, very_active_miles, sedentary_minutes, lightly_active_minutes,
    fairly_active_minutes, very_active_minutes, hr30_100_minutes, hr100_140_minutes,
    hr170_220_minutes, average_resting_hr, bmi, minutes_asleep, sleep_efficiency, weight, fbusername);

insert into patient (patient_id, fbusername, first_name, last_name, gender, date_of_birth,
    height_cm, email, _password)
    values (null, 'mdw', 'Mark', 'Ward', 'M', '2020-11-08', 170, 'mdw@mdw.com', '123purdue');

show databases;
use merck_data;
show tables;
desc patient;
select * from patient;
select * from study_specific_data;
select * from fitbit_data;
load data infile 'C:\ProgramData\MySQL\MySQL Server 8.0\Uploads\DrWardData.csv' into table test_fitbit_data;

show variables like "secure_file_priv";
show variables like "local_infile";
set global local_infile = 1;

drop database merck_data;
drop table study_specific_data;
show databases;fitbitdata
SELECT * FROM fitbitdata;
```

Figure 7.2:

7.4 Code

This creates the first two nodes. The yellow highlight section would be the user node. The green highlight section shows the relationship between the two nodes. The orange highlight section shows the other node which would be the username. The light blue highlighted section returns and stores the two nodes into the system.

```
1. create(u:user {name:'User'})-[:makes]->(n:user{name:'Username'}) return u,n
```

The yellow highlighted section codes the user node again. Instead of creating a new node for User we can just match the node with the new node you want to create. The green highlighted section describes the relationships between the two nodes. The blue highlighted section is the code for the new node. The orange highlighted section is the code for return which would store the nodes into the system.

```
2. match(u:user {name:'User'}) create (u)-[:enters]-> (DA:user{name: 'Date of Birth'})
```

This line of code codes for a new node “Last Name” which stores the last name of the patient. This node holds a relationship with the User node.

```
3. match(u:user {name:'User'}) create (u)- [:enters]-> (LN:user{name:'Last Name'}) retu
```

Similar to the code on the third line. A new node has been created, however this node is created to store the last name of the user.

```
4. match(u:user {name:'User'}) create (u)- [:enters]-> (FN:user{name:'First Name'}) return u,FN
```

A new node is created with this code. The node stores the Height of the user.

```
5. match(u:user {name:'User'}) create (u)- [:enters]-> (H:user{name:'Height'}) return u,H
```

The ID node is created, thus allowing for the user id to be stored.

```
6. match(u:user {name:'User'}) create (u)- [:enters]-> (I:user{name:'ID'}) return u,I
```

This line of code codes for a new node “Merck Data” which stores the last name of the patient. This node holds a relationship with the ID node.

```
7. match(I:user{name:'ID'}) create (I)- [:enters]-> (M:user{name:'Merck Data'}) return I,M
```

This line of code codes for a new node “Family History Cancers” which stores the family history cancers of the patient. This node holds a relationship with the Merck Data node.

```
8. match(M:user{name:'Merck Data'}) create (M)- [:enters]-> (FH:user{name:'Family History Cancers
```

Lines 9, 10, and 11 all code for three new nodes “Diagnostic notes”, “ Any other family history disease”, and “Input Data”. This node holds a relationship with the Merck Data node.

```
9.  match(M:user{name:'Merck Data'}) create (M)- [:enters]-> (ID:user{name:'Input Data'}) return
10. match(M:user{name:'Merck Data'}) create (M)- [:enters]-> (DN:user{name:'Diagnostic Notes'}) r
11. match(M:user{name:'Merck Data'}) create (M)- [:enters]-> (AD:user{name:'Any other family hist
```

This line of code helps put together all of the nodes. This is to help you check your database while you add new nodes. This line of code will be used throughout to double check the code, and to endure every branch is in the correct spot.

```
12. MATCH (n:user) RETURN n LIMIT 100
```

This code creates the relationship between the node “Username” and the new node “fitbit data”. The new node will store fitbit data.

```
13. match(n:user{name:'Username'}) create (n)- [:enters]-> (FD:user{name:'FitBit Data'}) return n
```

Lines 14,15,16,17,18, and 19 all create new nodes with the fit bit data node. All the new nodes are highlighted in yellow.

```
14. match(FD:user{name:'FitBit Data'}) create (FD)- [:enters]-> (WE:user{name:'Weight'
15. match(FD:user{name:'FitBit Data'}) create (FD)- [:enters]-> (CD:user{name:'Collect
16. match(FD:user{name:'FitBit Data'}) create (FD)- [:enters]-> (B:user{name:'BMI'}) r
17. match(FD:user{name:'FitBit Data'}) create (FD)- [:enters]-> (FC:user{name:'Floors 
18. match(FD:user{name:'FitBit Data'}) create (FD)- [:enters]-> (ARH:user{name:'Averag
19. match(FD:user{name:'FitBit Data'}) create (FD)- [:enters]-> (ST:user{name:'Steps'}
```

This line of code codes for a new node "Miles" which stores the miles of the patient. This node holds a relationship with the Fitbit Data Node

```
20. match(FD:user{name:'FitBit Data'}) create (FD)- [:enters]-> (Mi:user{name:'Miles'}
```

Lines 20,21,22,23,24,25 all code for new nodes that hold relationships with the Fitbit node.

```
21. match(FD:user{name:'FitBit Data'}) create (FD)- [:enters]-> (TM:user{name:'Total M
22. match(FD:user{name:'FitBit Data'}) create (FD)- [:enters]-> (VAM:user{name:'Very A
23. match(FD:user{name:'FitBit Data'}) create (FD)- [:enters]-> (LAM:user{name:'Lightl
24. match(FD:user{name:'FitBit Data'}) create (FD)- [:enters]-> (MAM:user{name:'Modera
25. match(FD:user{name:'FitBit Data'}) create (FD)- [:enters]-> (S:user{name:'Sleep'})
```

Lines 26,27 all codes for new nodes that holds relationships with the "Sleep node".

```
26.  match(S:user{name:'Sleep'}) create (S)- [:enters]-> (SE:user{name:'Sleep Efficien
27. match(S:user{name:'Sleep'}) create (S)- [:enters]-> (MAS:user{name:'Minutes Asleep
```

Lines 28,29,30,31,32,33,34,35,36 all codes for new nodes that hold relationships with the "minute node".

```
28. match(FD:user{name:'FitBit Data'}) create (FD)- [:enters]-> (M:user{name:'Minutes'
29. match(M:user{name:'Minutes'}) create (M)- [:enters]-> (OF:user{name:'hr 140-170'})
30. match(M:user{name:'Minutes'}) create (M)- [:enters]-> (SM:user{name:'Sedentary Min
31. match(M:user{name:'Minutes'}) create (M)- [:enters]-> (LAM:user{name:'Lightly Acti
32. match(M:user{name:'Minutes'}) create (M)- [:enters]-> (FAM:user{name:'Fairly Activ
33. match(M:user{name:'Minutes'}) create (M)- [:enters]-> (VAM:user{name:'Very Active 
34. match(M:user{name:'Minutes'}) create (M)- [:enters]-> (HR:user{name:'hr 30-100'}) 
35. match(M:user{name:'Minutes'}) create (M)- [:enters]-> (HR1:user{name:'hr 100-140'}
36. match(M:user{name:'Minutes'}) create (M)- [:enters]-> (HR2:user{name:'hr 170-220'}
```

Lines 37-49 all codes for new nodes that hold relationships with the "Pain Data" node. The highlighted portions are the new nodes.

```
37. MATCH (n:user) RETURN n LIMIT 100
38. match(u:user {name:'User'}) create (u)- [:enters]-> (PD:user{name:'Pain Data'}) return u,PD
39. match(PD:user{name:'Pain Data'}) create (PD)- [:enters]-> (D:user{name:'Diagnosis'}) return P
40. match(PD:user{name:'Pain Data'}) create (PD)- [:enters]-> (WI:user{name:'Water Intake'}) retu
41. match(PD:user{name:'Pain Data'}) create (PD)- [:enters]-> (MT:user{name:'Medication Timing'})
42. match(PD:user{name:'Pain Data'}) create (PD)- [:enters]-> (DA:user{name:'Date'}) return PD,DA
43. match(PD:user{name:'Pain Data'}) create (PD)- [:enters]-> (HA:user{name:'Happiness'}) return
44. match(PD:user{name:'Pain Data'}) create (PD)- [:enters]-> (OM:user{name:'Other Medication'})
45. match(PD:user{name:'Pain Data'}) create (PD)- [:enters]-> (SL:user{name:'Sleep'}) return PD,S
46. match(PD:user{name:'Pain Data'}) create (PD)- [:enters]-> (HW:user{name:'Hours Worked'}) retu
47. match(PD:user{name:'Pain Data'}) create (PD)- [:enters]-> (US:user{name:'Unusual Symptoms'})
48. match(PD:user{name:'Pain Data'}) create (PD)- [:enters]-> (M:user{name:'meals'}) return PD,M
49. match(PD:user{name:'Pain Data'}) create (PD)- [:enters]-> (SA:user{name:'Smoking/Alcohol'}) r
```

Chapter 8

Data visualization Team

8.1 Biometric Wearables

This project is a part of the Purdue data science initiative, where students are given an opportunity to work on the data-driven project, guided by an industry mentor. This Biometric Wearables project is a part of an ongoing collaboration between the Purdue and Merck industry. Following is the brief description of the project:

8.2 Purpose

The purpose of the project is to create an automated system that will capture the biometric data from wearable fitness technology using the mobile application and communicate its' result to Merck scientists.

8.3 Outcome

This automated system will be able to collect accurate and less biased data from the clinical trial patients because the digital process reduces the human error and reduce the pressure on the clinical trial patient with manual processing procedure.

8.4 Significance

This project may help to reduce the experiment biased by collecting more extensive data and using the technology. Also, this approach may help to reduce the clinical trial time and cost to ease of data collection.

8.5 Overview of Data visualization Team

This team is a part of a big team working on the above-described project, part of the Merck-Purdue data science collaboration. This team has been assigned the responsibility to design rich, interactive graphics, data visualization, and plotting method to facilitate the MERCK scientists. In addition, this team is responsible for brainstorming the data, and experiments with different visualization techniques that can help the project.

8.6 Previous Work Done on the project

The previous team that worked on the data visualization aspect of this project created a baseline shiny website that had some specific widgets created. There are two pages on the shiny website i.e., activity progress and step goals. The activity progress has a bar chart that shows the progress by comparing two biometric data points. The bar chart has an x-axis which is to show the dates and a y-axis which is to show the steps. One of the widget's features on this page was created to view a specific time period of the data. The second page is to show step goals using a pie chart.

8.7 Team Members

Following is the brief introduction of the data visualization team members:

8.7.1 Ahmed Ashraf Butt

He is a doctoral student at the School of Engineering Education, Purdue University. He is currently working as a research assistant on the CourseMIRROR project funded by the Institute of Education Sciences (IES). He is interested in designing educational tools and exploring their impact on enhancing students' learning experiences. Also, he is working on the Merck-Purdue data science collaboration responsible for creating an automated system that will capture the biometric data from the wearable fitness technology using the mobile application and communicate its result to Merck scientists.

8.7.2 Pika Seth

She is a second-year student in the College of Engineering studying Biomedical Engineering at Purdue University. She is currently involved with the Data Mine Corporate Program working on the Biometrics Wearable Project with Merck. She is involved with the data visualization aspect dealing with the programming of R Shiny. She is a part of the Red Cross Club and the Women in Engineering Program. Also, she is a part of the Sigma Delta Tau sorority.

8.7.3 Sharon Patta

She is a first-year student in the College of Science at Purdue University. She is studying data science. Her interests are in the fields of AI and machine learning.

8.7.4 Surya Suresh

He is a first-year student in the College of Science at Purdue University studying Computer Science. He is working with the Merck Data Mine group on the Data Visualization part of the project.

8.8 Team formation

The team has worked together to become more familiar with R programming and shiny application. Each member of the group explored different data visualizations and then, worked colloboratelу to implement a new visualization or improving the previous shiny app. Additionally, the group had several meetings to collectively present findings to one another and discuss errors and troubleshooting.

8.9 Term goals

We are going to begin working on adding more features to the old Shiny App as well as implementing our own visualizations that we think would be useful. We also need to figure out where we are getting our data from once the API is running because as of right now we are just taking it off of the CSV file that has Dr. Ward's data.

8.10 Visulization 1

In this visualization, we were able to create a visualization that adapts to the given data and creates a Data Table with all of the given data. We used the Data Tables library to accomplish this. The Data Table contains two main tabs. One tab contains all of the mean values of the given data. The other tab allows the user to navigate through all of the data points in a clean and efficient way.Specifically it allows the user to filter the data as well as list the data in any desired order. For example, the data can be listed in order from the days with the most steps to the days with the least. ### Code

I imported the libraries needed for the vizualization

```r
library(shiny) #Used for Shiny Dashboard
library(ggplot2) #Needed for Plots (Not specifically this Viz)
library(reshape2) #Needed for Plots (Not specifically this Viz)
library(DT) #Used for the Data Tables used in this Vizualization
```

biometric data 'bioFinal.csv' for biometric data from 2020 Feb. to present? - Merckteam's data 'DrWardData.csv' for biometric data from 2016 to 2019 April - Dr. Ward's data

Used to read data into the myDF data frame

```r
myDF <- read.csv("liveMerck.csv",TRUE,",")
```

active miles (stacked)

```r
miles1 <- subset(myDF, select = c('Date','Lightly.Active.Miles',
                                  'Moderately.Active.Miles',
                                  'Very.Active.Miles'))
miles <- melt(miles1)
miles$Date <- as.Date(miles$Date)
```

active minutes (stacked)

```r
minutes1 <- subset(myDF, select = c('Date', 'Sedentary.Minutes',
                                    'Lightly.Active.Minutes',
                                    'Fairly.Active.Minutes',
                                    'Very.Active.Minutes'))
minutes <- melt(minutes1)
minutes$Date <- as.Date(minutes$Date)
```

HR minutes (stacked)

```
hr1 <- subset(myDF, select = c('Date', 'HR.30.100.Minutes',
                               'HR.100.140.Minutes', 'HR.140.170.Minutes',
                               'HR.170.220.Minutes'))
hr <- melt(hr1)
hr$Date <- as.Date(hr$Date)
```

Finding out the collums index with numeric value in it.

```
columsIndexWithNumberValue <- unlist(lapply(myDF, is.numeric))
```

Extracting the columns with number value in it.

```
columWithNumber <- myDF[ , columsIndexWithNumberValue]
```

finding the mean of all collum's mean

```
meansOFColumWithNumber <- lapply(columWithNumber[,], mean,na.rm=TRUE)
```

Converting them back into data frame.

```
meansOFColumWithNumberDataFrame<-data.frame(matrix(unlist(meansOFColumWithNumber), nrow=length(me
```

Extracting the name of the collums names from columWithNumber.

```
colName<- data.frame(names(columWithNumber))
```

Extracting the name of the collums.

```
finalDataFrame <- data.frame(a=colName, c=meansOFColumWithNumberDataFrame)
```

changing the Collum name of the final dataframe

```
colnames(finalDataFrame)[1]<-"Variables"
colnames(finalDataFrame)[2]<-"Mean Values"
```

myDF to date format

```
myDF$Date = as.Date(myDF$Date)
```

Remove the X column from myDF

```
drops <- c("X")
myDF <- myDF[ , !(names(myDF) %in% drops)]
```

UI of the Application

```
# Uses the fluidPage and a Sidebar Layout
ui <- fluidPage(
  title = "Tester",
  titlePanel("Visualization 1"),
  sidebarLayout(
    sidebarPanel(
      # This panel contains the first Data Table
      conditionalPanel(
        #input.dataset is used in DT's to show which data is actually used
        'input.dataset === "myDF"',
        #Included a checkbox for what columns should be in the data Table
        #The pre selected columns are "Date" and "Steps"
        checkboxGroupInput("show_vars", "Columns of information to show:",
                           names(myDF), selected = list("Date", "Steps"))
      ),
      # This panel contains the second Data Table
      conditionalPanel(
        #input.dataset is used in DT's to show which data is actually used
        'input.dataset === "finalDataFrame"',
        #We can change this text to anything we want user to see
        helpText("Click the column header to sort a column.")
      ),
    ),
    mainPanel(
      # Allows the user to pick between the 2 tabs containing the Data Tables
      tabsetPanel(
        #name of the tabs
        id = 'dataset',
        #First tab named "myDF" contains Data Table from first set of data
        tabPanel("myDF", DT::dataTableOutput("mytable1")),
        #Second tab named "finalDataFrame" contains DT from second set of data
        tabPanel("finalDataFrame", DT::dataTableOutput("mytable2"))
      )
    )
  )
)
```

Server of the Application

```
server <- function(input, output) {
  # choose columns to display
  myDF2 = myDF[sample(nrow(myDF), nrow(myDF), replace = FALSE), ]
  #Used to render the actual Data Table based off any changes made by the user
  #Changes based on what variables are selected and the filter feature is
  #located on the top of the Data Table
  output$mytable1 <- DT::renderDataTable({
    DT::datatable(myDF2[, input$show_vars, drop = FALSE], rownames = FALSE, filter = 'top')
  })

  #Used to render teh Data Table and does not have many features like the first
  #Data Table. This one only contains the means (finalDataFrame)
  output$mytable2 <- DT::renderDataTable({
    DT::datatable(finalDataFrame, options = list(orderClasses = TRUE), rownames = FALSE)
  })
}
```

Runs the application

```
shinyApp(ui, server)
```

8.11 Visulization 2

In this visulization, we categorizes the data into three categories: Steps and Floors Climbed, Distance, and Activity Level. Each category contains bar graphs that display the data for each respective variable per month. Both the Distance and the Activity Level categories display visualizations that separate the data into different subcategories based on the intensity of the activity.

8.11.1 Code

```
library(shiny)
library(ggplot2)
library(reshape2)
# biometric data
# 'bioFinal.csv' for biometric data from 2020 Feb. to present? - Merckteam's data
# 'DrWardData.csv' for biometric data from 2016 to 2019 April - Dr. Ward's data

#This function reads in the CSV file that contains the data, named DrWardData.csv, and stores it
myDF <- read.csv("/class/datamine/corporate/merck/Merck201920/fitbit_data/DrWardData.csv")
```

```
# The subset function selects the data from the myDF data frame that separates the mil
# The melt function then takes the columns stored in miles1 and stacks them into a sin
# The as.Date function goes into the Date column of "miles" and correctly formats the
miles1 <- subset(myDF, select = c('Date','Lightly.Active.Miles', 'Moderately.Active.Mil
miles <- melt(miles1)
miles$Date <- as.Date(miles$Date)

# The subset function selects the data from the myDF data frame that separates the min
# The melt function then takes the columns stored in minutes1 and stacks them into a s
# The as.Date function goes into the Date column of "minutes" and correctly formats th
minutes1 <- subset(myDF, select = c('Date', 'Sedentary.Minutes', 'Lightly.Active.Minut
minutes <- melt(minutes1)
minutes$Date <- as.Date(minutes$Date)

# The subset function selects the data from the myDF data frame that separates the dat
# The melt function then takes the columns stored in hr1 and stacks them into a single
# The as.Date function goes into the Date column of "hr" and correctly formats the val
hr1 <- subset(myDF, select = c('Date', 'HR.30.100.Minutes', 'HR.100.140.Minutes', 'HR.
hr <- melt(hr1)
hr$Date <- as.Date(hr$Date)

# The as.Date function goes into the Date column of "myDF" and correctly formats the v
# The first format function goes into the year column of "myDF" and formats the values
# The second format function goes into the month column of "myDF" and formats the valu
myDF$Date = as.Date(myDF$Date)
myDF$year <- format(myDF$Date,'%Y')
myDF$month <- factor(format(myDF$Date,'%B'), levels = month.name)

# Every shiny app must include a UI header beginning with fluidPage()
# fluidPage() allows the user to adjust the size of their window while preserving the
ui<-fluidPage(
  # The first section of code uses the navbarPage() function to create the menu at the

  # Creates a menu of selections titled "Fitbit Dashboard"
  navbarPage("Fitbit Dashboard",
             # tabPanel 1

             # The tabPanel() function creates the "How to Use" tab in the Fitbit Dash
             tabPanel("How to Use",
                      # Includes Merck logo
                      img(src = "https://assets.phenompeople.com/CareerConnectResource
                      # Creates main title "Biometric Data Dashboard"
                      h1("Biometric Data Dashboard"),
                      mainPanel(
                        # Includes bolded text and link
```

```
           strong("Refer to the website here for more information:"),
           a("https://www.merck.com/index.html"),
           # Smaller heading than h1 to create title "How to Use My Dashboard"
           h3("How to Use My Dashboard"),
           # Outputs HTML "text2" which is included in the server block
           htmlOutput("text2")
         )
),

# tabPanel 2

# The tabPanel() function creates the "Activity Progress" tab in the Fitbit Dashboar
tabPanel("Activity Progress",
         sidebarLayout(
           # Creates title "Activity Progress" for sidebar panel
           sidebarPanel(h3("Activity Progress"),
                        # Allows user to input date range
                        dateRangeInput("dates", ("Date range")),
                        # Allows user to select single y-axis variable for graph
                        varSelectInput("variables", ("Variable:"), myDF[c(-1, -2)]),
                        # Allows user to select variable y-axis (includes all levels
                        selectInput("stack", ("Variable (Stacked):"),
                                    c("N/A" = "nana", "Active Miles" = "miles", "Act
           ),
           # Creates the main panel
           mainPanel(
             # Includes italicized text
             em("Select a date range and a variable!"),
             # Includes bolded text
             strong("Refer to the website here for more information:"),
             # Includes link
             a("https://www.merck.com/index.html"),
             # Displays the bar graph with selected variable
             plotOutput(outputId = "bioBarGraph")
           )
         )
),

# tabPanel 3

# The tabPanel() function creates the "Step Goals" tab in the Fitbit Dashboard and t
tabPanel("Step Goals",
         # Creates sidebar panel with smaller heading "10,000-Step Goal"
         sidebarPanel(h3("10,000-Step Goal"),
                      # Allows user to input one specific date
```

```
                              dateInput("goaldate", ("Date"))
                 ),
                 # Creates the main panel
                 mainPanel(
                   # Includes bolded text
                   strong("Refer to the website here for more information:"),
                   # Includes link
                   a("https://www.merck.com/index.html"),
                   # Displays pie chart
                   plotOutput(outputId = "bioDoughnut")
                 )
        ),
        # tabPanel 4

        # The tabPanel() function creates the "Visualization 2" tab in the Fitbit
        tabPanel("Visualization 2",
                 sidebarLayout(
                   # Creates title "Activity Progress" for sidebar panel
                   sidebarPanel(h3("Visualization 2"),
                   ),
                   # Creates the main panel
                   mainPanel(
                     # Includes bolded text
                     strong(HTML("Activity graphs are listed below <br/>")),
                     strong("Steps and Floors Climbed"),
                     # Displays the bar graph that shows total steps per month
                     plotOutput(outputId = "bioStep"),
                     # Displays the bar graph that shows total floors climbed per
                     plotOutput(outputId = "bioFloors"),
                     # Includes bolded text
                     strong(HTML("Distance <br/>")),
                     # Includes italicized text
                     em("This section shows the total distance walked per month in
                     # Displays the bar graph that shows total miles walked per m
                     plotOutput(outputId = "bioTM"),
                     # Displays the bar graph that shows total number of lightly
                     plotOutput(outputId = "bioLM"),
                     # Displays the bar graph that shows total number of moderate
                     plotOutput(outputId = "bioMM"),
                     # Displays the bar graph that shows total number of very act
                     plotOutput(outputId = "bioVM"),
                     # Includes bolded text
                     strong(HTML("Activity Level <br/>")),
                     # Includes italicized text
                     em("This section shows the time spent in each level of activi
                          # Displays the bar graph that shows the amount of time spent being ligh
                          plotOutput(outputId = "bioLA"),
                          # Displays the bar graph that shows the amount of time spent being fai
                          plotOutput(outputId = "bioFA"),
                          # Displays the bar graph that shows the amount of time spent being ver
                          plotOutput(outputId = "bioVA"),
                        )
                      )
             )
  )
)


server <- function(input, output) {

  # You can access the values of the widget (as a vector of Dates)
  # with input$dates,
  # e.g. output$value <- renderPrint({ input$dates })

  # text on tabPanel 1
  output$text2 <- renderUI({
    HTML(paste("", "1. Activity Progress with Bar Charts",
               "- Date Range: select a start and an end date to set a date range of the activity
               "- Variables: use the drop-down to select a variable of the activity progress you
               "- Variables (Stacked): use the drop-down to select a variable of the activity pro
               "", "2. Step Goals with Doughnut Charts",
               "- Date: select a date of the activity progress you wish to see",
               sep="<br/>"))
  })

  # bar charts on tabPanel 2
  output$bioBarGraph <- renderPlot({

    # counter for the start date - date range
    counter1 = 0
    for (myDate in as.character(myDF$Date)){
      counter1 = counter1 + 1
      if (myDate == input$dates[1]) {
        break
      }
    }

    # counter for the end date - date range
    counter2 = 0
    for (myDate in as.character(myDF$Date)){
```

```
    counter2 = counter2 + 1
    if (myDate == input$dates[2]) {
      break
    }
  }

  # subset of myDF with the selected date range
  mySubset <- myDF[c(counter1:counter2),]
  if (input$stack == "nana"){ # single bar chart - if stacked bar is N/A
    ggplot(data=mySubset, aes(x=Date, y=!!input$variables)) + geom_bar(stat="identit
  }else if(input$stack == "miles"){ # stacked bar chart - active miles
    ggplot(data=miles, aes(x=Date, y=value, fill=variable)) + geom_bar(stat="identit
  }else if(input$stack == "minutes"){ # stacked bar chart - active minutes
    ggplot(data=minutes, aes(x=Date, y=value, fill=variable)) + geom_bar(stat="identi
  }else if(input$stack == "hr"){ # stacked bar chart - HR minutes
    ggplot(data=hr, aes(x=Date, y=value, fill=variable)) + geom_bar(stat="identity",
  }
})
# Visualization 2
# Bar graph that shows total steps per month
output$bioStep <- renderPlot({
  ggplot(data = myDF, aes(x = month, y= Steps))+
    geom_bar(stat = "identity", color = "#007a73", width = 0.5, position = position_
    ggtitle("Total Steps per Month") +
    xlab("Month") +
    ylab("Steps")
})

# Bar graph that shows total floors climbed per month
output$bioFloors <- renderPlot({
  ggplot(data = myDF, aes(x = month, y= Floors.Climbed))+
    geom_bar(stat = "identity", color = "#007a73", width = 0.5, position = position_
    ggtitle("Total Floors Climbed per Month") +
    xlab("Month") +
    ylab("Floors Climbed")
})

# Bar graph that shows total miles walked per month
output$bioTM <- renderPlot({
  ggplot(data = myDF, aes(x = month, y= Total.Miles))+
    geom_bar(stat = "identity", color = "#007a73", width = 0.5, position = position_
    ggtitle("Total Miles Walked per Month") +
    xlab("Month") +
    ylab("Miles")
})
```

```
# Bar graph that shows lightly active miles walked per month
output$bioLM <- renderPlot({
  ggplot(data = myDF, aes(x = month, y= Lightly.Active.Miles))+
    geom_bar(stat = "identity", color = "#007a73", width = 0.5, position = position_dodge(width
    ggtitle("Lightly Active Miles Walked per Month") +
    xlab("Month") +
    ylab("Miles")
})

# Bar graph that shows moderately active miles walked per month
output$bioMM <- renderPlot({
  ggplot(data = myDF, aes(x = month, y= Moderately.Active.Miles))+
    geom_bar(stat = "identity", color = "#007a73", width = 0.5, position = position_dodge(width
    ggtitle("Moderately Active Miles Walked per Month") +
    xlab("Month") +
    ylab("Miles")
})

# Bar graph that shows very active miles walked per month
output$bioVM <- renderPlot({
  ggplot(data = myDF, aes(x = month, y= Very.Active.Miles))+
    geom_bar(stat = "identity", color = "#007a73", width = 0.5, position = position_dodge(width
    ggtitle("Very Active Miles Walked per Month") +
    xlab("Month") +
    ylab("Miles")
})

# Bar graph that shows the amount of time spent being lightly active per month
output$bioLA <- renderPlot({
  ggplot(data = myDF, aes(x = month, y= Lightly.Active.Minutes))+
    geom_bar(stat = "identity", color = "#007a73", width = 0.5, position = position_dodge(width
    ggtitle("Time Spent Being Lightly Active per Month") +
    xlab("Month") +
    ylab("Minutes")
})

# Bar graph that shows the amount of time spent being fairly active per month
output$bioFA <- renderPlot({
  ggplot(data = myDF, aes(x = month, y= Fairly.Active.Minutes))+
    geom_bar(stat = "identity", color = "#007a73", width = 0.5, position = position_dodge(width
    ggtitle("Time Spent Being Fairly Active per Month") +
    xlab("Month") +
    ylab("Minutes")
})
```

```
  # Bar graph that shows the amount of time spent being very active per month
  output$bioVA <- renderPlot({
    ggplot(data = myDF, aes(x = month, y= Very.Active.Minutes))+
      geom_bar(stat = "identity", color = "#007a73", width = 0.5, position = position_
      ggtitle("Time Spent Being Very Active per Month") +
      xlab("Month") +
      ylab("Minutes")
  })


  # doughnut chart on tabPanel 3
  output$bioDoughnut <- renderPlot({

    # counter for the date - daily goals
    counter3 = 0
    for (myDate in as.character(myDF$Date)){
      counter3 = counter3 + 1
      if (myDate == input$goaldate) {
        break
      }
    }

    # doughnut bar
    # daily steps of the selected date
    steps <- myDF[c(counter3:counter3),c("Steps")]
    # get a percentage - daily steps out of 10000 steps
    data <- data.frame(
      category=c("Steps","N/A"),
      count=c(steps, 10000-steps)
    )
    data$fraction = data$count / 100
    data$ymax = cumsum(data$fraction)
    data$ymin = c(0, head(data$ymax, n=-1))
    ggplot(data, aes(ymax=ymax, ymin=ymin, xmax=4, xmin=3, fill=category)) + geom_rect
  })
}
shinyApp(ui, server)
```

8.12 Critical decisions

Our group decided to use RStudio to test pieces of R code on our own local machines, but we ran into the problem of not being able to collaborate with our team in an efficient way, so we looked for an alternative method where we could all have access to update a file. We decided to create a repository on bitbucket where we all had access to the files and could update everything using git.

After setting up the colloboration, we started working to improve the previous dashboard and implementing new feature. To improve our skills in R shiny dashboard, we each implemented a different form of data visualization, including a histogram, a scatterplot, a box plot, and a pie chart, and implemented them using data from the previous year. Additionally, we looked at code from the previous year's team and tried to run their app and resolve any issues that occurred.

While implementing the visulization, we faced few difficuluties. For instance, in the viuslization 1, we have to modify the whole visualization as our initial visulization was having issue to modify itself during the user interaction. Similarly, second visulization was interactive initially and allowed the user to choose which variable they wanted to see on the graph. However, an issue we faced with this was that when the user selected a variable, the values on the y-axis would become N/A and the bars on the graph would all be the same height. After discussing several potential solutions to this issue, we decided that the best option would be to create several static graphs, allowing the user to scroll to the desired graph.

8.13 Technical Report

During the last sprint, we focused mainly on the visualization and refinement stages of our development process. In the visualization stage, we brainstormed, discussed, and developed the code for the two visualizations we created. In the refinement stage, we reviewed the issues in the code and discussed potential solutions. We then cycled back to the visualization stage in order to implement the solutions we discussed.

Chapter 9

Back End Team

9.1 Creation of Dummy Data

9.1.1 Why do we need dummy data?

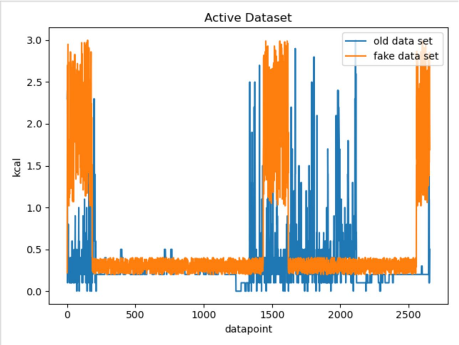
In order to effectively test the functionality of the backend code, a lot of pre-existing data is needed to fill the database to simulate a normal functioning environment for the app. Initially, we only had about 40 users' data, and it was not enough. Therefore, we decided to create fake user data based on pre-existing user data (dummy data).

9.1.2 How we generated dummy data?

In order to let dummy data make sense and follow pre-existing data's patterns, we studied the patterns of each pre-existing data category. There are active, basal, dist, elev, heart, and steps data. We mostly used user 185 and 186's dataset as the reference while creating dummy dataset.

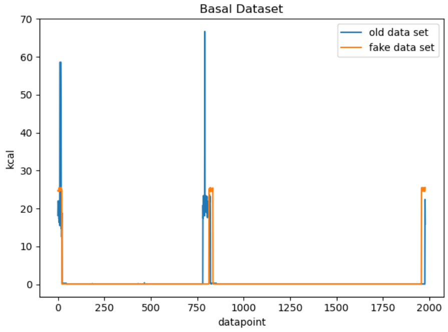
For the active data, we first calculated the mean value of the original data. Then we found out the number of data points above the mean. Because it can be observed that within each active data set, data points above the mean value cluster into three group, and one of them always starts at the beginning. Therefore, we also chose the start as the position of the first high value data cluster, and randomly chose position for the other two. The value of each data point was chosen based on the observed range of the reference data point.

The result of fake data vs. real data graph can be found below:

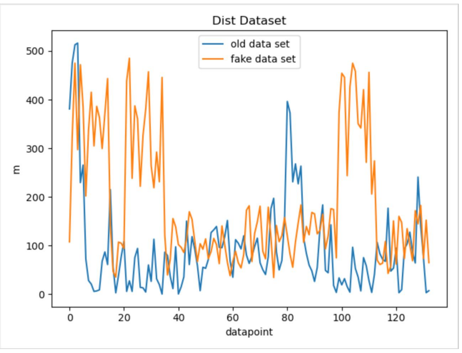


For other data categories, we utilized similar approaches while ajusting parameters and the possible range of random number generation based on each data categories' own characteristics. Below are the fake data vs. real data comparison graphs for other data categories.

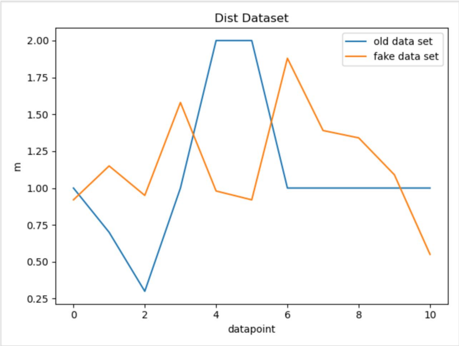
For basal data:



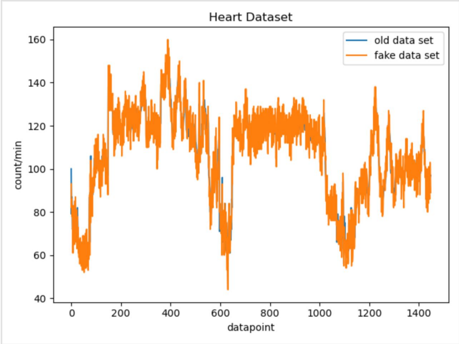
For Dist data:



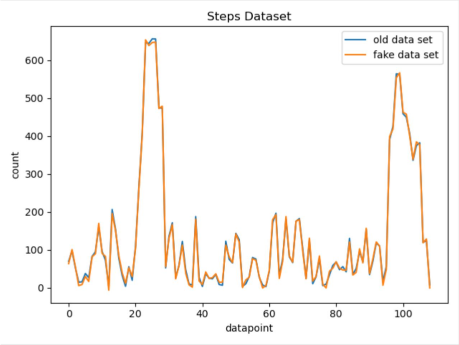
For Elev data:



For Heart data:



For Steps data:



9.1.3 How to run the scripts?

In order to run the scripts, some modifications to the GenerateFakeData.py are needed. There are three parameters that need modification: folder_name_list, real_name_list, and path. “folder_name_list” tells the name of the folders that will contain the dummy data. “real_name_list” tells the name of the reference files being used to generate dummy data. “path” is the current directory. Once these three parameters are set up correctly, just directly run the GenerateFakeData.py script with no input.

9.1.4 Potential improvements

Because our dummy data generation script is specifically modified for each data set, if in the future a new type of data appears, we need to rewrite our code in order to accommodate the new data type. Therefore, we can improve our code by applying more statictical knowledge so that the code can be applied to generate dummy data from more different data types.

9.2 Creating a connection between API Endpoints

9.2.1 List of API end-points:

1. Front-End Patient User input to the database
2. Biometric Data from the FitBit API to the database
3. Data from the database to the RShiny Dashboard for visualization

9.2.2 What framework we are going to use?

We are currently looking to use the framework Express.js to write the routes between the database and various API endpoints.

Express is a fast, light-weight web framework for Node.js. Express is a pretty good framework. It's the most popular node application framework out there. Express is widely used as middleware in node apps as it helps to organize your app into a MVC architecture. It's the E of the popular MEAN stack. Express manages following things very easily:

- Routing
- Sessions
- HTTP requests
- Error handling

At times writing code from scratch for above things can be time consuming. But by using express it's only a matter of few methods. Express also helps in organizing your code.

Link to set up Node

9.2.3 Setting up SQL database connection:

We created a connection with the MariaDB database in our Node Rest API server to be able to send and receive data. The following link shows a tutorial on how we set up the connection.

Link to setup SQL connection in Node

9.2.4 Design of the Routes

There are five kinds of routes:

GET: The GET method requests a representation of the specified resource. Requests using GET should only retrieve data and should have no other effect.

POST: The POST method requests that the server accept the entity enclosed in the request as a new subordinate of the web resource identified by the URI. The data POSTed might be, for example, an annotation for existing resources; a message for a bulletin board, newsgroup, mailing list, or comment thread; a block of data that is the result of submitting a web form to a data-handling process; or an item to add to a database.

PUT: The PUT method requests that the enclosed entity be stored under the supplied URI. If the URI refers to an already existing resource, it is modified; if the URI does not point to an existing resource, then the server can create the resource with that URI.

DELETE: The DELETE method deletes the specified resource.

PATCH: The PATCH method applies partial modifications to a resource

9.3 Hosting the database on a Server

9.3.1 Issues faced with hosting a database on Scholar:

1. Scholar is a small computer cluster, suitable for classroom learning about high performance computing (HPC).
2. It can be accessed as a typical cluster, with a job scheduler distributing batch jobs onto its worker nodes, or as an interactive resource, with software packages available through a desktop-like environment on its login servers.
3. We tried to create a connection to the sql server hosted on this server from our local but we faced issues because there was a firewall preventing access to the database from a foreign server
4. We tried running our Backend API on scholar but we were unable to install NodeJS, MySQLWorkbench and other packages on the server without authorization.

9.3.2 Work around

1. In order to install packages on scholar, we are expected to make requests to the administration with a list of all program lines to run in the form of

a SLURM job.

2. The Simple Linux Utility for Resource Management (SLURM) is a system providing job scheduling and job management on compute clusters. With SLURM, a user requests resources and submits a job to a queue. The system will then take jobs from queues, allocate the necessary nodes, and execute them.

3. To submit work to a SLURM queue, you must first create a job submission file.

More info can be found at the following link: SLURM Job

9.3.3 How we proceeded

Since we have to make a request each time we have to install a package, we decided to make just one request with a complete list of all installation. As a result, we wanted to host a temporary database on AWS that we can connect to and test on using our local machine.

We created a copy of the entire database on AWS with the following credentials:

- Hostname: rnr56s6e2uk326pj.cbetxkdyhwsb.us-east-1.rds.amazonaws.com
- Username: cb3i17t0aqn6a4ff
- Password: e2l4k9zn24shcj42

9.4 Adding data to the database

9.4.1 Adding a CSV to the database

1. The data engineering team had made a script that creates a CSV file with all the users FitBit data when they make a request to the API
2. We decided to make a python script to load this csv data onto the database we hosted on AWS.

9.4.2 POST information

1. The JSON to add a datapoint to the database is as follows:

```
obj = {'collection_date': row["Date"],
       'steps': row["Steps"],
       'floors_climbed': row["Floors Climbed"],
       'total_miles': row["Total Miles"],
       'lightly_active_miles': ["Lightly Active Miles"],
       'moderately_active_miles': row["Moderately Active Miles"],
       'very_active_miles': row["Very Active Miles"],
       'sedentary_minutes': row["Sedentary Minutes"],
       'lightly_active_minutes': row["Lightly Active Minutes"],
       'fairly_active_minutes': row["Fairly Active Minutes"],
       'very_active_minutes': row["Very Active Minutes"],
       'hr30_100_minutes': row["HR 30-100 Minutes"],
       'hr100_140_minutes': row["HR 100-140 Minutes"],
       'hr140_170_minutes': row["HR 140-170 Minutes"],
       'hr170_220_minutes': row["HR 170-220 Minutes"],
       'average_resting_heartrate': row["Average Resting HR"],
       'bmi': row["BMI"],
       'sleep_efficiency': row["Sleep Efficiency"],
       'weight': row["Weight"],
       'minutes_asleep': row["Minutes Alseep"],
       'fbusername': row["username"]
       }
```

2. We made structures in this form by reading the CSV information into a pandas dataframe and made a post request to our API

9.4.3 Patient information and Study Specific Data

The patient information is sent to the database form a webpage which will be used by Merck Scientists to load the patients they are studying during a clinical trial.

The JSON for a patient is as follows:

```
{
  patient_id,
  fbusername,
  first_name,
  last_name,
  gender,
  date_of_birth,
  height
}
```

The JSON for the study data is as follows:

```
{
    patient_id,
    input_date,
    family_history_cancer,
    family_history_heart_disease,
    diagnostic_notes
}
```

9.5 Firebase and Login System for the Phone App

(This part is also in frontend team's documentation, because it is relavant to both of the teams) Because our app's users are patients and their information is very confidential, It is important to implement our login and authentication system securely. Instead of building a secure login system from scratch by ourselves, we chose to use a pre-existing login authentication solution–firebase. In firebase, users' passwords are first encrypted and then stored. Even if hackers successfully gain access to firebase's database, they still would not know the passwords because everything stored there is the encrypted version.

The next step is to incorporate the firebase login system into our own database. Our current thought is that, once the user logs in through firebase, they will receive a key from firebase. Then whenever they want to modify or store data in our database, they send the key along with every HTTP request. The backend would check the correctness of the key and only allow accessibility if the key is correct.

Chapter 10

Front End Team

10.1 Utilization of Third Party Software

The sole third party software being used in our application is Firebase, a Google based software designed for user authentication purposes. The codes upon which this app is built are independently developed by our team members. The data used to develop the app is Biometric data collected currently only through Fitbit, though the goal is to expand the collection capability to be able to obtain Apple Watch data as well. This data is accessed through an external Merck API which is linked through an SCTP request.

10.2 Software Components

Our application has five main sections. The first section covers everything prior to successful user login. This includes the login page along with Firebase authentication. The second section is the home page where users would be able to view displayed data pertaining to themselves. The third section is a graphics page where graphs are presented, informing users their health progress and their default Fitbit data. The fourth section is the question response page where users are guided to complete questionnaires for the purpose of data collection beyond the default data collected by the wearable device. The final section is the user setting page where users can view and edit their information and settings; a log out option is also available through this section.

10.2.1 Pre-Login section

The pre-login section has been mostly completed, however we are yet to integrate the firebase completely within the page itself. The firebase portion of this page has been created. This portion does not contain any important functions besides the absolute base, standard functions required by firebase to run. The login page aspect of this section has a few important functions. For one, the entire app is stacked into a function called Loginpage to make it harder for consumers to access. The page runs similar to as expected. There's a space to enter a username and password and a login button which gives a notification that you are being logged in. TextInput and TouchableOpacity were both enabled to allow popup notifications and username and password entry. For the password entry, secureTextEntry was set to true, which turns the letters into dots as they are typed.

10.2.2 Home Page Section

The home page section is very simple; the current home page is where basic data such as sleep, heart rate, etc would be presented in a clean fashion as a summary. It is our goal to implement additional features to this page along with an in depth description of user's data through the graphics page.

10.2.3 Question Response Section

The question response page collects data to further bolster the health data collected by the users' wearables. It has many implemented functionality to facilitate the data collection process, for example, it displays information based on users' previous responses; This is done using the constructor function. It also uses the onChangeText function for the sake of clarity when the user selects a value. Ideally, a camera feature will be made available on this page in the event that a user wishes to upload an image to further elaborate on their health conditions.

10.2.4 User Setting Section

Finally, There is the user information page which contains all of the basic testing information and personal information that clinical trial patient would preferably want to know, users can also log out through this page if they so wishes. Ideally, this page would also bare a camera feature which allow users to set a profile picture. Codes are stored in a function creatively named Code, and consists of basic text and image functions with one TouchableOpacity function so far to log the user out.