Spellchecker (version NAD-SSE)

Kaeser Md. Sabrin
iSearch Team
SDSL
Dated: 07.04.2009

Introduction
Spellchecker for iSearch-NAD-SSE is in fact a full fledge query analyzer capable of performing a list of challenging tasks other than mere spellchecking. Initially build as a simple spell checking module, it now has the duty of front end query analyzing and generating PARSED, SPELLCHECKED, ANALYZED data to be fed into the Geo-ontology and Core-Indexer module. Let’s delve into the inside of this multi-tasking module.

Basic Capabilities
Spellchecker has the following query analyzing capabilities:

[01] Basic Spellchecking
   Input: shoman st
   Output: schoman street

[02] Multiple Proposals
   Input: schoman st
   Output: schoman street
   schoeman street
   schuman street
   schorman street
   Comment: outputs are sorted in decreasing similarity scores. Only outputs that exceeds a threshold matching score are generated

[03] Front Side Parsing
   The input query is parsed, unnecessary characters are removed, the query string is tokenized and sent to basic spell checking module.

[04] Phrase Searching
   Input: 1085 schoeman st, hatfield, western cape
   Output: 1085
   schoeman street
   hatfield
   western cape
[05] **NAD Level Detection**
   **Input:** 1085 schoeman st, hatfield, pretoria, gauteng
   **Output:** Number: 1085
   Street: schoeman street
   Suburb: hatfield
   Town: pretoria
   Province: gauteng

[06] **Auto-completion of Incomplete Query**
   **Input:** november state
   **Output:** november 2nd street
   november street
   free state

[07] **African Alternative Names Transformation**
   **Input:** csd noord tranvaal
   **Output:** cape town
   limpopo

[08] **Suffix Removal, Street Indicator Cleaning**
   **Input:** schoemanst
   **Output:** schoeman street
   **Input:** elizabeth ave
   **Output:** elizabeth avenue

[09] **African/English Numbers to Numerical Conversion**
   **Input:** twenty second st
   **Output:** 22nd street
   **Input:** een honderdste rd
   **Output:** 100th road

[10] (a) Only valid outputs are generated (those belonging to database)
    (b) Phrase searching is recursive covering all proposals for a token
    (c) Auto-completion is not performed for a very frequent word (street for example)
    (d) If some token belongs to multiple entity, all are reported (tokoza: suburb, town)
**Usage**

The communicator class is **QueryManager Class**

You first declare an `QueryManager` object, **QueryManager qMan**;

There are only 2 functions via which you can communicate with the QueryManager object.

```c
void qMan.preprocess( bool build, char[] path );
```

**bool build**, is a boolean variable, which must be true for the first time this object is used, (manipulated via index-builder), and false otherwise. Since the building of knowledge base for the spellchecker is time consuming, only the first time the entire knowledge base is created and dumped into some files. For later occasions, the knowledge base is readily read (not built) from the files.

**Char path**, is complete path of a folder, where initial files for spellchecker are kept, as well as files created by the spellchecker.

```c
void qMan.processQuery( char* query,
                        vector <string> &nSp, vector <string> Sp[],
                        vector <string> &unknownNumber, bool &spellCheck);
```

**char* query**, is the user query

**vector <string> &nSp**, is for returning non spatial (non geo-data) part of the user query

**vector <string> Sp[],** is for returning geo-data, divided into levels [16 at most] where each level contains zero or more data. Here levels represent street, province etc. level-0 is smallest level (st) where level-n is highest level (province).

**vector <string> &unknownNumber**, returns some numerical data not found within the database

**bool &spellcheck**, returns true if spellchecking was required, false if used data was correct

The entire spellchecker module is thread-safe, meaning, you can call the same global **QueryManager** object **qMan** from multiple threads, and get analyzed user query safely.
# Dependencies

The spellchecker module requires some files to start work with.

The following files are generated for every new NAD database by the index builder. The folder path containing these files are given to spellchecker in it’s preprocess() function.

- **levelinfo.txt**
  - containing a single integer indicating number of NAD levels
- **level0.txt, level1.txt, ..., leveln.txt**
  - these n test files contains data generated by the index builder for spellchecking
- **nonSpatial.txt**
  - any non-geo data that used query may contain and required for spellchecking

The spellchecker module creates the following files inside the same folder,

- A.info
- B.info
- C.info
- D.info
- E.info
- X.info
- Y.info

where, [A-E] files contain dumped Trie Tree (of different connotations) in pre-order traversal format for spellchecking purpose, [X-Y] contains dumped Red-Black Tree in pre-order format containing information needed for phrase search, auto-completion, level identification etc. The files are text files where data are separated by CR/LF.

The following static files have been generated by the spellchecker module, for its own usage, these files are kept inside “initSpell” folder

- Abbr.txt
- Afro.txt
- Alternatives.txt
- Frequentword.txt
- Numbers.txt
- Roads.txt
- level0Alter.txt
- level1Alter.txt
- level2Alter.txt
- level3Alter.txt
Algorithmic Details

Let us explain the spell-checking algorithm with respect to an example. We will also shed some light on file storage.

Initially we have a dictionary. Now let there is a dictionary word, “honesty”. Let us see how we arm ourselves to identify this word in face of a spell error. There are some basic assumptions about the kind of error a user can make or how the user behaves. They are,

1. If user has no idea about the word, he would probably try to type how the word sounds. This gives us the starting point to work with Double-Metaphone sound algorithm, a celebrated algorithm, which characterizes a word by its pronunciation.
2. A user may make error in 1 or 2 letters in his typing. Research shows that, over 90% of typing errors fall within the range of maximum 2 character error, most of them being a vowel, as consonants are very strongly pronounced.
3. A correct word should be identified immediately, without spending any computation time.

With those in mind, let’s see what to be done with “honesty”. We keep the word in it’s entirety in a trie, and run a straight check if the word is valid at first step, if it is, we just pass it. This first trie is dumped into, “A.info” file with a pair of line indicating key and value, [the value is not important here].

Now, we run double-metaphone on “honesty”, and get a sound string. We keep this string in “B.info”, as key-value pair, where, key is double-metaphone and value is a large string consisting of word-index separated by “|”. Note that we keep word index instead of words themselves in order to save memory as well as increase efficiency. All these words have a similar sound like “honesty”.

The files “C.info” keeps the word “honesty” with one missing character at a time, that is we keep, “hnesty”, “hoesty”, “honsty”, “honety”, “honesy”, “honest” etc. In “D.info” we keep the double-metaphone of the previous mentioned word. Both these files keep pair of lines for each node of the trie, key and value [string index as before]. We mention again that a same double-metaphone could the root of a number of words, especially when there are missing characters.

In file “E.info” we keep (index-string) pair for reverse string finding, that is, when we are in need of a string whose index is ‘k’, we retrieve it from “E.info”. Note that all the files are loaded into Memory at program initiation. Finding using index is very fast, since index has only 10 character [0-9] creating a very flat trie for fast access.

Other two files “X.info” and “Y.info” are required for phrase search and NAD level identification. For phrase search we need the entire string, not words, so keep the entire string in Y and in X we keep the with each string the identifying level. These files are also organized as (key-value) pairs.
Now suppose a user types, “honsty”. We do the following,

01. “honsty” is not found in the dictionary [A.info]. So we need to spell correct it.
02. We find the double-metaphone of “honsty”. Suppose that is “d0”. Using “d0” we pick up all the words having double-metaphone “d0” from double-metaphone trie [B.info] and store for later computation.
03. We find a single character mismatch search using “honsty” and pick those that match. We also run an extra miss search from query to figure out the 2 missed character search. That is we look for “hnsty”, “hosty”, “honty”, “honsy” etc. in both single miss trie [C.info], and their double-methphone of single-miss double-metaphone trie [D.info].
04. Thus we have accumulated a set of probable suggestion words from extracted indices [E.info].
05. Next we run some heuristic scoring operations on these words. We run Dameru-Levenstein String Matching algorithm on these words with compared to user query. We multiplicatively add, some heuristic score based on which trie the suggestion have come from and number of missed character in used for calculation.
06. We rank our suggestions based on the scores and propose a set of solutions crossing a similarity threshold.

Apart from these there are quite few minute heuristics and tricks to make the result fast and accurate.

For phrase search we combine the proposals iteratively and extract the best matches from [Y.info]. Level identification is straightforward once we have figured out the phrases [X.info].

To make it clear again, in all [A-E], we have just dumped the tries into the files where each pair of line is a node of the trie as [key-value]. This is done so that we need not re-calculate the trie again, we just reload the trie into Memory directly.

That’s in a nutshell the working principle of basic spell-checker.
Module Interaction

Core Indexer

Index Builder

SPELLCHECKER

QueryManager

Driver

SearchDictionary

Levenstein

PreProcess

WordData

HashTable

Trie

DoubleMetaphone

MyHeader