Team#1: CS 267 Data Mining Pipeline

##### Spring 2017

##### Group Members:

* Wei-Chung (Wayne) Huang
* Sidharth Mishra
* Samuel Ordonia
* Michael Symonds

This documentation provides a brief overview of the layout of the entire codebase of the project. It also talks about the work distribution over the various modules of the project. For more documentation about usage and setup related stuff please refer to the **README.txt** documents included with the project directories. The **README.txt** documents will walk you through the setup and usage procedures.

Note: For information about the usage of the various methods/functions of the PDF processor module, please refer to the `**pdf\_processing\_demo.py`** script**.**

**Warning: Please use Python v3.x. Minimum supported version is Python v3.5.2. For DocPruner and SimplicalComplexDataCreator Java(JRE) version should be Java8. The code heavily relies on lambdas and JDK-8 features. For the Apriori C++ port, for optimum compilation on Mac use MacOSX - gcc with language version set to C++11. On Windows, for the Apriori C++ port, please include `\_CRT\_SECURE\_NO\_WARNINGS` in the Preprocessor Definitions of the Visual Studio 2013 project for a successful build.**

### PDF Processor module:

**Github:** <https://github.com/sidmishraw/cs-267-project/tree/master/pdf_processing>

Author: Sidharth Mishra

Tool(s): Python v3.x, PDFMiner by Yusuke Shinayama, Chardet

The PDF processor module(pdf\_processor.py) is located inside the `pdf\_processing` package.

#### High level overview:

The module uses Yusuke Shinayama’s `PDFMiner` library to parse and decompress bytes of the PDF document. It uses `chardet` library to recognize the encoding of the bytes obtained after decompressing the PDF pages and then decodes it to obtain the Postscript code from it. After obtaining the PostScript code for each page, the module then uses regex to extract out the actual needed text and makes it into a JSON file.

#### Detailed overview of the module(methods needed to use the module):

**get\_pdf\_contents(pdf\_file\_name)**: Takes the PDF file name(qualified name) and returns the mapping of the contents of the pages of the PDF document. The contents of the PDF document is extracted from the `Catalog` of the PDF document.

**extract\_words()**: Preserves the word's position from the PDF to the JSON. Extracts the words(textual)/phrases from the decoded(decompressed) contents of the PDF pages and returns a python dict holding the mapping from page#number to the words in the page.

**cleanse\_extracted\_words(pdf\_page\_words\_dict)**: The next phase after obtaining pdf\_page\_words\_dict from `**extract\_words()**`. This will cleanse the words removing hyphenation and other symbols at the start or end of the words.

**build\_pdf\_json()**: The reading order of the words is not preserved but the grouping order is preserved. Extracts the text and groups similar text together depending on their font, size, weight behavior in the pdf document. For this reason their reading order is not preserved but they get logically grouped based on their roles in the document.

**cleansed\_pdf\_json(pdf\_dict)**: The next phase after obtaining pdf\_page\_words\_dict from `**build\_pdf\_json()**`. The reading order of the words is not preserved but the grouping order is preserved. The second pass that cleans the PDF JSON structure to make a listing of the words in just pages.

Note: More information on using the `**pdf\_processor.py**` module can be found in the `**pdf\_processing\_demo.py**` file. It has documentation and example usage of the module to obtain the JSON files.

Note: The modules are well documented and self explanatory.

### pdf\_downloader\_script.py

Author: Sidharth Mishra

Tool(s): Python v3.x

#### High level overview:

The `pdf\_downloader\_script.py` will ingest the PDF documents located in `input\_pdfs` directory and output the generated JSON documents into `pdf\_jsons` and `pdf\_grouped\_jsons` directories.

### Utilities:

Authors: Michael Symonds, Sidharth Mishra

Tool(s): NLTK library, Python v3.x

#### High level overview:

The `utils.py` module is located in the `pdf\_processing` package and fetches stop words and standardizes the words obtained from the PDF JSONs by stemming them and removing stop words.

#### Detailed overview of the module(methods needed to use the module):

**fetch\_stopwords(language)**: Fetches the stopwords from the files downloaded from NLTK's corpora APIs.

**standardize\_words(words)**: Takes the list of words it needs to standardize and then returns a list of standardized words. The list of standardized words contains only single words, all phrases are removed so goes for empty strings. Fetches the stopwords from the words list downloaded by NLTK.

### DocPruner:

**Github**: <https://github.com/sidmishraw/docpruner>

Author: Sidharth Mishra

Tool(s): Java 8

#### High level overview:

Prunes the bad PDFs(probably scanned images of IEEE documents from IEEE Xplore) and moves them out of the input\_pdfs folder and moves folders pdf\_jsons and pdf\_grouped\_jsons out of the cs267\_project folder so that the PDF - JSON generation process can be started from scratch.

### data\_compare.py

Author: Sidharth Mishra

Reviewed by: Michael Symonds

Tool(s): Python v3.x

#### High level Overview:

Script to compare the data generated by the pymining’s seqmining module and our C++ port for Apriori. It lists the differences it found in the 2 outputs if they are different else it just has a blank screen.

### SimplicalComplexDataCreator

**Github**: <https://github.com/sidmishraw/simplicalcomplexdatgen>

Author: Sidharth Mishra

Tested and reviewed by: Michael Symonds

Tool(s): gson, java 8

#### High level overview:

This a jar file. This application reads the PDF JSON files(cleansed) from the `output\_s\_jsons` directory and generates the `.dat` files inside the `output\_s\_dats` directories. It generates `s\_output.dat` file that maintains the reading order of the word indices and `sorted\_s\_soutput.dat` that contains the sorted word indices in each row, so, the reading order is not maintained.

It also generates the `word\_map.json` and `doc\_map.json` files that have the entire mapping for all the words in the document set and documents in the document set respectively.

### main.py

Authors: Sidharth Mishra, Samuel Ordonia

Tool(s): Python v3.x, NLTK library

High level Overview:

This is the main driver for the pipeline. It runs the `**read\_input\_files()**` from the `build\_tables.py` module to cleanse and stem the JSON documents obtained from the PDF processor and outputs them to the `output\_s\_jsons` directory.

Note: This portion for now only runs the `read\_input\_files()` from the preprocessor `build\_tables.py` module to generate the cleansed JSON documents into `output\_s\_jsons` directory. This is then ingested by the `**SimplicalComplexDataCreator**` to generate the `.dat` files. In order to use the Python - C++ Simplicial Complex bridge, first you'll need to build and recompile the source of the Simplicial Complex bridge on your machine/platform and obtain the shared libraries. This is because of the nature of shared libraries. Then, uncomment the commented/muted code in the `main.py` script and run it. The script's source is well documented and is self explanatory.

### process\_results.py

Author: Sidharth Mishra

Tool(s): Python v3.x, Chardet

#### High level overview:

Processes the `results.txt` obtained by running either Simplicial Complex C++ program or the Apriori C++ port and uses the `word\_map.json` file to obtain the words from their respective indices and outputs the final result into `final\_simplicialcmplx\_results.json` or `final\_apriori\_results.json` depending on the application that produced the `results.txt` file.

### Bat/Shell scripts for setup and run of the pipeline and README.txt:

* **start\_json\_generation.bat** and **start\_json\_generation.sh**
* **dat\_file\_generation.bat** and **dat\_file\_generation.sh**
* **process\_results.bat** and **process\_results.sh**

Author bat/shell Scripts: Sidharth Mishra

Author README.txt: Michael Symonds

#### High level overview:

The `**start\_json\_generation.bat**` and `**start\_json\_generation.sh**` automate the process of generating JSONs from the `input\_pdfs`.

The JSON files will be generated in `pdf\_jsons` directory and the Grouped JSON files will be generated in `pdf\_grouped\_jsons` directory. It runs the PDF processor on the PDF documents in `input\_pdfs`.Then, it runs the **DocPruner** to weed out bad PDF documents from the document set.

If the Output is "Successfully moved the generated files", the bad PDFs will be pruned and moved into `**badPDFs**` directory and the generated directories `pdf\_jsons` and `pdf\_grouped\_jsons` will be moved into `**badGeneratedFiles**` directory. Verify your PDF counts, replace the bad PDFs and re-run this batch/shell script.

Else, if it said "Nothing to move :)" you can now run the `dat\_file\_generation.bat` to generate the dat files.

The `**dat\_file\_generation.bat**` and `**dat\_file\_generation.sh**` automate the process of generating the cleansed JSON documents into `**output\_s\_jsons**` directory by running the **preprocessor** on the JSON documents obtained from the PDF processor and then running the **SimplicialComplexDataCreator** on those cleansed JSON documents to obtain the `s\_output.dat` and `sorted\_s\_output.dat` files inside `output\_s\_dats`. The `word\_map.json` and `doc\_map.json` are also generated by the **SImplicialComplexDataCreator** at the project directory root.

The `**process\_results.bat**` and `**process\_results.sh**` automate the process of obtaining the final word patterns from the word indices by running the `**process\_results.py**` script on the `**results.txt**` file.

The `**README.txt**` has detailed information on using the individual scripts and setup of the Python environment. It walks you through the process of running the scripts and obtaining the `.dat` files which can be used with the Simplicial Complex C++ program and Apriori C++ port.

Note: For more information about the Apriori C++ port, please refer to the `README.txt` file inside its project folder.

### Apriori C++ Port:

Github gist(Mac/Linux version): <https://gist.github.com/sidmishraw/2fba1cc478eda676150d266735aa89a9>

Authors - Sidharth Mishra, Michael Symonds

Tool(s) - C++ (language standard C++11), gcc, Visual Studio 2013

Requirements - x64 operating system/ 64 bit operating system (recommended), atleast 8GB RAM for larger document sets with low thresholds.

#### High level overview:

Apriori C++ Port is a C++ port of the apriori algorithm implemented in the python version found in pymining/seqmining on the Github repository at: <https://github.com/bartdag/pymining/blob/master/pymining/seqmining.py>

For a more detailed description about the program and the procedure to run on a Windows machine, please refer to the included **README.txt** with the Visual Studio project solution package.

For running on a Mac OSX/Linux machine, please install Eclipse CDT plugin and copy in the source file `**apriori\_unix.cpp**` into a new C++ project or build the source file `**apriori\_unix.cpp**` to get the executable. The source should be compiled with compile with language standard set to C++1x (preferred C++11).

This is a recursive implementation of the apriori algorithm which retains the order of the frequent itemsets being counted. It takes in the number of rows(documents in the document set), number of columns(total number of words in the document set), the path to the `**s\_output.dat**` or `**sorted\_s\_output.dat**` file and the minimum threshold(a decimal number representing the percentage, for eg - 0.5 = 50% and 0.001 = 0.1%). Unlike the Simplicial Complex program, it doesn’t take an limit for the maximum number of rules to be calculated. It will keep on computing the frequent patterns till it finishes scanning through the entire document set and has found all possible patterns for the given threshold limit or the program runs out of main memory. To find the total number of words on the document set, please look at the `**word\_map.json**` document obtained by running the **SimplicalComplexDataCreator** application. For the total document count `doc\_map.json` is the document to refer.

The source code is well documented and has loggers muted. For a better understanding of the algorithm, we recommend using a small dataset and enabling all the loggers in the source code. It will give a better walkthrough on how the recursive calls for computing the frequent patterns work. (By default, only the finding of frequent pattern is logged.)

The program writes the frequent patterns it found to the `**results.txt**` file. It can be monitored by doing a `tail -f results.txt` on a Mac/Linux machine and opening the results.txt file in Notepad++ like application on windows.

Note: Please build the source on a 64 bit machine for better performance. Since, apriori doesn't have an upper limit for the maximum size of pattern to look for, it will try and find all the patterns possible. This goes on till it finds all the patterns possible or it runs out of memory. For larger document sizes with a larger number of words, it is possible for the program to fail building if not built in the x64 mode in case of Windows(Visual Studio). There is no such problem on Mac OSX since it has 64 bit OS only. Same goes for Linux, need to be careful about the underlying OS while building from source.

**Warning**: **On Windows, please use Visual Studio 2013 onwards to build the source. The minimum language standard is C++11. Please include `\_CRT\_SECURE\_NO\_WARNINGS` in the Preprocessor Definitions of the Visual Studio 2013 project for a successful build.**

**Warning**: If you copy the C++ source code of the Apriori port and plan to run it on a Mac OSX/Linux machine, please replace the statement on line 446 to `runtimeAlgorithm = (double) (difftime(clock(), clockStart) / CLOCKS\_PER\_SEC);` and build it with gcc. Also, don't forget to remove the windows specific headers `windows.h` and `psapi.h`. The main reason for replacing line 446 is because different machines/platforms have different clock speeds.

### process\_simplicial\_complex.py

Author: Wei-Chung Huang

Tool(s): Python, Shell Script

#### High level overview:

This script will automatically build the simplicial complex and run it with the dat file which is generated from user’s input PDFs.

Before using this script, user needs to generate the necessary dat file from the scripts **`start\_json\_generation`** and **`dat\_file\_generation`**.

There are two versions of scripts in Windows and Linux which are process\_simplicial\_complex.bat and process\_simplicial\_complex.sh respectively.

### Python - Simplicial Complex Bridge and shared library version:

Author: Wei-Chung Huang

Tool(s): C/C++, gcc version 4.8

#### High level overview:

This portion is used to be a bridge from Python PDF processor to simplicial complex. Generally the original simplicial complex is a stand alone version, however this module integrated it to a shared library in order to make the whole pdf mining became a pipeline process. Considering the future use, this library created several APIs to facilitate high level development and testing.

#### Interface description :

* Requirements

Linux/Unix

gcc 4.8 or higher

* Compilation

$> g++ -fPIC -shared SCExport.cpp SimplicialComplexNew.cpp -o SCExport.so

* API manuals - SCExport (.so)
  + **createInstance()**: Create an instance of Simplicial Complex

*Output*:

-- instance of Simplicial Complex -> objType

* + **removeInstance()**: Release instance resource

*Input*:

-- instance of Simplicial Complex -> objType

* + **initialize()**: Initialize an instance

*input*:

-- instance of Simplicial Complex -> objType

-- number of rules -> int

-- threshold -> float

-- number of columns -> int

-- number of rows -> int

-- input file path (set NULL is there is no input file) -> string

* + **setBitMapRow()**: Set the bit-vector for each row of the bit-map

*input*:

-- instance of Simplicial Complex -> objType

-- row length -> int

-- row index in bit-map -> int

-- raw data (string vector) -> string

* + **process()**: Process with the user initialization

*Input*:

-- instance of Simplicial Complex -> objType

* + **directProcess()**: Static process without creating instance from user side

*Input*:

-- number of rules -> int

-- threshold -> float

-- number of columns -> int

-- number of rows -> int

-- raw data (string vector) -> string

* Example usage 1
  + Input a file path to initialize()
  + Call process() directly
* Example usage 2
  + Set NULL file path in initialize()
  + Call setBitMapRow() for every rows or input bitmap

Note:

* A general calling sequence would be like this:  
  createInstance() -> initialize() -> setBitMapRow() -> (loop) -> process() -> removeInstance()
* The output will be written into a file result.txt
* main.cpp provides a demo usage of this library

### Document pre/post-processing(~/apriori/build\_tables.py)

Author: Samuel Ordonia

Tool(s): Python (pref 3.6)

This portion of the pipeline occurs after the PDFs have been parsed into JSON files. The build\_tables script constructs the following:

* Inverted index of all terms per document. This data structure is then outputted to JSON files with tokens as keys and token position arrays as values.
* The token cardinality per document.

The inputs are the text or JSON files, depending on what is run before. If the PDF processor portion is run, it looks at the JSON files. This is recommended approach. Refer to ~/input\_pdfs/inputs for examples on different inputs sent through this portion.

The outputs for this portion can be found in the ~/outputs folder. The doc frequencies can be written to a csv or a text file, depending on the input file type of choice for Simplicial Complex. The inverted index is outputted to a JSON file, since it contains document-term-position key-value format.

Database can take in all the tokens. It doesn’t care about lower bound or upper bound. It stores all the key terms. This portion keeps everything and does not stem or filter anything.

The next phase of the data processing is to convert the input into a bitmap that can be fed directly into the Simplicial Complex using the Python-C++ bridge.

We take a document. It has tokens. Each token occurrence has a value, and it is sent via a bitmap into the Simplicial Complex. Repeat for each document (aka, pdf). Then, the Simplicial Complex executes.