

Alexander Crawford

100569102

CSCI 2020 – Project

April 18, 2015

“Related Term” Crawler

Problem

In advertising today, one of the most powerful tools being developed is targeted marketing. Targeted marketing is the process of determining the particular context an ad is being displayed in (eg. the topic of a webpage), and displaying advertisements relevant to that particular context. Google is an especially egregious example of this, as its targeted advertising scans email content or common search terms and produces relevant information. Through targeted advertising, companies and websites can more accurately ensure the right products are being pitched to the right customers.

However, one of the major drawbacks of current targeted marketing algorithms is their reliance on human input. While most targeted marketing algorithms are able to, say, determine whether a particular webpage is more suitable to show advertisements for cheap computer parts or advertisements about modern video games, there still needs to be human input to designate what related terms are (like “motherboard” being a computer part, or “Call of Duty” being a “video game”). This algorithm simulates the concept of automatic term distinction – being able to pull from certain resources, and determine the overarching theme related to a topic.

Design Theory

The following is an explanation of the basic theory behind how the crawler works, presented in step-by-step form. In this example, a few particular terms may become confusing, so here's a quick glossary:

Phrase: The user's key word or phrase – we specifically want to get the words that surround it.

Context: The user's submitted file, which contains the phrase. Provides information about how the word is used.

Sources: The server's catalogue of media, used to determine related terms. Each has an associated frequency chart, which details which words appear most frequently in it.

Degree of separation: The maximum distance away a word can be from the start or end of a phrase, and still be considered possibly relevant. A degree of separation of 1 indicates that the word must be adjacent to the phrase; a degree of separation of 2 indicates that a maximum of 1 word may be between the phrase and the word, and so on and so forth.

1. The client queries the server, providing a phrase and a context file.
2. The server assesses the context file, and pulls out the most frequent words in that file (discarding common words).
3. The server selects the most suitable sources by selecting those whose frequent words most closely match those of the context file.
4. The server crawls through its selected sources, looking for the phrase. Each time it finds the phrase, it takes the relevant words around it and records them, assigning a score based on how close it is to the phrase.
5. The server calculates the most relevant words by their closeness and frequency of appearance near the phrase.
6. The server returns the best words to the client.

Design Features (relevant to CSCI 2020)

- **Sockets – Server/Client Setup**: The related term crawler operates through two main programs: one ParserServer (located in the server folder) and one or many ParserClient (located in the client folder). The server maintains all the sources, and does all the calculation, while the client merely submits requests and contexts.
- **File I/O – Setting Storage**: The ParserClient's data for connecting to the ParserServer is not specified as a command-line argument – instead, a data file called config.dat exists that holds default data. Settings can be changed by running the ParserClientConfig file.
- **File I/O – Frequency Calculation Storage**: Frequency calculations done on sources on the ParserServer are stored in the "frequencies" folder, so frequencies don't have to be calculated each time. While on a small scale the time savings is negligible, for large files and large numbers of source files, this equates to a large amount of time saving, as the frequency charts for every source needs to be looked at to determine which the best sources may be.
- **Threads – Simultaneous Reading**: Threads are employed for doing 'best word' calculations simultaneously. Once the best sources to read from have been discovered, a separate thread is

dispatched to read each document. Thread safety is ensured by acquiring a lock specific to the document before reading, and acquiring a lock specific to the final tally before dumping information received from a specific thread.

- **Threads – Simultaneous User Access:** Threads are also used to allow multiple users to access the ParserServer simultaneously. The ParserServer constantly listens for connections, and dispatches a ClientThread object to deal with any incoming connections.
- **Threads – Maximum Thread Cap:** The ThreadManager class exists to ensure that the server is not bogged down by too many threads attempting to operate simultaneously. Any time a thread wants to activate, a request is submitted to the ThreadManager, which checks how many threads are running currently. If there's still room, it allows the thread to run and increments the number of running threads, but if there's not room, it adds the thread to an internal queue. Whenever a thread terminates, whether naturally or through exception, the `releaseThread()` function is called, which both decrements the number of threads currently running and checks whether the next thread in the queue can be run. In addition, to prevent deadlock situations, all threads that call the ThreadManager must give a minimum number of consecutive threads that must be running in order for the thread to make progress (including the thread itself), and if less resources than that are currently available, then the thread is pushed onto the queue.
- **Regex – Forward and Backward Word Parsing:** The words surrounding the phrase sought for are selected via regular expressions. A separate regular expression is created through for each potential "step away" the word could be (so that separate calculations can be made based on how far away the word is), by first matching the known phrase, and then counting right by the number of words desired. *However*, this becomes more complicated when looking for words left of the known word, as simply trying to match, say, a group of 8 words followed by the word "Alice" would result in a lot of backtracking with the regular regex. However, this program implements the ability to check backwards by literally reversing the searched string, the searching regular expression, and the stream the regex is sifting through! This allows us to easily check forward and back for words that we want.

Project Structure

This section details the directories and files present in the project, and their purposes.

-src

-server – Contains the server code.

-sources – Contains .txt source files. Server automatically reads them when started. Files in here are [OPTIONAL] and can be deleted or replaced at your leisure.

-frequencies – Contains .dat files of source file word frequency. Automatically generated and loaded if detected; can be deleted to force the server to recompile and regenerate frequency lists.

-exclude.txt – List of common words for the program to exclude when looking for relevant words.

-ParserServer.java – Server. Contains all the relevant code; broken down in the next section.

-client – Contains the client code.

-ParserClient.java – The main client program. Communicates with the server.

-ParserClientConfig.java – Manages ParserClient’s connection host/port.

-config.dat – Holds connection information. Created by ParserClientConfig.

-example.txt – [OPTIONAL] A sample context file to query the server with.

ParserServer Class Breakdown

This section details the main classes that ParserServer contains, and what their purpose is.

- **ParserServer:** The main class; contains the main function. When it starts, it loads all the sources in the sources folder into memory (via DocumentThreads), and then waits for connections for clients (dispatched as ClientThreads).
- **FreqWord:** A comparable object that stores both a word and an integer indicating its frequency of appearance.
- **Document:** An object that represents a text file and attached frequency table. Features save and load commands that access the equivalent frequency table in the frequency folder (assuming the document represents a source).
- **WaitingThread:** An object that stores a thread waiting to execute and the minimum number of consecutive threads it demands in order to make progress. Used in the ThreadManager queue.
- **ThreadManager:** A static class that organizes threads to ensure that not too many threads are run at the same time. Threads that want to run query this class, and are either allowed to run or queued up as WaitingThread objects, to be released (in order) once enough other threads have finished.
- **DocumentThread:** A thread that’s tasked with reading a source file and adding it to the ParserServer’s list of sources. These threads are only created and started by the ParserServer when the server is initialized.
- **SearcherThread:** A thread that’s tasked with searching through a pre-existing document, finding relevant terms near the phrase, and returning the frequency results to its parent thread. These threads are only called as a result of a search request, and therefore a class that extends RequestThread.
- **RequestThread:** An abstract thread class that implements some fundamental properties that should be a part of any request handling thread – specifically, the phrase being looked for, a map of words to frequencies, and the number of thread spaces to free up in the ThreadManager if this thread crashes unexpectedly.
- **ClientThread:** A thread that represents a request from a connected client, designated by a Socket object. This thread takes care of receiving user input, deploying SearcherThreads to the best sources, processing the received frequency data from SearcherThreads and using it to determine which are the best related terms, before sending them to the user.
- **ServerThread:** A currently defunct class that acts very similarly to the ClientThread, but instead represents a request from the server itself to do its own lookup, and display the results over

stdout. While it currently does not function in the code, the `self_test()` function in `ParserServer` triggers it – this class remains around for the purposes of future development.

- **ParserUtil:** A static class containing a collection of important functions that are either used in several different classes, or are likely to be changed drastically. These include:
 - **standardize(String):** Standardizes a word to be placed in the frequency tables. This currently entails stripping any non-alphabet characters from the beginning and the end of the word, and converting it to lowercase.
 - **loadDocument(String):** Creates a brand new Document object by reading from a file.
 - **loadDocument(BufferedReader):** Creates a brand new Document object by pulling lines from a stream, until the end command (<<end>>) is read.
 - **findBestSources(Document, Integer (number of words), Integer (number of documents)):** Returns an array of the best source documents, matching most closely the document provided in the arguments. Currently, this is done by matching a certain number of most frequent words (provided in the arguments) from the context document to the frequency with which they appear in source documents.
 - **produceRegex(String,Integer (separation)):** Produces the complicated regex expression that is used for finding a word a certain number of steps away from the phrase. Longer explanation of what that regex means can be found in the source code.
 - **calculateQuality(Integer (proximity), Integer (max proximity)):** Produces a value for the purposes of determining whether a word found by the phrase is relevant or not. Right now, the result of this calculation is by making a word one step further away than another from the phrase be worth half as much.

Usage & Options

This section lists how each class can be used, and how it should be called at the command line.

PARSERCLIENT:

`java ParserClient [context] [topic] [flags]` – Scans the source provided for the particular topic to generate a best-guess as to the supertopic. Then, picks the best resources on that topic that contain the specified phrase, and picks the best topics, returning them to the user.

[context]: The .txt file that contains context information.

[topic]: The word or phrase that will become the phrase the server will search for.

[flags]:

-n [num]: Number of sources. Defaults at 3 – given more, the program will use more sources to calculate the related terms.

-d [num]: Degrees of separation. Defaults at 3 – given more, the further away the program will check for related terms. Less, the closer.

-simple: Simple output mode. Instead of outputting all the bells and whistles, runs silently, and just returns the #1 word.

PARSERCLIENTCONFIG:

java ParserClientConfig – Edits the connection configuration for the parser client through an interface. It will ask you for a host name and port number (default: 127.0.0.1 and 4444, respectively), and store the result in config.dat.

PARSERSERVER:

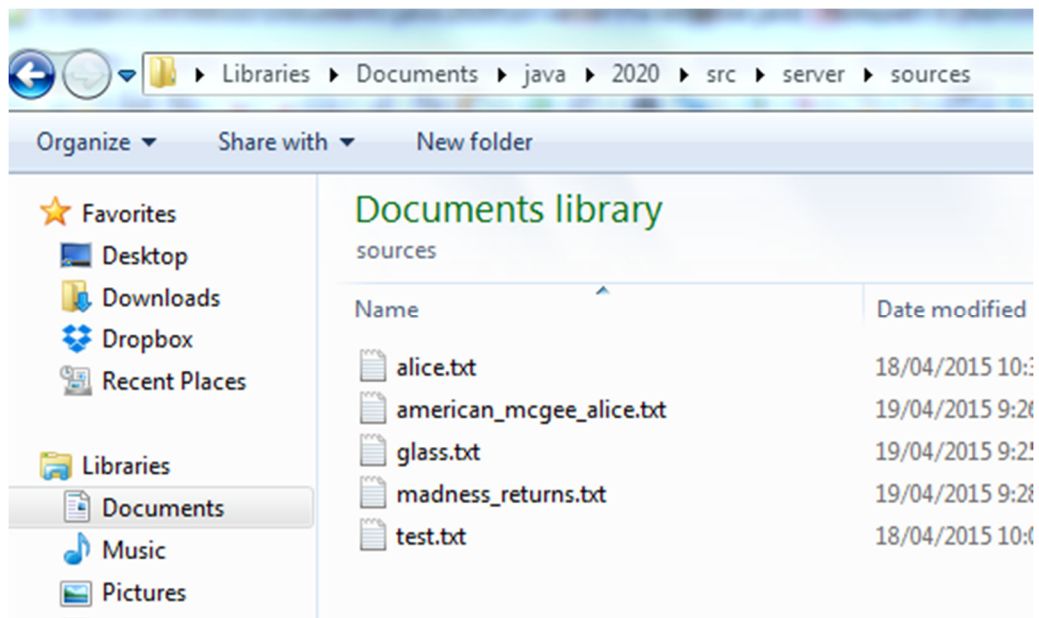
java ParserServer [port] - Deploys the server. The server will continue to loop endlessly, accepting connections – simply closing the command line window or pressing CTRL+C will terminate the program.

[port] – The port number to begin the server on.

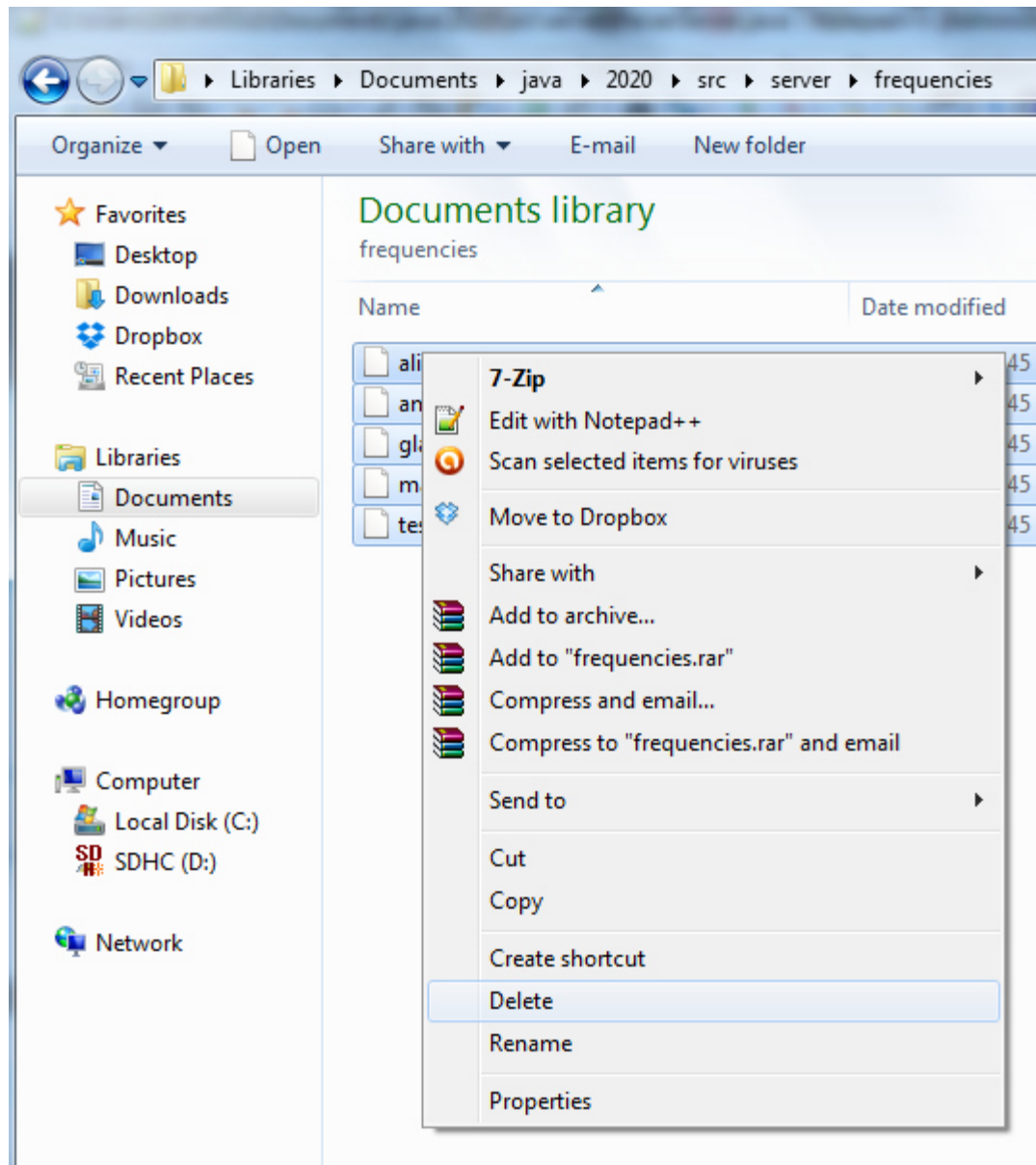
Execution Instructions

This section explains the process for running the tests, including which files should be placed where.

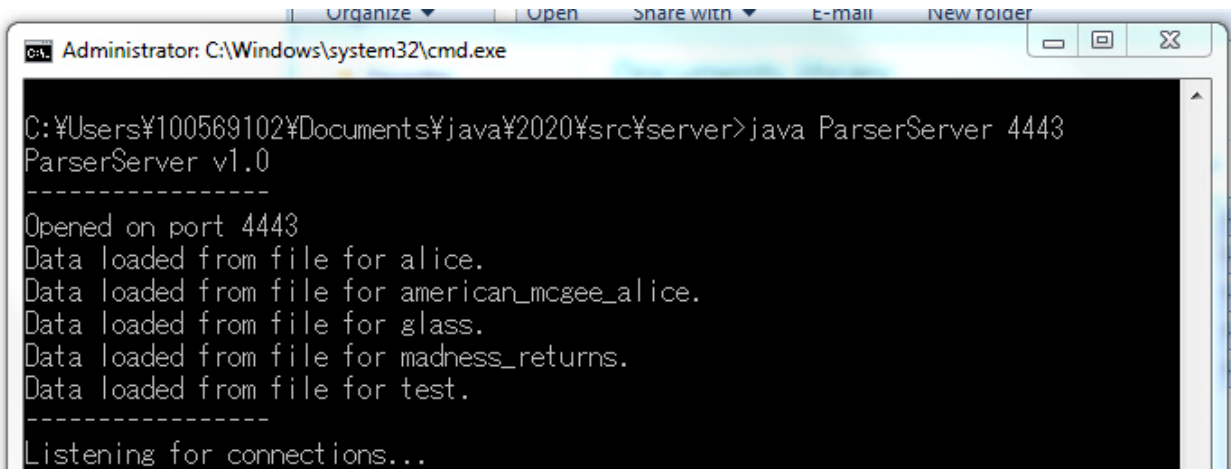
Serverside



1. Place any source documents you want the server to be able to read in the /server/sources folder.



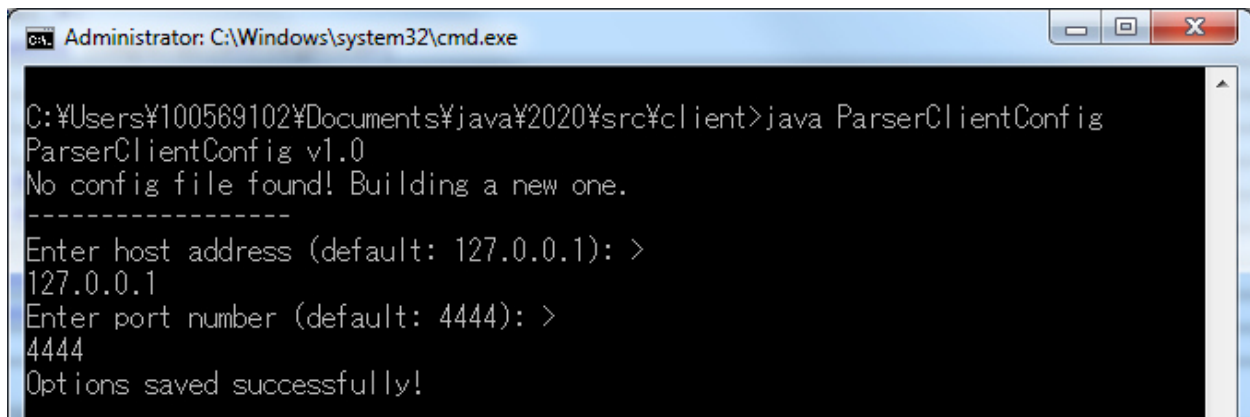
2. [OPTIONAL] Delete some or all of the .dat files in the frequencies folder. (This will make the program boot more slowly, but it forces a recompile of the frequency lists of any sources that may have changed.)



```
C:\Users¥100569102¥Documents¥java¥2020¥src¥server>java ParserServer 4443
ParserServer v1.0
-----
Opened on port 4443
Data loaded from file for alice.
Data loaded from file for american_mcgee_alice.
Data loaded from file for glass.
Data loaded from file for madness_returns.
Data loaded from file for test.
-----
Listening for connections...
```

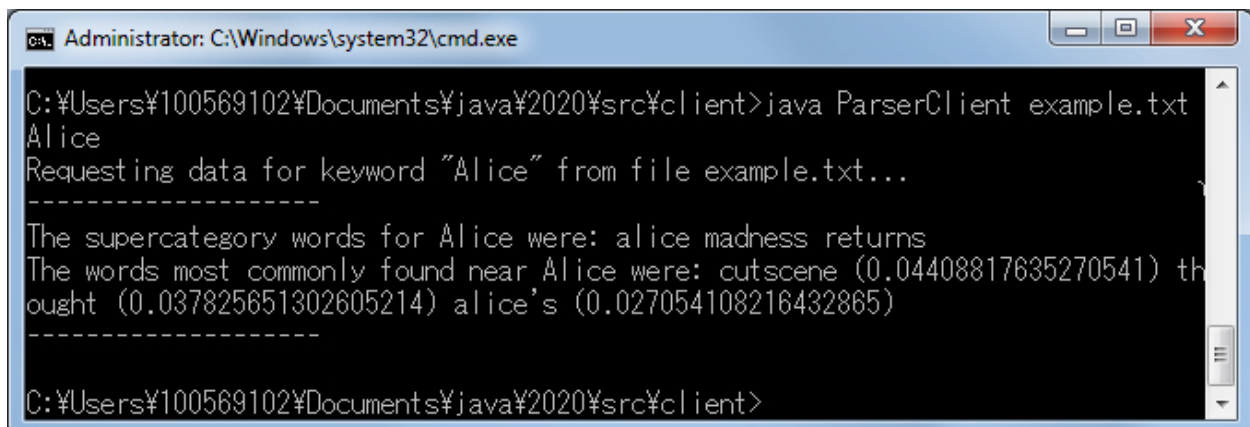
3. Run the server program. [OPTIONAL]: Specify the port number as a command line argument.

Clientside



```
C:\Users¥100569102¥Documents¥java¥2020¥src¥client>java ParserClientConfig
ParserClientConfig v1.0
No config file found! Building a new one.
-----
Enter host address (default: 127.0.0.1): >
127.0.0.1
Enter port number (default: 4444): >
4444
Options saved successfully!
```

1. Run ParserClientConfig, and answer the questions. Note that this step is optional, if the config.dat file already exists and contains correct server connection information.

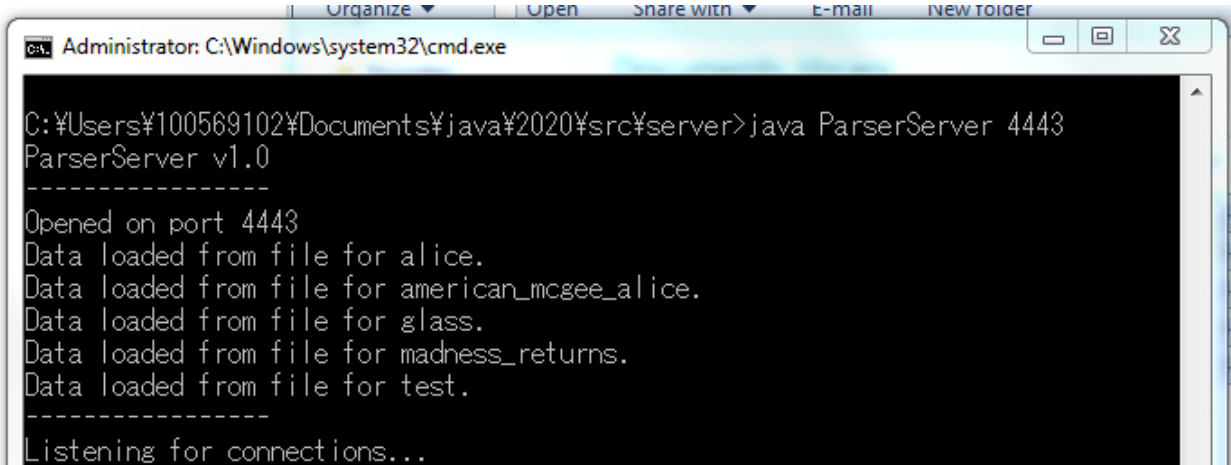


```
C:\Users¥100569102¥Documents¥java¥2020¥src¥client>java ParserClient example.txt
Alice
Requesting data for keyword "Alice" from file example.txt...
-----
The supercategory words for Alice were: alice madness returns
The words most commonly found near Alice were: cutscene (0.04408817635270541) th
ought (0.037825651302605214) alice's (0.027054108216432865)
-----
C:\Users¥100569102¥Documents¥java¥2020¥src¥client>
```

2. Run ParserClient. The bare minimum command line requirements are the path to the context file, and the phrase to investigate.

Example Execution

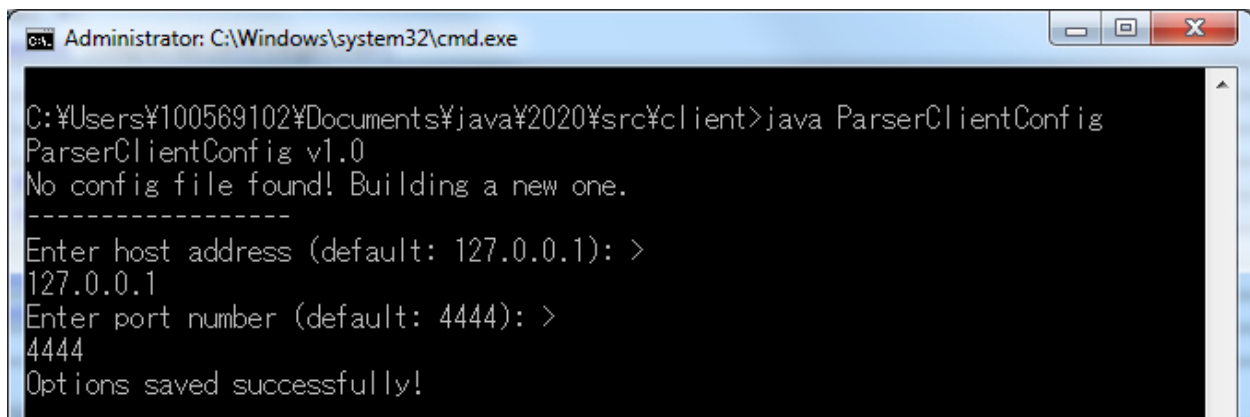
The environment provided in the file is set up to run a sample test, using four source files (Through the Looking Glass, Alice in Wonderland, and two scripts from media based on Alice in Wonderland) and a sample file (a game review of an Alice in Wonderland videogame). It can be run locally as follows:



```
Administrator: C:\Windows\system32\cmd.exe

C:\Users\¥100569102¥Documents¥java¥2020¥src¥server>java ParserServer 4443
ParserServer v1.0
-----
Opened on port 4443
Data loaded from file for alice.
Data loaded from file for american_mcgee_alice.
Data loaded from file for glass.
Data loaded from file for madness_returns.
Data loaded from file for test.
-----
Listening for connections...
```

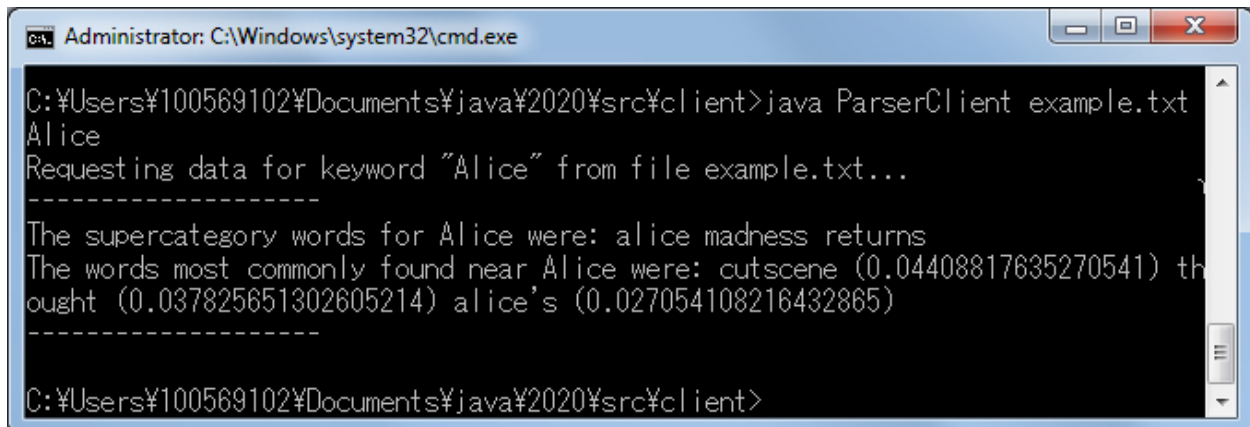
1. (in the server directory) `java ParserServer 4444`



```
Administrator: C:\Windows\system32\cmd.exe

C:\Users\¥100569102¥Documents¥java¥2020¥src¥client>java ParserClientConfig
ParserClientConfig v1.0
No config file found! Building a new one.
-----
Enter host address (default: 127.0.0.1): >
127.0.0.1
Enter port number (default: 4444): >
4444
Options saved successfully!
```

2. (in the client directory) `java ParserClientConfig`



```
Administrator: C:\Windows\system32\cmd.exe

C:\Users\¥100569102¥Documents¥java¥2020¥src¥client>java ParserClient example.txt
Alice
Requesting data for keyword "Alice" from file example.txt...
-----
The supercategory words for Alice were: alice madness returns
The words most commonly found near Alice were: cutscene (0.04408817635270541) th
ought (0.037825651302605214) alice's (0.027054108216432865)
-----
C:\Users\¥100569102¥Documents¥java¥2020¥src¥client>
```

3. (in the client directory) java ParserClient example.txt Alice

Future development

Despite the features of the program, there is still a great amount of room for improvement. Future improvements could include:

- **Better error handling:** Right now, the error handling code is limited and sometimes fairly spotty. As long as the system receives the input expected, it works just fine, but there may well be cases not yet accounted for.
- **Better file reading:** Files are currently read line-by-line, as opposed to all at once, making regex testing complicated. The current code does contain a dummied-out function for converting a file to a stream for more accurate regex, but this becomes more complicated for the purposes of client-server interactions, as the client still sends its context file to the server line-by-line – changing that would require some considerable restructuring of the code.
- **Security:** Currently, the server accepts any connection freely, provided it structures its request appropriately. In a real-life situation, you would only want paying customers to be able to use the ParseServer's database.
- **Less naïve algorithms:** The way the code determines what phrases are relevant right now is fairly naïve, and is based more on a best-guess approach than any practical knowledge. The current procedure is to make the frequency of words that appear 2 steps away from a phrase worth $\frac{1}{2}$ as much as words that appear 1 step away, make words 3 steps away worth $\frac{1}{2}$ that of 2 steps away, and so on and so forth. However, the calculation for determining value of a word based on its distance is currently removed from the code in such a way that it would be very easy to edit, leaving lots of room to improve without significantly damaging the main code.
- **Server commands:** The server currently loops infinitely while waiting for connections, and therefore is currently unable to accept other commands while operating. If the connection-acceptance loop was placed in another thread, the server could then accept commands, such as a manual command to re-evaluate source frequency files, or to quit the program.