CMPUT 660 Assignment 2

Logan Gilmour

October 29, 2013

1 Introduction

I chose to analyze all of Github's .java files. My hope is to determine which projects are most popular among developers by examining who is using which projects. My goal is to create an index by which similar projects can be compared in order to determine which is more popular. It may also be possible to cluster the projects to recommend superior alternatives for given projects.

2 Analysis

My first attempt at analysis is a naive iterative algorithm derived from pagerank. This involves several steps.

First, I extract the data from Github. This is accomplished by iterating through a list of all Java projects hosted on Github extracted from the GHTorrent Mysql dump. For each repository in that list, I use Github's API to download a tarball of the most recent version of the repository. I then write each .java file in that archive directly to a running Hadoop Distributed Filesystem in the form of a sequence file of key-value pairs (the default format for hadoop intermediate data). The key is the id of the repository, and the value is the contents of the .java file. All intermediate files are in this same Hadoop Specific format for key-value pairs, which are not human-readable.

3 MapReduce

3.1 InitRank

The first job, 'InitRank', takes each of these entries and transforms it into a java qualified type name, an initial rank of 1, and a list of imports. This is accomplished using the Java Parser project. There is no reduce component to this job, as each file should represent a unique type. (This is not necessarily true, but I am assuming it for now.) I also increment a variety of counters here, to get an idea of how many java sources are parsing correctly, and how many contain the necessary components.

```
**Input**: $RepoId, FileContents$
**Output**: $Type, (Rank, ImportedTypes[])$

Parse the FileContents;
Make a qualified type out of the PackageDeclaration and
TypeDeclaration;
Make a list of qualified types out of the ImportDeclarations;

emit $QualifiedType, (1, ImportedTypes[])$;
```

3.2 Rank

Once the individual .java file contents have been parsed and transformed into a qualified type and its imports, the 'Rank' job can be run over the files. The Map function of 'Rank' takes the rank of the type passed, and then emits each imported type with an equal portion of the parent type's rank. We also emit the parent type as-is, in order to maintain the structure of the graph. (We don't know the imports of our imports, so we cannot emit well-formed nodes. They will be reconstructed in the reduce.)

```
**Input**: $Type, (Rank, ImportedTypes[])$
**Output**: $Type, (Rank, ImportedTypes[])$

$portion = Rank/count(ImportedTypes[])$;
for import in ImportedTypes[] do
    emit $import, (portion, null)$;
end
emit $Type, (Rank, ImportedTypes[])$
```

The reduce function then sums each outputted rank by key (the qualified type). This yields updated rank values for all types, and adds in types only imported (not actually sourced on github). These latter types lack outgoing edges, but can still gain reputation. We make sure to find and re-attach the list of imports for each type in order to support further iterations. Each iteration increases the rank of popular Types, meaning that they will increase the rank of their imports, which will increase the rank of their imports, and so on. Currently, I do 10 iterations.

```
**Input**: $Type, Listof(Rank, ImportedTypes[])$
**Output**: $Type, (Rank, ImportedTypes[])$

OutputRanking = $(Type, (0, null))$;
for Ranking in List do
    if Ranking's Imports is not null then
        We have the ranking containing our import;
        so put them into OutputRanking;
    end
    Add Ranking's rank to OutputRanking;
end
emit (Type, OutputRanking);
```

3.3 Aggregate

The 'Aggregate' job is composed of a Map and an Identity reduce. The Map function simply outputs the Rank passed in as the key, and the Type passed in as the value, so that the output can be sorted by Rank rather than Type. The reduce is only needed to force the output from the Map function to be sorted - it just emits any keys and values passed in.

4 Metrics

4.1 Input Data

My input data is 20gb and is composed of almost 14 million .java files from 210 thousand projects.

4.2 Results

Most of the top-ranked results were built-in types, or ubiquitously used types from projects like Apache Log4J, JUnit, and Android. Log4J, JUnit, and Android all have repos on Github, so it makes sense that they rose to the top. The results definitely seem to favour imports that would be used many places in a project, which makes sense given that I did not differentiate projects, rather treating each type as an isolated peer to all other types. It may be necessary to group the ranking project by project in order to dampen this effect.

The three highest ranked types were ArrayList, List, and IOException. It may be best to remove types not actually sourced at Github from evaluation in order to better understand the rankings. Evaluation of these ranking will likely require a number of subjective choices between competing projects, followed by lookups to see which project has a higher index (which the subjective choice will hopefully validate).

About 2 million .java files apparently had no imports, though I think this may merit investigation. This makes sense for compilation units like Enums, however. Also, 600 thousand contained no package statement. Presumably, these use the default package. 150 thousand files failed to parse properly. I suspect encoding issues here. A more tolerant parser might fix this, but further investigation is required to say whether this would be useful.

4.3 Computation Cost

My initial cluster was composed of a name node and five data nodes, each running on a two-core Medium instance on Cybera's cloud. My initial run took 48 minutes, running at most 2 map tasks per machine concurrently. I mistakenly configured it to only run four reduce tasks, though this appears to have fully utilized the machines; both the map tasks and reduce tasks appear to have generally used upwards of 95% of each vm's available CPUs.

I then added five more Medium data nodes (doubling the number of slaves), and rebalanced the data to spread it evenly across the new and old nodes. Without changing anything else, the same job took 26 minutes.

I then configured the number of Reducers to more fully use the new resources. The Hadoop wiki suggests a value of $0.95(maxconcurrentmaps * totalcores)$, so I set the total number of reducers to 19. However, the job again took 26 minutes.