Process Mining Event Log Preprocessor

By: Milan Patel

UIC GPIP Internship

08/21/2020

Table of Contents

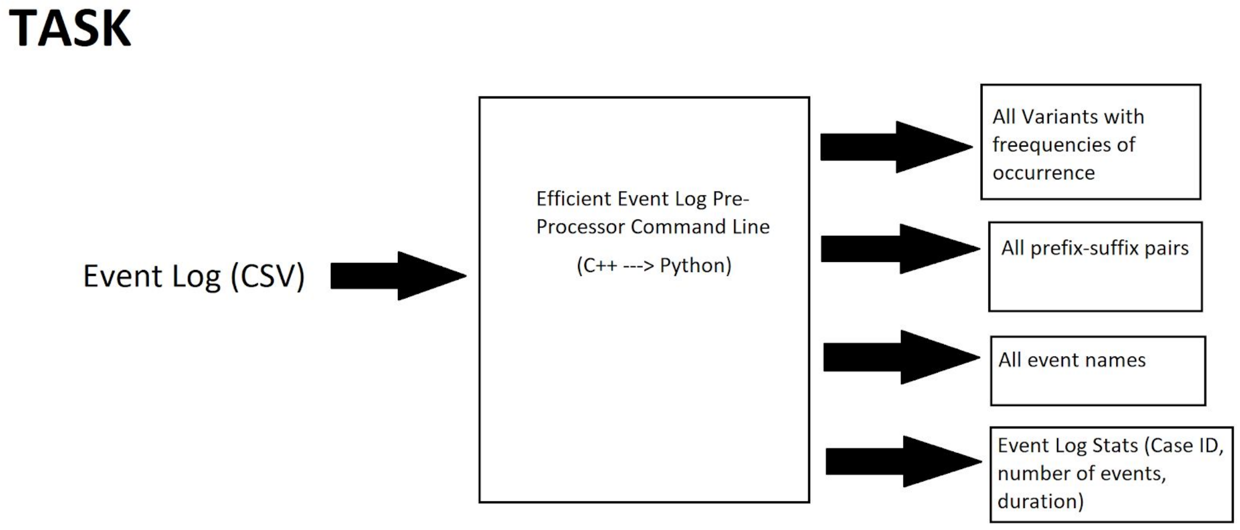
1. Introduction: Page 2
2. Objective: Page 3
3. Development of Project C++
    a. Week 1: Page 4
    b. Week 2: Page 7
    c. Week 3: Page 8
4. Development of Project Python
    a. Week 4: Page 11
    b. Week 5: Page 13
5. Conclusion: Page 16

Introduction

The GPIP internship which I took part in over the summer of 2020 was focused on Process Mining. In the beginning, I was clueless about what process mining was. I quickly learned that it was a newer field of computer science, but had a lot of interesting aspects. Process mining is composed of gathering data and information and provides details about how key processes are being performed. One of the earliest examples that were brought to my attention regarding processing mining was the events that take place at a hospital. Every day at a specific hospital, there are multiple patients which enter the facility. Each of those patients (case) represents a process case with various events, such as checking in, surgery, discharge, and paying the bill in regards to a hospital. A trace is defined as the sequence of events including all data that is associated with the events. For example, the event "check-in", the event would contain data regarding time, cost, and various other metadata. The combination of all these patients would create a large event log typically in XES or CSV format. Additional terms that I learned are variants and prefix-suffix pairs. A variant is a sequence of events from end to end, so if patient A did actions, a, b, and c the variant for that would be ABC. In a large event log, there would be multiple occurrences of this variant along with others. For the example above, the 1st prefix would be "a" and the 1st suffix would be "bc". The nth prefix of a variant is the sequence of event names from the first to the nth event of a particular variant. The nth suffix of a variant is the sequence of event names from the (n+1)th to the last event of a particular variant.

Objective

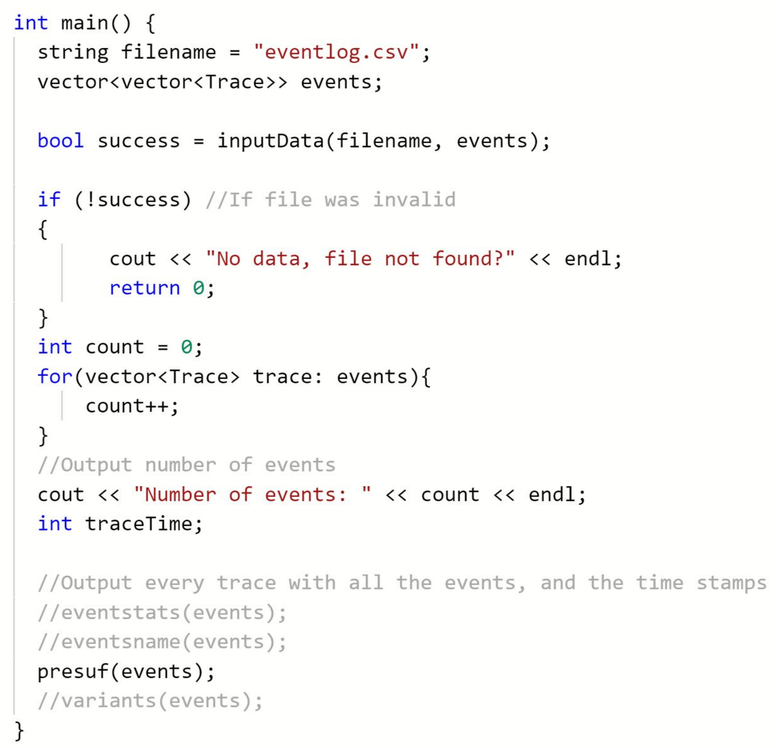
Reading and understanding all this data in an event log would be very complicated for a normal human with large chunks of data which is where my task came into play. The problem that I was assigned to tackle was to create an efficient event log preprocessor. Going into this internship I was unfamiliar with Python so I decided to create this project in C++ for three weeks while learning Python concurrently, then convert the project to Python, and take the last week to document and debug my code. The flowchart below details some of the tasks and how I expected to go about this project.


TASK
Event Log (CSV)
Efficient Event Log Pre-Processor Command Line
(C++ ---> Python)
All Variants with freequencies of occurrence
All prefix-suffix pairs
All event names
Event Log Stats (Case ID, number of events, duration)

I intended to read in a CSV file and using my code parse various details in an easy to read JSON output format. I would output all variants with frequencies of occurrence, all the prefix-suffix pairs found in the event log, all the unique events name, and various metadata stats such as how many events there were, how many events there were per trace, and how long the trace took place for. I had some familiarity with parsing CSV files and using efficient data structure methods to gather and store the data in C++ so I began to plan out the project to fit in the six-week timeline that I had.

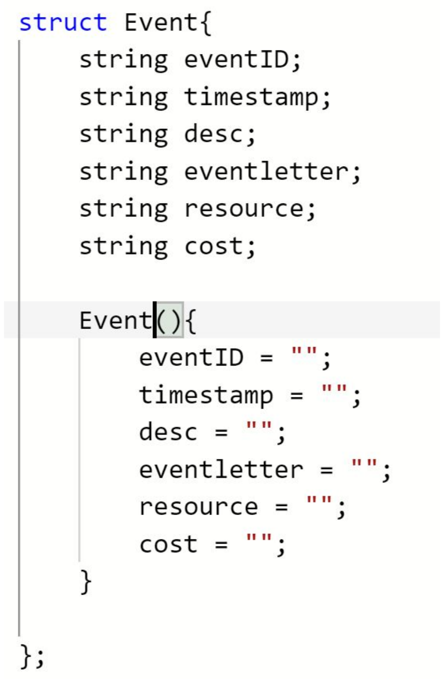
Development - Week 1

In the first week of the internship, I started the base C++ code with the event log being parsed properly. I thought of various methods of how I would store the events, and metadata such as a timestamp. I began experimenting with an example CSV file which was similar to the hospital example given in the introduction. The code below is the main function that I programmed.



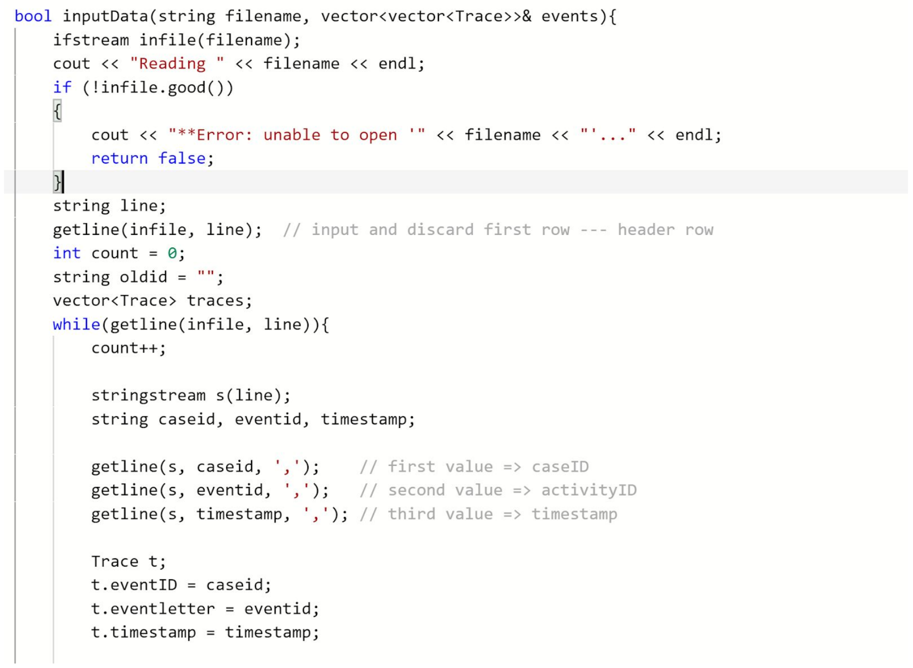
```cpp
int main() {
  string filename = "eventlog.csv";
  vector<vector<Trace>> events;

  bool success = inputData(filename, events);

  if (!success) //If file was invalid
  {
        cout << "No data, file not found?" << endl;
        return 0;
  }
  int count = 0;
  for(vector<Trace> trace: events){
    count++;
  }
  //Output number of events
  cout << "Number of events: " << count << endl;
  int traceTime;

  //Output every trace with all the events, and the time stamps
  //eventstats(events);
  //eventsname(events);
  presuf(events);
  //variants(events);
}
```

I decided to use a vector of vectors, allowing for expandability and the object stored inside the second vector was a structure named Event. The code below outlines the details of the Event structure.



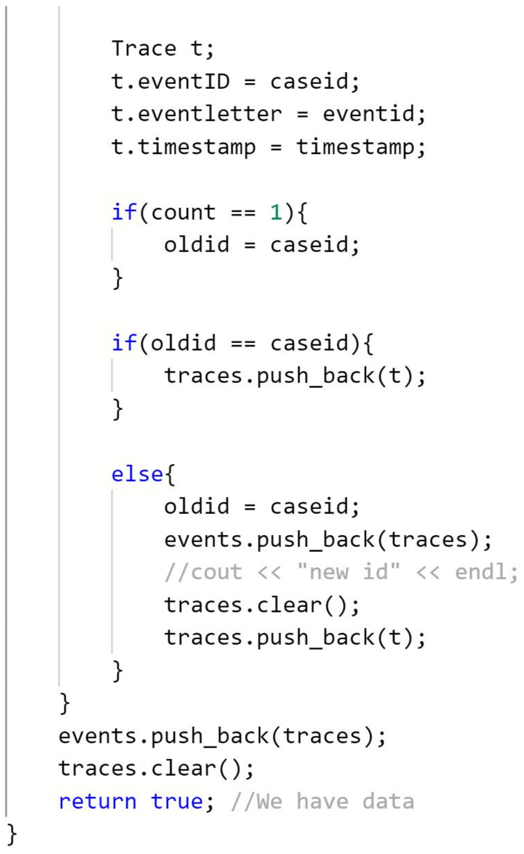
```
struct Event{
    string eventID;
    string timestamp;
    string desc;
    string eventletter;
    string resource;
    string cost;

    Event(){
        eventID = "";
        timestamp = "";
        desc = "";
        eventletter = "";
        resource = "";
        cost = "";
    }

};
```

The struct served the purpose of storing all caseID, timestamp, description, cost, and various metadata that would be contained in the event log CSV file. As seen in the main function, a separate function called inputData was called to properly gather all the information in the file into the vector, which was passed as a reference to the main function to store the information properly.



```
bool inputData(string filename, vector<vector<Trace>>& events){
    ifstream infile(filename);
    cout << "Reading " << filename << endl;
    if (!infile.good())
    {
        cout << "**Error: unable to open '" << filename << "'..." << endl;
        return false;
    }
    string line;
    getline(infile, line);  // input and discard first row --- header row
    int count = 0;
    string oldid = "";
    vector<Trace> traces;
    while(getline(infile, line)){
        count++;

        stringstream s(line);
        string caseid, eventid, timestamp;

        getline(s, caseid, ',');    // first value => caseID
        getline(s, eventid, ',');   // second value => activityID
        getline(s, timestamp, ','); // third value => timestamp

        Trace t;
        t.eventID = caseid;
        t.eventletter = eventid;
        t.timestamp = timestamp;
```

The function above details if the file is opened properly, if the name is not provided correctly, the function would be exited. Using a loop to go through and commas to split the data, the proper information was stored into a new instance of the Event structure.

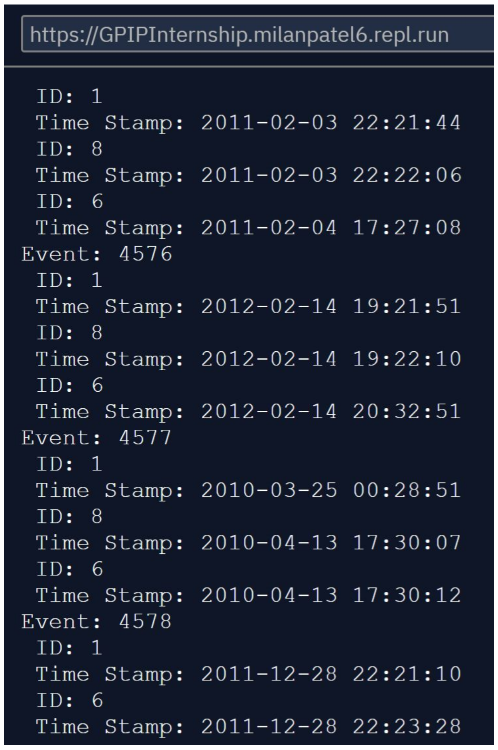


```
        Trace t;
        t.eventID = caseid;
        t.eventletter = eventid;
        t.timestamp = timestamp;

        if(count == 1){
            oldid = caseid;
        }

        if(oldid == caseid){
            traces.push_back(t);
        }

        else{
            oldid = caseid;
            events.push_back(traces);
            //cout << "new id" << endl;
            traces.clear();
            traces.push_back(t);
        }
    }
    events.push_back(traces);
    traces.clear();
    return true; //We have data
}
```

I used a temporary variable to keep track of the caseID. If the number of cases counted was one, the oldID variable would be set as caseID, and the trace would be pushed back. Once an event occurred with a new case, then the oldID would have to be updated, the traces vector would have to be added to the events vector, and the traces vector would have to be reset for the new trace. This allowed me to keep track of all the separate events and individual cases for each event.

Development - Week 2

For the second week, I was able to gather proper output and display everything correctly as shown in the output below.



```
https://GPIPInternship.milanpatel6.repl.run
```

```
 ID: 1
 Time Stamp: 2011-02-03 22:21:44
 ID: 8
 Time Stamp: 2011-02-03 22:22:06
 ID: 6
 Time Stamp: 2011-02-04 17:27:08
Event: 4576
 ID: 1
 Time Stamp: 2012-02-14 19:21:51
 ID: 8
 Time Stamp: 2012-02-14 19:22:10
 ID: 6
 Time Stamp: 2012-02-14 20:32:51
Event: 4577
 ID: 1
 Time Stamp: 2010-03-25 00:28:51
 ID: 8
 Time Stamp: 2010-04-13 17:30:07
 ID: 6
 Time Stamp: 2010-04-13 17:30:12
Event: 4578
 ID: 1
 Time Stamp: 2011-12-28 22:21:10
 ID: 6
 Time Stamp: 2011-12-28 22:23:28
```

A nested for loop algorithm allowed me to go through the vector of cases, and for each case go through each event that contained trace information and display both the ID and timestamp associated with the event. The use of an efficient data structure allowed for this process to be quick. During the second week, I also took some time to research various Python syntax which I was previously not familiar with.

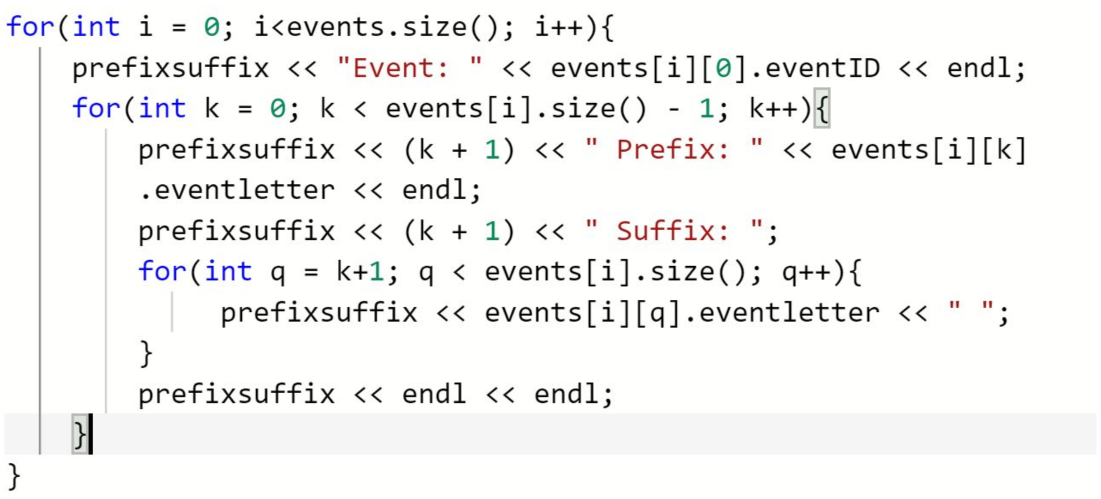
```
void eventstats(vector<vector<Trace>>& events){
  ofstream eventstats;
  eventstats.open("eventstats.txt");
  for(int k = 0; k < events.size(); k++){
    eventstats << "Event: " << events[k][0].eventID << endl;
    for(int i = 0; i < events[k].size(); i++){
        eventstats << " ID: " << events[k][i].eventletter << endl;
        eventstats << " Time Stamp: " << events[k][i].timestamp << endl;
    }
  }
  eventstats.close();
}
```

Development - Week 3

During the third week, I developed the function to display all variants of the event log, the prefix-suffix function to output all proper prefix-suffix pairs in the event log, and the unique event names function. The code below is the variant function.

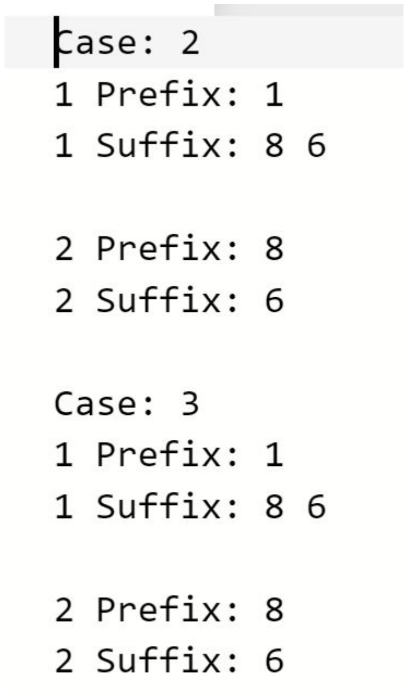
```
void variants(vector<vector<Trace>>& events){
    map<string, int> variants;
    ofstream vartxt;
    vartxt.open("variants.txt");
    for(int i = 0; i < events.size(); i++){
        string var = "";
        for(int k = 0; k < events[i].size(); k++){
            var = var + events[i][k].eventletter;
        }
        if(variants.size() == 0){
            variants.emplace(var, 1);
        }
        else{
            if(variants.count(var) == 0){
                variants.emplace(var, 1);
            }
            else{
                variants[var] = variants[var] + 1;
            }
        }
    }
```

To determine the frequency of each variant properly I decided to use a map data structure. I went through every event, and for each event, I added a loop going through all the caseIDs to determine the end to end string for the specific variant. If the map had nothing in it, I would add that string variable to the map with frequency 1. If the variant was found in the map, then the map would be edited with that key to have a value of 1 more than the previous to keep track of frequency. The second function, the prefix-suffix function was a bit more tricky as I had to go through every pair in each trace. The code for the C++ for the function did have a bug which will be addressed later.



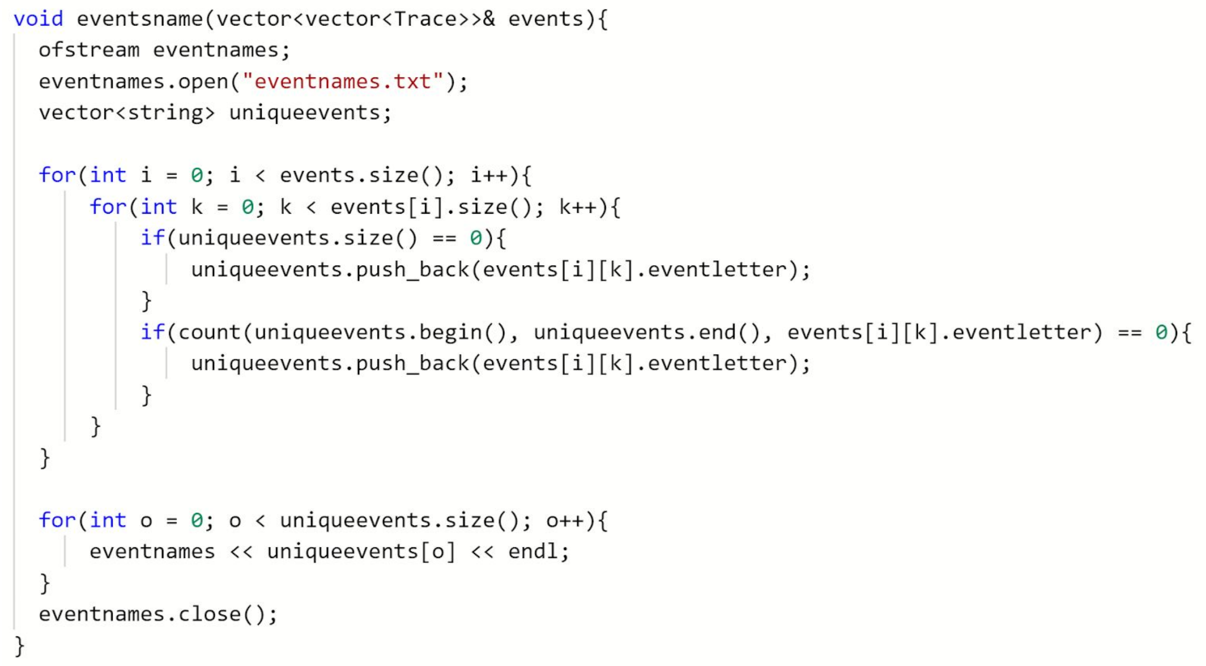
```
for(int i = 0; i<events.size(); i++){
    prefixsuffix << "Event: " << events[i][0].eventID << endl;
    for(int k = 0; k < events[i].size() - 1; k++){
        prefixsuffix << (k + 1) << " Prefix: " << events[i][k]
        .eventletter << endl;
        prefixsuffix << (k + 1) << " Suffix: ";
        for(int q = k+1; q < events[i].size(); q++){
            prefixsuffix << events[i][q].eventletter << " ";
        }
        prefixsuffix << endl << endl;
    }
}
```

I went through every event and for each event, I was able to gather the first case ID to determine that that was the 1st prefix-suffix pair. And after I accessed that I used a third nested loop to output the rest of the suffix and merge the string without starting a new line. The output for this is as shown below.



```
Case: 2
1 Prefix: 1
1 Suffix: 8 6

2 Prefix: 8
2 Suffix: 6

Case: 3
1 Prefix: 1
1 Suffix: 8 6

2 Prefix: 8
2 Suffix: 6
```

The issue with this is for case 2, the 1st prefix and 1st suffix is correct, however, for the 2nd prefix, it should be 1 8 with the suffix being 6. This error was addressed later in the Python version of the project. The last function was the unique events function. The code for that function is displayed below.

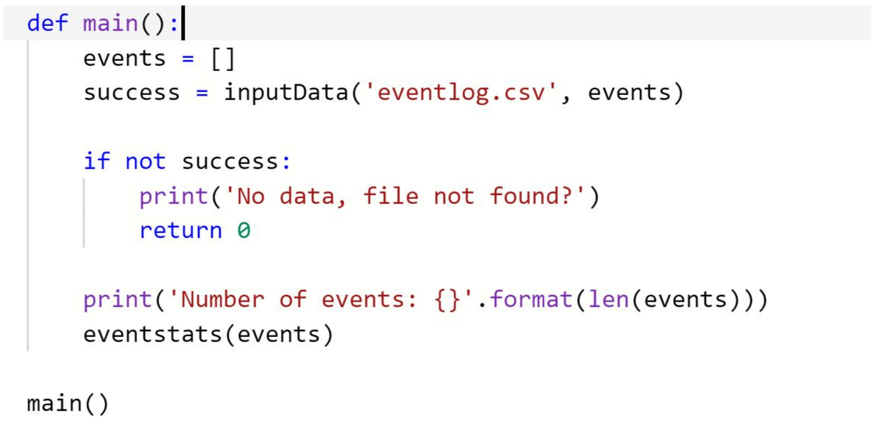


```
void eventsname(vector<vector<Trace>>& events){
  ofstream eventnames;
  eventnames.open("eventnames.txt");
  vector<string> uniqueevents;

  for(int i = 0; i < events.size(); i++){
      for(int k = 0; k < events[i].size(); k++){
          if(uniqueevents.size() == 0){
              uniqueevents.push_back(events[i][k].eventletter);
          }
          if(count(uniqueevents.begin(), uniqueevents.end(), events[i][k].eventletter) == 0){
              uniqueevents.push_back(events[i][k].eventletter);
          }
      }
  }

  for(int o = 0; o < uniqueevents.size(); o++){
      eventnames << uniqueevents[o] << endl;
  }
  eventnames.close();
}
```

I used a vector of strings, to determine all the unique events' names. I went through all the events, using a nested for loop. If unique events were empty and this was the first event, I would add it to the unique events vector. If the same event was found in the vector then it would not be added, however, if it was not found, then the event was added to the vector. In the end, loop through all the unique events and output it properly to the text file. By doing this, a hospital event log, for example, would be able to see all the different unique events, such as check-in, check out, registration, and discharge.

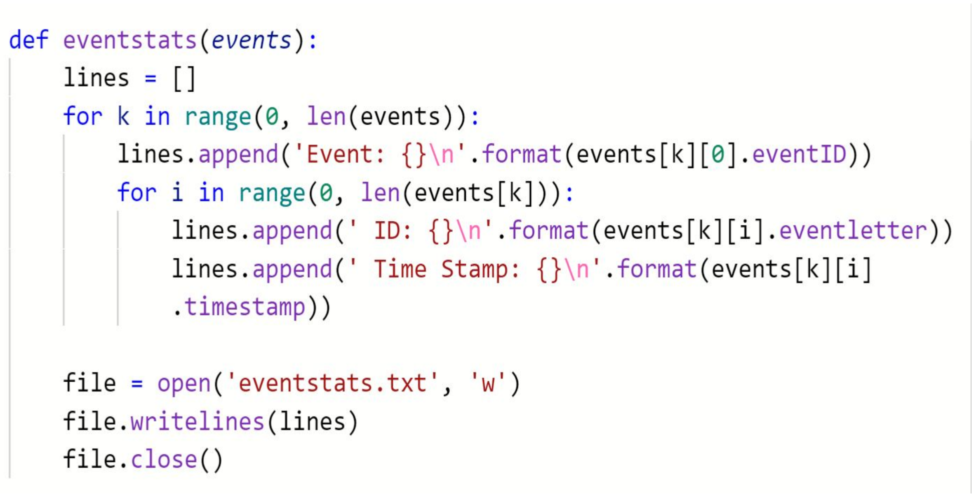
Development - Week 4

For the fourth week, I began to port over the C++ project to Python. Before this, I was unfamiliar with Python, and Python syntax. I did a lot of research on how I would accomplish some of the tasks such as using a map, or a vector. I quickly learned about lists that took care of most of my problems.



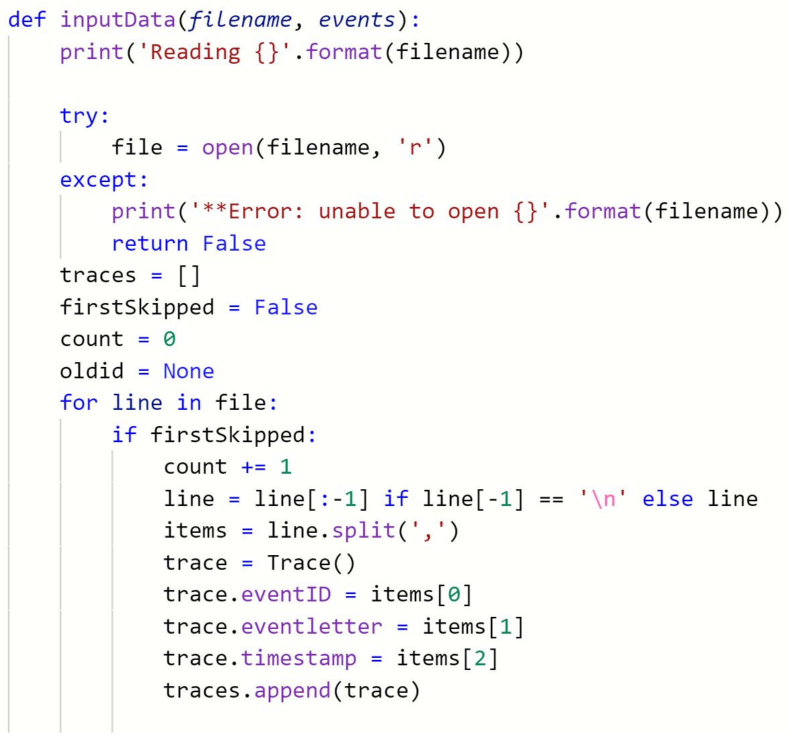
```python
def main():
    events = []
    success = inputData('eventlog.csv', events)

    if not success:
        print('No data, file not found?')
        return 0

    print('Number of events: {}'.format(len(events)))
    eventstats(events)

main()
```

The list of events was used as the parameter for the remaining functions, allowing for easy access and modification. The code regarding inputting the data from the CSV was similar however a class was used rather than a structure for the trace information.



```python
def eventstats(events):
    lines = []
    for k in range(0, len(events)):
        lines.append('Event: {}\n'.format(events[k][0].eventID))
        for i in range(0, len(events[k])):
            lines.append(' ID: {}\n'.format(events[k][i].eventletter))
            lines.append(' Time Stamp: {}\n'.format(events[k][i]
            .timestamp))

    file = open('eventstats.txt', 'w')
    file.writelines(lines)
    file.close()
```

The event stats function used the same logic going through every event, and adding that information with the proper format to the lines list. A file is then opened to output all the information from the lines.

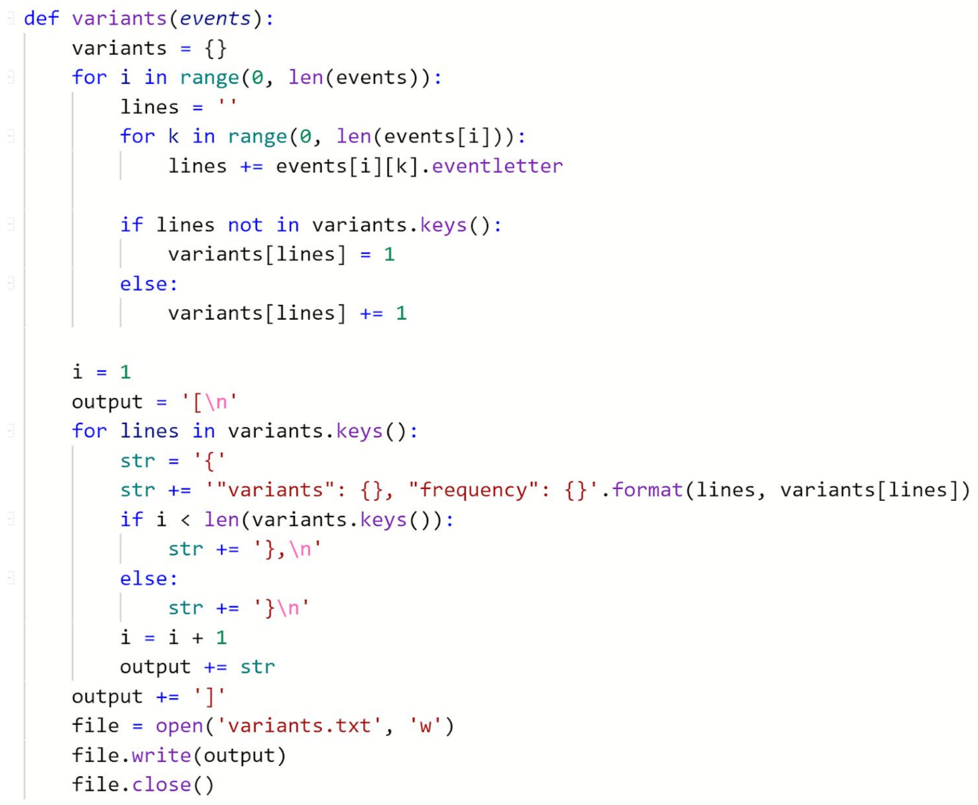


```python
def inputData(filename, events):
    print('Reading {}'.format(filename))

    try:
        file = open(filename, 'r')
    except:
        print('**Error: unable to open {}'.format(filename))
        return False
    traces = []
    firstSkipped = False
    count = 0
    oldid = None
    for line in file:
        if firstSkipped:
            count += 1
            line = line[:-1] if line[-1] == '\n' else line
            items = line.split(',')
            trace = Trace()
            trace.eventID = items[0]
            trace.eventletter = items[1]
            trace.timestamp = items[2]
            traces.append(trace)

```

The input data function used the filename as a parameter and the events list. The first line would be skipped, and a new instance of the event class would be created. The trace would be added to traces, and each trace would be filled with the proper events, creating an indirect 2D array for easy access for events, event letters, and time stamps.

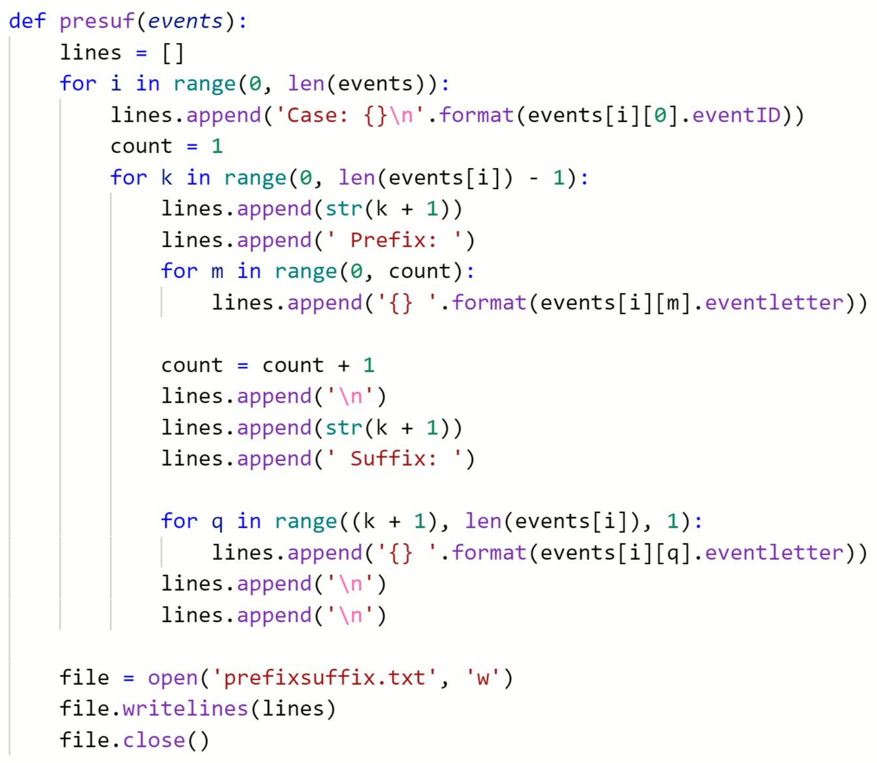
Development - Week 5

For the fifth week, I worked on completing the Python version of the project. I completed the variants, event names, and prefix-suffix functions. For the function for the variants, I had to research dictionaries in python to create an efficient for key-value pairs, for frequencies relating to a specific variant.



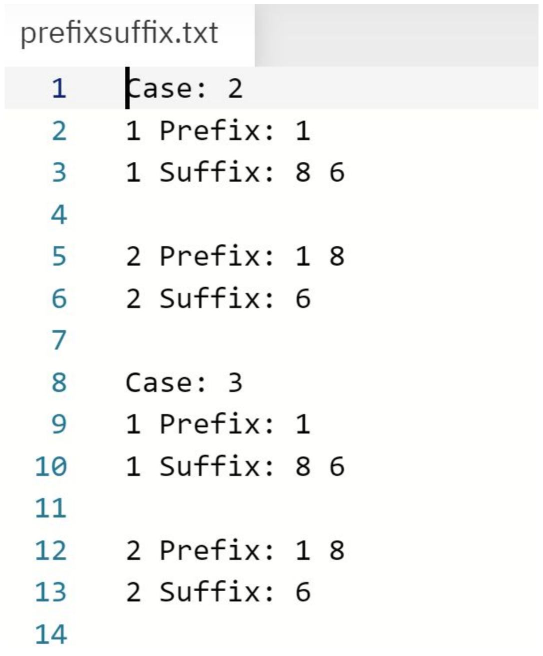
```python
def variants(events):
    variants = {}
    for i in range(0, len(events)):
        lines = ''
        for k in range(0, len(events[i])):
            lines += events[i][k].eventletter

        if lines not in variants.keys():
            variants[lines] = 1
        else:
            variants[lines] += 1

    i = 1
    output = '[\n'
    for lines in variants.keys():
        str = '{'
        str += '"variants": {}, "frequency": {}'.format(lines, variants[lines])
        if i < len(variants.keys()):
            str += '},\n'
        else:
            str += '}\n'
        i = i + 1
        output += str
    output += ']'
    file = open('variants.txt', 'w')
    file.write(output)
    file.close()
```

For every trace, I used a string variable to store the end to end variant. If that variant was not found in the dictionary then the “frequency” value would be updated to be one, however, if the variant was found, then the value would be adjusted to be one more. For more effective output I wrote the dictionary in JSON format using a string that would be added to the output variable, which would finally be written to the file. For the prefix-suffix function, I had to address the problem from earlier regarding the pairs not being output properly. I used the same concept of a lines list, to keep track of the text. For the outer loop, I went through all the unique events and labeled it the pair based on the loop iterator. Using the counter variable I was able to count which place in the cases I was at, to determine what to print for the prefix information and what to print for the suffix information as seen in the screenshot below.



```
def presuf(events):
    lines = []
    for i in range(0, len(events)):
        lines.append('Case: {}\n'.format(events[i][0].eventID))
        count = 1
        for k in range(0, len(events[i]) - 1):
            lines.append(str(k + 1))
            lines.append(' Prefix: ')
            for m in range(0, count):
                lines.append('{} '.format(events[i][m].eventletter))

            count = count + 1
            lines.append('\n')
            lines.append(str(k + 1))
            lines.append(' Suffix: ')

            for q in range((k + 1), len(events[i]), 1):
                lines.append('{} '.format(events[i][q].eventletter))
            lines.append('\n')
            lines.append('\n')

    file = open('prefixsuffix.txt', 'w')
    file.writelines(lines)
    file.close()
```

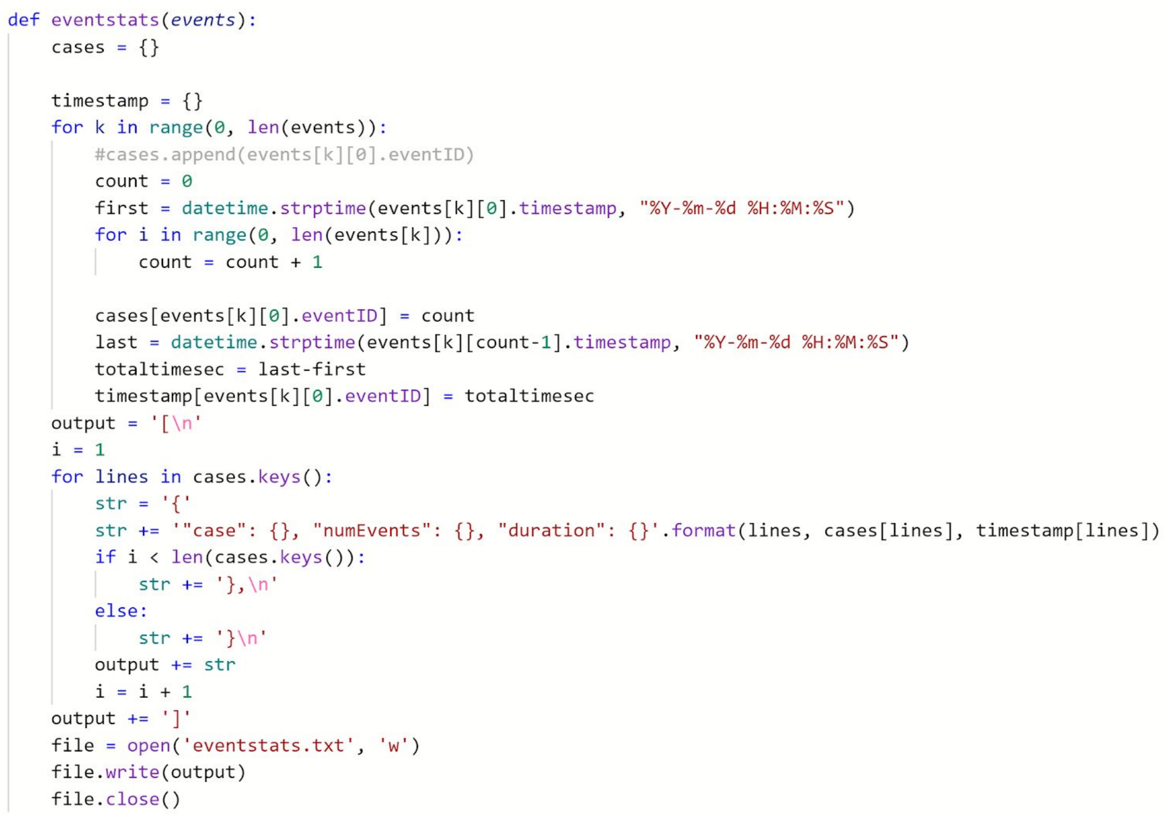
By going through the entire file, the prefix information was outputted as shown below.



prefixsuffix.txt

```
Case: 2
1 Prefix: 1
1 Suffix: 8 6

2 Prefix: 1 8
2 Suffix: 6

Case: 3
1 Prefix: 1
1 Suffix: 8 6

2 Prefix: 1 8
2 Suffix: 6

```

The event stats function was modified in the Python version of the project. To output in a correct JSON format, I had to keep track of the duration of each case.



```python
def eventstats(events):
    cases = {}

    timestamp = {}
    for k in range(0, len(events)):
        #cases.append(events[k][0].eventID)
        count = 0
        first = datetime.strptime(events[k][0].timestamp, "%Y-%m-%d %H:%M:%S")
        for i in range(0, len(events[k])):
            count = count + 1

        cases[events[k][0].eventID] = count
        last = datetime.strptime(events[k][count-1].timestamp, "%Y-%m-%d %H:%M:%S")
        totaltimesec = last-first
        timestamp[events[k][0].eventID] = totaltimesec
    output = '[\n'
    i = 1
    for lines in cases.keys():
        str = '{'
        str += '"case": {}, "numEvents": {}, "duration": {}'.format(lines, cases[lines], timestamp[lines])
        if i < len(cases.keys()):
            str += '},\n'
        else:
            str += '}\n'
        output += str
        i = i + 1
    output += ']'
    file = open('eventstats.txt', 'w')
    file.write(output)
    file.close()
```

By creating a case dictionary and a timestamp dictionary I was able to match up the duration of each specific case. I used an import for time, to parse the CSV file effectively to grab the time in seconds of the first event in the trace, and the last event in the trace. By subtracting the two I was able to get the total in seconds of each trace. For the output, I used the same key-value pairs for both dictionaries to create a nice JSON formatted output which was written to a text file.

Conclusion

During this internship, I learned many new things about process mining and the fundamentals of everyday processes that can be analyzed with event logs. The hospital event log example gave me an eye-opening perspective on the uses of what process mining could be used for and how being able to parse data effectively can help someone perform a task. I had to go out of my way to learn new techniques to solve this problem effectively which further enhanced my problem-solving skills which can always be used in the future. The process of porting my code over from one language to another language was also a tricky task, considering I was never familiar with Python before this internship, however in the real world that skill may be needed to adapt quickly and pick up to something that I may have not seen previously. Learning syntax, techniques, and various libraries in Python did take time but I am grateful for that opportunity to push me ahead to learn new things. One of the challenges I had fun taking on was maintaining a proper timeline to stay on track with the project. I had created a timeline for myself at the beginning of the internship where I said I'd have this done by week one, finish another task by week two, and so on. It would be easy to get distracted or procrastinate and not get the job you assigned yourself for that week down and delay it for the end due date of the project. However, I was able to get all my tasks done weekly and present them to the professor and my mentor allowed me to show them what I had accomplished every week. I learned that in the real world, deadlines and timings are important and that I should space out my work so that I can complete it all in a reasonable time frame. One of my favorite parts of the project was creating the Python code after knowing the C++ code logic. Even though I knew the logic, being unfamiliar with the language caused me some issues and errors which I had to look up online to get rid of. I also learned the usefulness of a data processor which is efficient and can save time, because event logs could be tens of thousands of lines long and if the problem is not efficient, could take hours to just read in the data. By having an effective method of organizing the data, then sorting it, and filing it into various data structures, the information becomes easily accessible allowing for run times to be exponentially faster.