Discovering Affixes Automatically

COMPUTATIONAL MORPHOLOGY AND SYNTAX

Luke Ellul
42498M | ARI2571

Contents

Contents

Structure of Trie

The functions that operate the trie data structure are found in the directory **src** which are **node.js** and **trie.js**.

node.js contains functions that deal with the node itself. The function **newNode** creates a new node. A **node** has a character, a Boolean specifying whether the node contains a character which signifies the end of a word, an object that holds properties such as the suffix score and the prefix score, and an object containing a mapping of the characters which are children to the node and their respective nodes.

The file **trie.js** contains functions that deal with the trie data structure. The function **createBranch** adds new nodes to a node's children. The function **insertWords** takes a list of words and adds them to a trie. The function **insertWords** can also add a list of words to an already existing trie. The function **containsWord** checks if a word occurs within a trie.

trie.js also has two important functions: **store** and **restore**. **store** takes a trie and converts it to a JavaScript object which can then be converted to JSON and stored on disk. **restore** takes a JavaScript object and converts it into a working trie. Restoring a trie from disk is much faster than regenerating the trie from a wordlist thus with **store** and **restore** we can save a lot of time.

Running an example

The file **trieExample.js** located in the **example** directory contains some examples which make use of the previously mentioned functions. First, the script adds 10 words to a new trie then checks if each of those words is recognized by the trie. The script then outputs an array of 10 strings each with the word “yes” signaling that each word was recognized by the trie.

Then, the script checks if 10 other words which are not in the trie are recognized by the trie. The script outputs an array of 10 strings each with the word “no” signaling that the trie didn’t recognize one word.

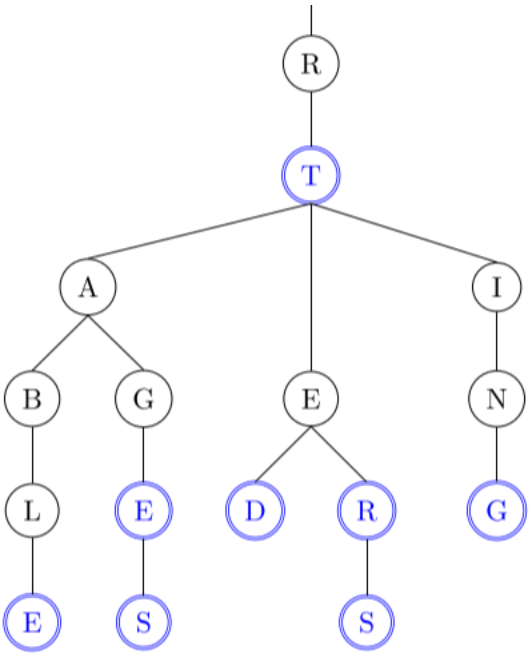
The script then loads a word list, **list1.txt** found in the directory **wordlists** which contains 7776 words and adds them to a new trie. The trie is then stored inside a file called **trie1.json** under the directory **tries**. On my computer (8GB RAM, Corei5), the script took around 4 seconds to generate the trie. Only one core was used since JavaScript is single-threaded.

The script then loads the file **trie1.json** and restores it into a working trie. This was accomplished in less than a second. The script checks if the words **abdomen** and **ejisgfhv** occur in the restored trie giving a **yes** and a **no** respectively. This was also accomplished instantly.

Each word from the word list was assigned a token count of 1.

Finding Affixes

The files **morphemes.js** and **boundary.js** deal with discovering suffixes and prefixes. The function **getGoldenChildren** in **morphemes.js** gets the nodes returns the nodes representing characters that end a word that are under the inputted node. For example:



If we input the node representing the character “R” in the function **getGoldenChildren**, the function returns the nodes representing the characters that are surrounded with a blue outline in the above picture. The function **getGoldenCount**, also in **morphemes.js**, makes use of **getGoldenChildren** to get the sum of all the counts of the golden children of a node. This way, we can get the counts of all the words that start with “reporte” for example.

The function **getSuffixes** in **trie.js** takes a Node as input and returns an array of all the suffixes under that node. For example, consider the following list:

```
//list of words and their counts
const list = {
    report: 3900,
    reporter: 241,
    reporters: 82,
    reported: 609,
    reportable: 5,
    reportage: 63,
```

```
    able: 5900
};
```

If we feed the word representing “r” into the function like so:

```
getSuffixes('', getNodeChildren(newRoot)['r'])
```

we get the following output:

```
[ 'eporters', 'eported', 'eportable', 'eportage' ]
```

The function **getRealSuffixes** in **morphemes.js** works similarly to **getSuffixes** but instead of returning an array of strings of all the suffixes, it returns an array of nodes that have a high suffix score. So using the above list once again, we’re going to call **getRealSuffixes** to get an array of nodes that have a high suffix score from the words that start with “r”. Then for each of the returned nodes that have a high suffix score, we will call **getSuffixes** to get a list of strings that build up that suffix. The following code demonstrates this:

```
getRealSuffixes(getNodeChildren(updatedSuffixes)['r'])
.map(node => getSuffixes('')(node))
```

Since the Node representing the character “t”, which signals the end of the word “report” is the only node with a high suffix score in the above list, the function returns the suffix strings that originate from “report”:

```
[ [ 'ers', 'ed', 'able', 'age' ] ]
```

Boundaries

The file **boundary.js** deals with testing boundaries between characters in a word in order to discover suffixes and prefixes. The functions **Test1**, **Test2** and **Test3** take a suffix and check whether it's sufficient using the same methods described in the paper and the assignment spec. The functions **PrefixTest2** and **PrefixTest3** do the same tests as **Test2** and **Test3** but for prefixes instead of suffixes.

The function **updateSuffixBoundaries** updates the suffix scores under a node by iterating over the nodes of all children and their children's nodes and for each node it calculates the suffix score. This way, the suffix score is stored in the trie so a separate data structure for storing the suffix scores is not required.

The function **updatePrefixBoundaries** does the same thing as **updateSuffixBoundaries** but instead computes the scores for prefixes.

The functions **boundarySuffixCheckLog** and **boundaryPrefixCheckLog** work similartly to the above functions but with every iteration, they log the current suffix or prefix being processed and the scores being tested.

The file **index.js** contains functions that test the above functions and provide an output like that of the assignment spec. We're going to use the list used above as an example. We first create a root node. A root node is the same as other nodes except it stores the character " " representing a blank space. The root node contains 26 children as nodes. Each child represents a letter in the English alphabet.

First, we start by logging an iterative suffix boundary check for the word "reporters":

```
//iterative suffix boundary check for "reporters"
boundarySuffixCheckLog(' reporters', '', newRoot, newRoot);
```

The function logs the following information:

```
Boundary check for:  -reporters
Test 1 for   is False
Test 2: Freq of alphaA / Freq of alpha = is it approx equal to 1?
alphaA: 10800 / alpha: 0 = True
Test 3: Freq of alphaAB / Freq of alphaA = is it much less than 1?
alphaAB : 4900 / alphaA: 10800 = True
```

```
Boundary check for:  r-eporters
Test 1 for  r is False
Test 2: Freq of alphaA / Freq of alpha = is it approx equal to 1?
alphaA: 4900 / alpha: 10800 = False
Test 3: Freq of alphaAB / Freq of alphaA = is it much less than 1?
alphaAB : 4900 / alphaA: 4900 = False


Boundary check for:  re-porters
Test 1 for  re is False
Test 2: Freq of alphaA / Freq of alpha = is it approx equal to 1?
alphaA: 4900 / alpha: 4900 = True
Test 3: Freq of alphaAB / Freq of alphaA = is it much less than 1?
alphaAB : 4900 / alphaA: 4900 = False


Boundary check for:  rep-orters
Test 1 for  rep is False
Test 2: Freq of alphaA / Freq of alpha = is it approx equal to 1?
alphaA: 4900 / alpha: 4900 = True
Test 3: Freq of alphaAB / Freq of alphaA = is it much less than 1?
alphaAB : 4900 / alphaA: 4900 = False


Boundary check for:  repo-rters
Test 1 for  repo is False
Test 2: Freq of alphaA / Freq of alpha = is it approx equal to 1?
alphaA: 4900 / alpha: 4900 = True
Test 3: Freq of alphaAB / Freq of alphaA = is it much less than 1?
alphaAB : 4900 / alphaA: 4900 = False


Boundary check for:  repor-ters
Test 1 for  repor is False
Test 2: Freq of alphaA / Freq of alpha = is it approx equal to 1?
alphaA: 4900 / alpha: 4900 = True
Test 3: Freq of alphaAB / Freq of alphaA = is it much less than 1?
alphaAB : 4900 / alphaA: 4900 = False


Boundary check for:  report-ers
Test 1 for  report is True
Test 2: Freq of alphaA / Freq of alpha = is it approx equal to 1?
alphaA: 4900 / alpha: 4900 = True
Test 3: Freq of alphaAB / Freq of alphaA = is it much less than 1?
alphaAB : 932 / alphaA: 4900 = True
```

```
Boundary check for:  reporte-rs
Test 1 for  reporte is False
Test 2: Freq of alphaA / Freq of alpha = is it approx equal to 1?
alphaA: 932 / alpha: 4900 = False
Test 3: Freq of alphaAB / Freq of alphaA = is it much less than 1?
alphaAB : 323 / alphaA: 932 = True


Boundary check for:  reporter-s
Test 1 for  reporter is True
Test 2: Freq of alphaA / Freq of alpha = is it approx equal to 1?
alphaA: 323 / alpha: 932 = False
Test 3: Freq of alphaAB / Freq of alphaA = is it much less than 1?
alphaAB : 82 / alphaA: 323 = True
```

The boundary check at report-ers is the only boundary that passed all tests. In fact, when we called the function **getRealSuffixes** earlier, it returned the node at this boundary.

Now, we're going to log an iterative prefix boundary check for the word "reportable":

```
//iterative prefix boundary check for "reportable"
boundaryPrefixCheckLog(' reportable', '', newRoot, newRoot);
```

The function logs the following output:

```
Boundary check:  -reportable
Test 1 for reportable is True
Test 2: Freq of Bbeta / Freq of beta = is it approx equal to 1?
Bbeta: 5 / beta: 0 = True
Test 3: Freq of ABbeta / Freq of Bbeta = is it much less than 1?
ABbeta : 0 / Bbeta: 5 = True


Boundary check:  r-eportable
Test 1 for eportable is False
Test 2: Freq of Bbeta / Freq of beta = is it approx equal to 1?
Bbeta: 0 / beta: 0 = False
Test 3: Freq of ABbeta / Freq of Bbeta = is it much less than 1?
ABbeta : 5 / Bbeta: 0 = False
```

```
Boundary check:  re-portable
Test 1 for portable is False
Test 2: Freq of Bbeta / Freq of beta = is it approx equal to 1?
Bbeta: 0 / beta: 0 = False
Test 3: Freq of ABbeta / Freq of Bbeta = is it much less than 1?
ABbeta : 0 / Bbeta: 0 = False


Boundary check:  rep-ortable
Test 1 for ortable is False
Test 2: Freq of Bbeta / Freq of beta = is it approx equal to 1?
Bbeta: 0 / beta: 0 = False
Test 3: Freq of ABbeta / Freq of Bbeta = is it much less than 1?
ABbeta : 0 / Bbeta: 0 = False


Boundary check:  repo-rtable
Test 1 for rtable is False
Test 2: Freq of Bbeta / Freq of beta = is it approx equal to 1?
Bbeta: 0 / beta: 0 = False
Test 3: Freq of ABbeta / Freq of Bbeta = is it much less than 1?
ABbeta : 0 / Bbeta: 0 = False


Boundary check:  repor-table
Test 1 for table is False
Test 2: Freq of Bbeta / Freq of beta = is it approx equal to 1?
Bbeta: 0 / beta: 5900 = False
Test 3: Freq of ABbeta / Freq of Bbeta = is it much less than 1?
ABbeta : 0 / Bbeta: 0 = False


Boundary check:  report-able
Test 1 for able is True
Test 2: Freq of Bbeta / Freq of beta = is it approx equal to 1?
Bbeta: 5900 / beta: 0 = True
Test 3: Freq of ABbeta / Freq of Bbeta = is it much less than 1?
ABbeta : 0 / Bbeta: 5900 = True


Boundary check:  reporta-ble
Test 1 for ble is False
Test 2: Freq of Bbeta / Freq of beta = is it approx equal to 1?
Bbeta: 0 / beta: 0 = False
Test 3: Freq of ABbeta / Freq of Bbeta = is it much less than 1?
ABbeta : 5900 / Bbeta: 0 = False


Boundary check:  reportab-le
Test 1 for le is False
Test 2: Freq of Bbeta / Freq of beta = is it approx equal to 1?
Bbeta: 0 / beta: 0 = False
Test 3: Freq of ABbeta / Freq of Bbeta = is it much less than 1?
ABbeta : 0 / Bbeta: 0 = False
```

```
Boundary check:  reportabl-e
Test 1 for e is False
Test 2: Freq of Bbeta / Freq of beta = is it approx equal to 1?
Bbeta: 0 / beta: 10800 = False
Test 3: Freq of ABbeta / Freq of Bbeta = is it much less than 1?
ABbeta : 0 / Bbeta: 0 = False
```

The boundary at report-able, is the only boundary that passed all prefix tests. The reason why obviously, is because I added the word “able” to the list.

The above example is stored in the file **index.js** and can be run without any modification. **index.js** also stores a trie of the above word list as a JSON file in **trie.json** located in the **tries** folder. The trie is updated with the computed prefix and suffix scores so they can be easily retrieved without recomputing them.

Using a Real Wordlist

The file **wordlist.json** located under the directory **wordlists** contains an object with 5000 keys as words and their frequency counts taken from here: https://www.wordfrequency.info/. The site provides a free 5000-word sample from the COCA corpus along with the frequency counts.

The file **demo.js** located in the root directory uses this wordlist to automatically discover potential affixes. The script first loads the file, **wordlist.json**, and converts it into a trie called BigTrie. Then it calls the **updateSuffixBoundaries** function to update the suffix scores in the trie and the **updatePrefixBoundaries** function to update the prefix scores.

Then, it uses the **store** function to store the updated trie inside a JSON file called **BigTrie.json**. The file is saved under the directory **tries** located in the root directory.

The script then, attempts to restore the trie from **BigTrie.json** using the **restore** function. Then it calls the **getRealSuffixes** function on the restored trie and logs an array of the potential suffixes that were computed and discovered. Of course, we could have used the original trie instead of the restored one, but I wanted to show that a restored trie functions just as well as an immediately generated trie. Restoring from an already generated trie of course prevents the hassle of regenerating the suffix and prefix scores. Here's a sample of the output that show the discovered suffixes:

```
suffixes found in Restored BigTrie:
[ 'ish',
  '-American',
  'ic',
  's',
  'ly',
  'back',
  'ionnaire',
  'naire',
  'ity',
  't',
  'ist',
  'ism',
  'ice',
  'ify',
  'n',
  'friend',
  'ed',
  'ing',
  'y',
  'ment',
  'or',
  'est',
  'ow',
  'th',
  'rself',
  'ngster',
  'ster',
  'ty',
  'standing',
  'mine',
  'take',
  'lying',
  'go',
  'graduate',
```

The above process can simply be run by running the **demo.js** script as is. Depending on the processor speed it might take a few seconds, so please be patient 😊

Technicalities

Node.js (server-side JavaScript) was used to write the trie structure. Looking at the code, you may notice there aren't any classes, inheritance and things like that. This is because I used a functional approach rather than the traditional OOP style programming. The script makes use of a lot of higher order functions (functions that accept functions as parameters), functional composition and recursion instead of traditional loops. Each function has a Haskell style description over it detailing the types of parameters it accepts and what it returns.

To run this program Node.js 8 or later is required as well as npm (Node Package manager) in order to install the required dependencies. Node.js and NPM can both be installed from https://nodejs.org/en/ for any platform. After installing simply issue the command **npm install** in the root directory of the program to install the required dependencies. Then issue **node index.js**, also in the root directory to run the previously mentioned examples.

The **node** command can be used to run any script.

The packages **Ramda**[1] and **ramda-fantasy**[2] were used to help in accomplishing functional programming techniques.

[1] http://ramdajs.com/
[2] https://github.com/ramda/ramda-fantasy