ARBITER Technology and Maturity Scoring

March 2015

Table of Contents

1. INTRODUCTION 1

2. DOCUMENT PROCESSING 2

3. TECHNOLOGY score 4

4. Maturity score 6

# 

# INTRODUCTION

Technology scores and maturity scores are typically generated on a subset of the available data, and often created for a set of domains (computers, molecular biology, mechanical engineering, and so on). In addition, scores are usually created for each year. What this amounts to is that we usually create a corpus of patents for a given year and a given domain and then process this corpus as a unit. The workflow outlined in this document follows that assumption.

The code has been tested on RHEL 6 and Mac OS 10.6 and has the following requirements:

|  |  |
| --- | --- |
| Java version 1.6 or higher |  |
| Python version 2.6 or 2.7 |  |
| xsltproc and xmllint | These typically come standard with Mac OS X or Linux. |
| Mallet | The code uses version 2.0.7 of the Mallet toolkit, available at [http://mallet.cs.umass.edu/](http://mallet.cs.umass.edu/" \t "_blank). Other versions will most likely work as well, but were not tested. You can get the version we use at [http://www.cs.brandeis.edu/~marc/fuse/downloads/tools/](http://www.cs.brandeis.edu/~marc/fuse/downloads/tools/" \t "_blank) |
| Stanford tools | <http://nlp.stanford.edu/> |
| Classifier model | There is a small classifier models for technology scores bundled in with the code, which is sufficient when you just try to see if the code runs, but does not have great performance. Technology classifier models are available at [http://www.cs.brandeis.edu/~marc/fuse/downloads/models/](http://www.cs.brandeis.edu/~marc/fuse/downloads/models/" \t "_blank) |

In some cases, you may have to set some environment variables to avoid encoding issues. The following variables need to be set (using export on a bash shell or setenv on a c shell):

|  |  |
| --- | --- |
| LC\_ALL | en\_US.UTF-8 |
| LANG | en\_US.UTF-8 |

Generating time series for technology and maturity scores is a process that includes various steps: selecting corpora and performing all needed document-level pre-processing, generating technology scores and maturity scores for all corpora, and creating the time series.

# DOCUMENT PROCESSING

The code described in this section collects documents into a corpus and then processes these documents. The documents can be US or Chinese patents from LexisNexis or Web of Science abstracts. Processing includes document structure parsing (which also takes care of issues specific to a document source), sentence splitting, tokenization, tagging, term extraction and feature extraction. This phase is a requirement for subsequent processing.

The input is a list with file path specifications where each line contains two or three tab-separated fields: a year, a file path and an optional shortened path. An example is printed below.

1980 data/patents/xml/us/1980/12.xml 1980/12.xml   
1980 data/patents/xml/us/1980/13.xml 1980/13.xml   
1980 data/patents/xml/us/1980/14.xml   
0000 data/patents/xml/us/1980/15.xml

The file path points to the actual location of the file to be processed for this corpus. The shortened path can be used to specify a local path in the corpus, if it is not given, the local path will be the entire file path. So with the four lines above, we will have a short local path for two of the files. The year can be given some imaginary value if it is not known or if it does not matter for current processing.

There are two ways to create and process a corpus: one more suited for smaller corpora (up to a couple of thousand files) and one suited for larger corpora. They are equivalent in the sense that the results are identical when the two approaches are applied to the same list of files. In both cases, the top level code is in ontology/doc\_processing.

The simplest way is to use the main.py script, which requires as input a file list and an output directory (using the -c and -f options respectively, or --filelist and --corpus when using their long forms):

python main.py \  
 -f data/lists/sample-us.txt \  
 -c data/sample-us

Note that the command is spread out over several lines for explanatory considerations and for space reasons. There are two more options. The language defaults to English, but to specify that the language is Chines use "-l cn". To use verbose messages (basically printing filenames when they are processed), use -v or --verbose. See the documentation string in main.py for more information.

The invocation above will look for the Stanford tools at a couple of places, which are quite specific to the local setup used by code developers. Most users may need to use two extra options for specifying the location of the Stanford tagger and segmenter:

--stanford-tagger-dir PATH   
--stanford-segmenter-dir PATH

The paths point to the root of the installation of these tools, that is, the directory that includes the bin directory.

The main limitation with the one-size-fits-all main.py script is that processing time can get rather high for large corpora, for example, processing 40,000 US patents takes 1-2 days on a high-end desktop. The process can easily be parallelized by splitting the corpus in smaller chunks, but in that case some extra bookkeeping is needed, especially to prepare for subsequent processing. In addition, the script has been proven to a bit brittle at times. There is error trapping at the document processing level, but there are some ill-understood errors that are known to let the tagger hang at times.

As an alternative way to create and process a corpus, there is a slightly more complicated series of batch processing steps that can be taken. To achieve the same results as with the main.py script we would do the following

python step1\_init.py \  
 -f data/lists/sample-us.txt \  
 -c data/corpora/sample-us   
python step2\_process.py -c data/sample-us -n 4 --populate   
python step2\_process.py -c data/sample-us -n 4 --xml2txt   
python step2\_process.py -c data/sample-us -n 4 --txt2tag   
python step2\_process.py -c data/sample-us -n 4 --tag2chk

The main difference here, apart from using five commands instead of one, is that main.py processes all documents in the list but for these batch scripts the number of files to be processed has to be given (it defaults to processing one document). This allows for some more flexibility in how you want to process a corpus. See the documentation string of the two batch scripts for more information. The resulting corpus is almost identical to the one created with main.py, barring some configuration settings like timestamp and the initialization command used.

**Corpus Structure.**

The result of the above processing is a directory data/sample-us, in which the corpus is initialized and populated with the files in data/lists/sample-us.txt. The directory structure is as follows (only the relevant parts are shown):

|-- config   
| |-- files.txt  
| |-- general.txt   
| `-- pipeline-default.txt `-- data  
 |-- d0\_xml 'import of XML data'  
 |-- d1\_txt 'results of document structure parser'  
 |-- d2\_seg 'segmenter results'  
 |-- d2\_tag 'tagger results '  
 |-- d3\_phr\_feats 'results from feature extraction'  
 `-- workspace 'work space area'

The script copied the file list to config/files.txt so there is a local copy of the list with all input files. Processing results are in d0\_xml, d1\_txt, d2\_seg (Chinese only), d2\_tag and d3\_phr\_feats. The structure of those directories are mirrors of each other and look as follows:

`-- 01  
 |-- state  
 | |-- processed.txt  
 | `-- processing-history.txt  
 |-- config  
 | |-- pipeline-head.txt  
 | `-- pipeline-trace.txt  
 `-- files  
 |-- 1980  
 | | US4192770A.xml.gz  
 | ` US4236596A.xml.gz  
 `-- 1981  
 | US4246708A.xml.gz  
 ` US4254395A.xml.gz

All files are compressed. The first part of the directory tree is a run identifier, usually always '01' unless the corpus was processed in different ways (using different chunker rules for example). The structure under the files directory is determined by the third column in the file list.

# TECHNOLOGY score

The technology classifier takes feature files created by the document processing code and generates technology scores for all terms in those documents. It runs off a corpus, generates a technology score for each term in each document, and then summarizes the score over the corpus.

The input to the classifier is taken from the feature files in a corpus that are the endpoint of the document level processing. Input can be given as a complete corpus (which is what the scripts mention above generate) or as a list of files. In the former case the code picks out the right files from the corpus. The input files have lines in the following format (all fields are tab-separated):

term\_id year term feature+

All fields, including the features, are tab-separated. The term\_id is the name of an input file followed by an underscore and a number. Each feature has a name and a value, for example, next2\_tags=IN\_NN.

The top-level script for running the technology classifier is

ontology/classifier/run\_tclassify.py

The classifier needs to be run from the directory it is in.To run the classifier on a corpus, you can do something like the following.

python run\_tclassify.py  
 --classify  
 --corpus ../doc\_processing/data/patents/corpora/sample-us  
 --model data/models/technologies-010-20140911/train.model  
 --output data/classifications/sample-us

Change the --corpus and option depending on where your corpus lives. The corpus used in this example is the sample corpus included in the distribution. Use the --output option to specify what directory classification results are written to. The --model option here uses the model that is shipped with the code. This model is sufficient when you just try to see if the code runs, but eventually you will want to get a better model, as noted in the introduction. New models can be created by following directions in the readme file in the classifier directory.

The command above assumes that Mallet is installed in one of the default spots, which typically isn't the case because the default spots are quite indiosyncratic. So you probably need to add the --mallet-dir option to hand in your local Mallet bin directory:

python run\_tclassify.py  
 --classify  
 --corpus ../doc\_processing/data/patents/corpora/sample-us  
 --model data/models/technologies-010-20140911/train.model  
 --output data/classifications/sample-us  
 --mallet-dir /tools/mallet/mallet-2.0.7/bin

More details on running the classifier are in run\_tclassify.py.

The classification results for the files in the corpus or file list are concatenated and put in the output directory, which has the following content:

classify.MaxEnt.out.gz   
classify.MaxEnt.out.s1.all\_scores.gz   
classify.MaxEnt.out.s2.y\_scores   
classify.MaxEnt.out.s3.scores.sum   
classify.MaxEnt.out.s4.scores.sum.az   
classify.MaxEnt.out.s4.scores.sum.nr   
classify.MaxEnt.stderr   
classify.info.filelist.txt   
classify.info.general.txt   
classify.mallet.gz

The last file is the file with the input to the classifier. The first file contains the raw results of the classifier and the second just the yes scores from the raw results. The results have scores not for each term occurrence but for each term in a document. The file that one is most likely to use is classify.MaxEnt.out.s3.scores.sum, which has lines as follows:

senses flag 0.439934 1 0.439934 0.439934  
text message 0.342870 2 0.225453 0.460288

The columns contain: the term, the technology score over the entire corpus or list of files, the number of documents in the corpus or file list that the term occurs in, the lowest score, and the highest score.

Time series for technology scores can be created by (i) processing corpora for a set of years, (2) generating the technology classification for that corpus, and (3) taking the output of the classifier on each corpus as saved in the classify.MaxEnt.out.s4.scores.sum.az file and grabbing the first two columns using the cut command:

cut –f1,2 data/classifications/sample-us > tscores-1980.txt

Assuming that the sample test corpus has document all from one year, this will give the technology scores for that year.

# Maturity score

To create time series for maturity scores, three steps are required: (i) running the pattern matcher that collects evidence of usage for a term, (ii) calculate the usage rate, and (iii) create the time series.

Calculating the maturity score starts with running the pattern matcher, using the maturity patterns (more properly called “usage patterns”). The input to this component is the same as the input to the technology classifier, that is, it takes the feature files from a corpus.The top-level script for running the pattern matcher is

ontology/matcher/run\_matcher.py

To run the matcher on all files of a corpus, do the following (new lines and spaces added for clarity, the command should be on one line):

python run\_matcher.py \  
 --corpus ../doc\_processing/data/sample-us \  
 --output maturity

This runs the matcher on the corpus specified, which is the corpus created by the document-level processing, and puts the results in a directory data/o2\_matcher/maturity inside the corpus. There is a --patterns option with possible values MATURITY and PROMISE, but it does not need be used here since the value defaults to the former. Another option of note is the –language option with values “en” and “cn”, where the first is the default. Using “cn” ensures that the Chinese pattern set will be used. See run\_matcher.py for more command line options.

The pattern matcher creates six output files:

match.info.config.txt   
match.info.features.txt   
match.info.filelist.txt   
match.info.general.txt   
match.results.full.txt   
match.results.summ.txt

The first four contains various pieces of information, including a list of all features found, a copy of the file list, and a file with the processing parameters. The file match.results.full.txt contains the raw output of the matcher:

1980 US4192770A.xml\_71 maturity-provide process prev\_V=provides   
1980 US4192770A.xml\_142 maturity-have boiling point prev\_V=have   
1980 US4192770A.xml\_158 maturity-activate alumina prev\_V=activated   
1980 US4192770A.xml\_165 maturity-provide metallic ions prev\_V=provide   
1980 US4192770A.xml\_583 maturity-have affect prev\_V=has   
1980 US4192770A.xml\_646 maturity-have effect prev\_V=have

The third column has the name of the pattern that matched, the fourth the term for which we had a match, and the fifth the actual value of the feature that matched.

Finally, match.results.summ.txt contains a summary with match counts for each term:

1 member  
 1 metallic ions  
 2 path  
 1 permanent magnet  
 1 polarity  
 3 poles   
 1 positionable control element

This file, like the previous file, is tab-separated, but note that there can be leading spaces (the count is right adjusted for the convenience of the person opening the file and inspecting its contents).

The next step is to calculate usage rates from the results of the pattern matcher. The usage rate is a number between 0 and 1 that is calculated as follows:

Most terms have zero scores, which corresponds to no matches found (that is, no evidence of usage found). The closer the number gets to 1, the closer the term usage is relative to the term with the most matches.

To calculate usage rates we use the script in

ontology/maturity/collect\_usage\_data.py

which combines the match results and the results of the technology classifier, where the technology classifier, apart from the technology scores, also provides frequency data needed for calculating the usage rate. Assuming matcher results and technology scores as created with the example commands above, we create the usage scores as follows:

python collect\_usage\_data.py \  
 --corpus ../doc\_processing/data/sample-us \  
 --tscores ../classifier/data/classifications/sample-us \  
 --output data/usage-sample-us.txt \  
 --language en

In the output, terms are presented with four numbers as follows:

0.0029 0.0000 2 0 accelerometer senses  
1.0000 0.0000 1 0 accelerometer sensor  
0.0555 0.1201 5 2 accelerometer signal  
0.9189 0.0757 1 1 accelerometer signal accel  
0.9656 0.0757 1 1 accelerometer signal trace

The first column has the average technology score for the term in the corpus, the second the usage rate, the third the number of documents the term occurs in, and the fourth the number of matches for the term.

The last step is creating the time series. This is slightly more involved as with the technology scores, where we could simply take a set of classification results, each for a particular year, and grab the scores for that year. Maturity scores are different in several ways:

1. Maturity scores tend to be meaningless below a certain term frequency threshold, we stipulate that a term has to occur in at least 25 documents, but this is a bit arbitrary.
2. Maturity scores do not rise and fall like the technology scores can, and sometimes do. Rather, we assume that maturity scores do not fall in lockstep with the usage rates and that there is a delay.
3. Usage scores do not tend to carry great meaning for terms that are not technologies. For example, the term “caution” can have a high maturity score because it is often found on the context “use caution”, which triggers one of the usage patterns.

When creating the maturity score time series, we enforce the first two, but not the last. Now, let’s assume we have four files with usage scores, from the years 2010-2013:

usage-2010.txt  
usage-2011.txt  
usage-2012.txt  
usage-2013.txt

We can create a file with terms with a document frequency of 25 or higher with the count\_terms.py script (also in ontology/maturity):

python count\_terms.py usage-201\*.txt

This script will write its output to terms-0025.txt. This is a memory intensive step. With a few dozen corpora, each with 20-50K files, the machine running this likely needs 16GB of memory.

Finally, the create\_time\_series.py script takes the usage files and the file with frequent terms and creates the time series:

python create\_time\_series.py –t terms-0025.txt usage-201\*.txt

The script creates two output files, one with a maturity time series based on usage rates and one with maturity time series based on raw frequencies. The first of these should be much more useful, the second file is created because an early version of the system did not generate enough matches to create scores based on usage rates. The names of the output files are hard-coded and include a timestamp, for example:

out-20141120:195652-frequency-based.txt  
out-20141120:195652-usage-based.txt

The first file has scores as given in FUSE phase 1, where each term-year pair got a 0, 1, or 2 (unavailable, immature, mature). These scores are based on frequency counts only and are a fallback. The second file has a score between 0 and 1 for each term-year. These scores are based on the usage rates. The content of the second file looks as follows:

0.0228 0.0182 0.0146 0.0117 abrasive fluid  
0.0179 0.0143 0.0115 0.0092 abrasive fluids  
0.0572 0.0458 0.0706 0.0565 abrasive force  
0.0748 0.0598 0.0479 0.0383 abrasive forces  
0.1145 0.1155 0.2240 0.2754 abrasive grain  
0.3196 0.2934 0.2571 0.2206 abrasive grains  
0.1329 0.1063 0.0850 0.0680 abrasive grit

The top of the file has a row with headers, which contain the years. These headers are calculated from the file names, so if the file names do not have years in them (a sequence of four digits) then the year will not be printed as a header.