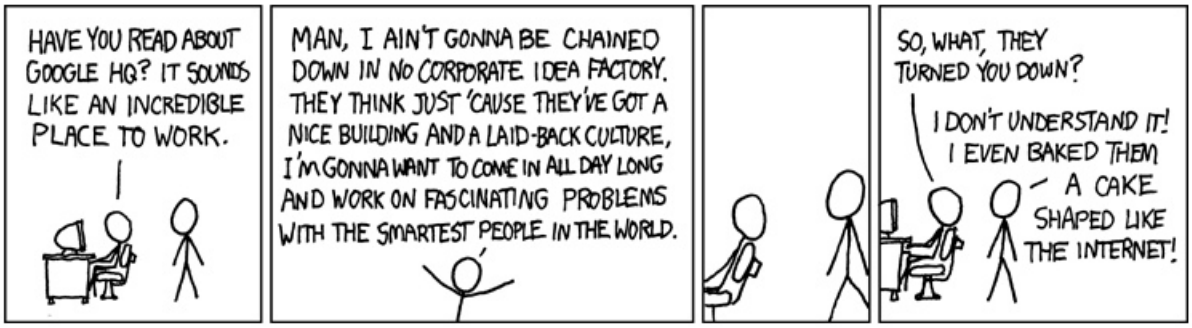
CS458 - Assignment 1
Due: Friday Sept. 21 at 6pm



The goal of these assignments is to give you experience in developing a working IR system. We will build from the ground up each component of the system. In the end, we will have a barebones system that does basic retrieval and will be the framework for your final project, where you can further extend some aspect that you find interesting.

The first step for an IR system (and for most natural language processing systems) is to to read in the text and decide what will be the processing unit to be worked with. For your first assignment, you will write a program that will tokenize the documents and we will experiment with different tokenization and token normalization techniques. You will see how these different techniques affect the dictionary size.

Before starting this assignment read through the **entire** document. At the end I've included some helpful hints and tools that may be useful. If there is any ambiguity or question about what you are being asked to do, ask me!

1 Get the basic code up and running

The Java starter code can be found in:

```
/home/dkauchak/PUBLIC/cs458/assign1/code/
```

You may use whatever environment you like. If you're going to use Eclipse (which I strongly encourage), these steps will get you started.

1. Create a new project: File → New → "Java Project". Enter "assignment1" for the project name and click "finish".

2. Grab the starter files from the location above and copy them into your source directory (replace "<workspace>" with the location of your workspace and "<starter_location> with the location of the starter code you downloaded).

   ```
   cp -r <starter_location>/search <workspace>/assignment1/src
   ```

3. Refresh the items so they show up in Eclipse. Click on the project and hit "F5" or right-click on the project and select "Refresh"

4. The code should compile, but there is not a main class yet, so it won't run.

2 Initial corpus statistics

Depending on how we tokenize the text and how we normalize the words, the size of our dictionary is impacted.

1. Read through all of the classes and make sure you understand what each class does/represents.

2. Finish the `Dictionary` class.

   This class should keep track of the number of **unique** strings that are added. For example, if we call `addWord` four times with "test", "test", "test2", "banana", then the size of the dictionary would be 3. If we call `addWord` four times with "test", "Test", "test2", "banana" the size of the dictionary would be 4.

3. Finish the `TDTReader` class.

   The class should iterate through the corpus file and generate new `Document` objects for each article. The document IDs for the documents should start at 0 and increment by one for each new document read. We'll be using a moderate sized corpus consisting of news articles. The documents are concatenated into one large file and each document is delimited by

   ```
   <DOC> ... <\DOC>
   ```

   I've given you two data files at:

   ```
   /home/dkauchak/PUBLIC/cs458/assign1/data/
   ```

   They're gzipped (so you'll need to use `gunzip` before using them). `tdt-corpus.sample` is a small sample of the entire data set and is a good one to use when you're debugging your code. When you're ready, the entire corpus is in `tdt-corpus.text_only`

4. Get some initial statistics about the number of words in the full corpus. You'll need to create a new class, call it `Experimenter` (you won't hand this in). To get some initial statistics we'll use a very basic tokenizer which I've provided. Do the following in a function in the `Experimenter` class:

- Create a new `SimpleTokenizer` object
- Create a new `TDTReader` object to read the data from the TDT data file
- Set the `SimpleTokenizer` as the tokenizer for the reader
- Create a new `Dictionary` object
- Iterate through all of the documents in the file using the `TDTReader` and add all of the words to the dictionary
- Output the number of words in the dictionary
- Call this function from a main class. You should get a dictionary size for the entire corpus in the 100K-200K range.

3 Improved tokenizer

The tokenizer we used so far only splits the text based on whitespace. We're going to implement an improved tokenizer that does the following:

- Tokens are delimited by whitespace
- Single quotes at the beginning and end of words should be separate tokens
- Numbers should stay together. A number can start with a '+' or a '-', can have any number of digits, commas and periods interspersed, but must end in a digit (note this is a more general definition that accepts things like "192.168.1" and other things like "-129.,24.34").
- An abbreviation is any single letter followed by a period repeated 2 or more times. In regex terms, "`(\w\.){2,}`". For example, "I.B.M.", "S.A.T." and "p.m." are all valid abbreviations, while "Mr." or "I.B" are not. All abbreviations should have the periods removed, i.e. "I.B.M" becomes "IBM".
- Finally, "`. , ? : ; " ‘ ( ) % $`" should all be treated as separate tokens, regardless of where they appear (as long as they don't conflict with the above rules). So "$10,000" becomes two tokens "$" and "10,000" and "I wondered,is this a test?" becomes 8 tokens, with the "," and "?" as separate tokens.

Finish the `ImprovedTokenizer` class to reflect the above tokenization rules. I've included a tokenization of the sample corpus with the other data files (`tdt-corpus.sample.token`).

Before you do any actual coding, I suggest you write a bunch of test cases that test various types of input. I've included a few examples of how you might do this in the skeleton class file. This way, you can run it over and over again as you make incremental changes to see what progress you're making and also make sure that you didn't break something that was working already. This can be tricky to get right, so it is extremely beneficial to spend 15-20 minutes up front filling in some test cases.

You may do this however you like, but my suggestion would be to do it in four steps:

1. split up the tokens initially based on whitespace and the special characters
2. do another pass over this set of tokens and take care of the single quote constraint (note, you can't just split on all single quotes, since this would break up words like "aren't")
3. do a second pass over this new set of tokens and put back together abbreviations
4. do a final pass and put back together numbers.

4 Data normalization

The last step to finish up our tokenization system is to implement some token normalization techniques to further reduce the size of our vocabulary. Finish the `TokenProcessor` class. This class should support the following optional normalization techniques:

- lowercase: Lowercase all letters in the tokens
- stem: Stem all the tokens. I've provided a class "Porter" that implements the Porter stemming algorithm. Use the "stem" function of this class to stem the tokens.
- numbers: Replace all numbers in the data with "`<NUM>`". Use the same definition of what is a number as described in the tokenization section.
- stoplist: Remove from the list of tokens any strings that occur in the `stoplist`. Note stoplisting is case insensitive, so if "about" is in the stoplist then you should remove "about", "About", "ABOUT", etc.

A few further rules for data normalization:

- Apply the stoplist removal **before** doing any stemming
- Be efficient about looking up whether a token exists in the stoplist
- If you're lowercasing and stemming, apply lowercasing before stemming
- If a stoplist hasn't been set, you're program should still run and just not remove any tokens.

5 Impact of tokenization and normalization on dictionary size

We now have everything in place to experiment a bit with our data. Earlier, we measured the dictionary size using the `SimpleTokenizer`. We're going to compare that with our other techniques. You'll need to make one small change to your `Experimenter` class to add the data normalization. Do this. Included with your code, you must handin a short document with the following in a *table*:

1. A list of the dictionary sizes using your code for the following settings:

- `SimpleTokenizer` alone (from above)
- `ImprovedTokenizer`
  All settings below, assume the `ImprovedTokenizer`:
- number folding
- lowercasing
- stemming
- stoplist using the provided `stoplist`
- stoplist using the 30 most frequent tokens (hint, you don't actually need to code this up :)
- stoplist using the 150 most frequent tokens
- number folding AND lowercasing AND stoplist using the stoplist file
- All normalization techniques

2. In addition to the table of data, write a few sentences explaining your results.
3. Finally, also write a few sentences describing your recommendation for the settings of the above parameters. You may want to play with other combinations. Make sure to *justify* your answer.

Hints/Comment

- I'm not too worried about efficiency for the code for now, but it also shouldn't take forever to run. If it takes longer than a few minutes to run through the whole corpus you've probably done something really inefficient.
- For those of you still fairly new to Java, check out the following tools/resources in JavaDoc (search online) which likely will be useful for this assignment:
  - `java.util.regex.*` – and related, `String.match`, `String.replaceAll`, `String.split`
  - `java.lang.StringBuffer`
  - `java.util.HashSet`
- It's likely you'll run into memory issues at some point. To increase the size of the Java virtual machine memory allotment do the following:
  - Go to "Run Configurations...", by clicking on the downward arrow next to the button you use to run your program.
  - Select the "Arguments" tab
  - Under VM arguments enter "-Xmx512m". This tells the VM to use 512M for the heap. You can increase this further if need be, but 512M should do it for this assignment.
  - If you are running all of this from the command-line (instead of in Eclipse) you can just add this flag in your call to Java.

What to turn in and how to turn it in

- What to turn in:
  - A "jar" file of your code. To make my life easier (and to give you experience creating .jar files) you will hand in a jarred version of your code. Basically, this is like tarring or zipping up your data, but it's a format that allows you to link other java programs to your code. You can read more about it online. To generate a .jar file when you're ready, do the following:
    * Select File → "Export..."
    * Select Java → "JAR file" and click "next"
    * In the "resources to export", first, make sure that all boxes are unchecked
    * Then, open the projects/folders until the "search" package is showing. Select the box next to the search package. This will check both "search" as well as the "src" folder.
    * Make sure that both "Export generated class files and resources" and "Export Java source files and resources" are checked
    * Click the "Browse..." button and select a filename and location to save the file. You should save the jar file as "search.jar"
    * Click "Finish". This will create the jar file in the specified location.
  - A document describing your token statistics and your answers to the questions above. Any reasonable format is fine.
- Make a directory with your last names and the assignment number (e.g. `kauchak_scharstein1`). Put your jar file in this directory and put your writeup in the directory.
- `tar` and `gzip` the directory. For example:

  ```
  tar -cvf kauchak_scharstein1.tar kauchak_scharstein1
  ```

  then

  ```
  gzip kauchak_scharstein1.tar
  ```

- Submit the tar, gz file online (e.g. `kauchak_scharstein1.tar.gz`).