Application Developer's Guide of *Language Identification*

Jun Jiang

February 8, 2010

Contents

1 How to build and link with the library **2**

2 How to run the demo **3**

3 How to build the models **5**

4 Interface description **6**

5 How to use the interface **7**

5.1 How to get string representation of identification result 8

5.2 How to tokenize sentence 8

5.3 How to print out language region 9

5.4 How to configure the identification process 9

5.4.1 OPTION_TYPE_LIMIT_ANALYZE_SIZE 10

5.4.2 OPTION_TYPE_BLOCK_SIZE_THRESHOLD 10

5.4.3 OPTION_TYPE_NO_CHINESE_TRADITIONAL . . . 11

Chapter 1

How to build and link with the library

CMake[1] is a prerequisite as the build system. The system could be built using script build.sh in directory build like below.

```
$ cd build
$ ./build.sh
```

After the project is built, the library target liblangid.a is created in directory lib, and the executables in directory bin are created for demo and test.

To link with the library, the user application needs to include header files in directory include, and link the library files liblangid.a in directory lib.

An example of compiling user application test.cpp looks like:

```
$ export LANGID_PATH=path_of_langid_project
$ g++ -I$LANGID_PATH/include -o test test.cpp $LANGID_PATH/lib/liblangid.a
```

[1] http://www.cmake.org

Chapter 2

How to run the demo

To run the demo bin/test_langid_run, please make sure the project is built (see 1).

Below is the demo usage:

```
$ cd bin
$ ./test_langid_run -t encoding [-f INPUT_FILE]
$ ./test_langid_run -t language [-f INPUT_FILE]
$ ./test_langid_run -t list [-f INPUT_FILE]
$ ./test_langid_run -t segment [-f INPUT_FILE]
$ ./test_langid_run -t sentence [-f INPUT_FILE]
```

In above demo usage, the INPUT_FILE could be set as an input file to analyze. If this option -f is not set, it would analyze each line from standard input and print its result. Below describes each demo usage in detail.

Demo of identifying character encoding of an input file.

```
$ ./test_langid_run -t encoding -f INPUT_FILE
```

Demo of identifying single primary language from standard input in UTF-8 encoding.

```
$ ./test_langid_run -t language
```

Demo of identifying a list of multiple languages of an input file in UTF-8 encoding.

```
$ ./test_langid_run -t list -f INPUT_FILE
```

Demo of segmenting a multi-lingual input file in UTF-8 encoding into single-language regions.

```
$ ./test_langid_run -t segment -f INPUT_FILE
```

Demo of sentence tokenization for an input file in UTF-8 encoding.

```
$ ./test_langid_run -t sentence -f INPUT_FILE
```

Chapter 3

How to build the models

In below Knowledge interface, it is necessary to load the encoding model and language model in binary format.

```
virtual bool loadEncodingModel(const char* fileName) = 0;
virtual bool loadLanguageModel(const char* fileName) = 0;
```

These two models have been built in directory db/model/ beforehand. In the case of supporting other platforms, you might need to build the binary models by yourself. Then you could run the script langid_build_model.sh in directory bin like below.

```
$ cd bin
$ ./build_model.sh
```

Now the model files encoding.bin and language.bin are created in directory bin, then you could load them using above Knowledge interface.

Chapter 4

Interface description

The C++ API of the library is described below:

Class Factory creates instances for identification, its methods below create instances of Factory, Analyzer and Knowledge.

```
static Factory* instance();
virtual Analyzer* createAnalyzer() = 0;
virtual Knowledge* createKnowledge() = 0;
```

Class Knowledge manages the linguistic information, its methods below load the encoding model and language model.

```
virtual bool loadEncodingModel(const char* fileName) = 0;
virtual bool loadLanguageModel(const char* fileName) = 0;
```

Class Analyzer executes the identification, its methods below could identify encodings, identify languages in UTF-8 text, and also segment the UTF-8 multi-lingual document into single-language regions. Analyzer::sentenceLength() could be used for sentence tokenization in UTF-8 text.

```
virtual void setKnowledge(Knowledge* pKnowledge) = 0;
void setOption(OptionType nOption, int nValue);

virtual bool encodingFromString(const char* str, EncodingID& id) = 0;
virtual bool encodingFromFile(const char* fileName, EncodingID& id) = 0;

virtual bool languageFromString(const char* str, LanguageID& id) = 0;
virtual bool languageFromFile(const char* fileName, LanguageID& id) = 0;
virtual bool languageListFromString(const char* str, std::vector<LanguageID>& idVec) = 0;
virtual bool languageListFromFile(const char* fileName, std::vector<LanguageID>& idVec) = 0;
virtual bool segmentString(const char* str, std::vector<LanguageRegion>& regionVec) = 0;
virtual bool segmentFile(const char* fileName, std::vector<LanguageRegion>& regionVec) = 0;

virtual int sentenceLength(const char* str) = 0;
```

Chapter 5

How to use the interface

In general, the interface could be used following below steps:

1. Include the header files in directory include.

```
#include "langid/language_id.h"
#include "langid/factory.h"
#include "langid/knowledge.h"
#include "langid/analyzer.h"
```

2. Use the library name space.

```
using namespace langid;
```

3. Call the interface and handle the result.

In the example below, the return value of some functions are not handled for simplicity. In your using, please properly handle those return values in case of failure. The interface details could be available either in document docs/html/annotated.html, or the comments in the header files of directory include/langid.

```
// create instances
Factory* factory = Factory::instance();
Analyzer* analyzer = factory->createAnalyzer();
Knowledge* knowledge = factory->createKnowledge();

// load encoding model for encoding identification
knowledge->loadEncodingModel("db/model/encoding.bin");

// load language model for language identification or sentence tokenization
knowledge->loadLanguageModel("db/model/language.bin");

// set knowledge
analyzer->setKnowledge(knowledge);
```

```
// identify character encoding of string
EncodingID encID;
analyzer->encodingFromString("...", encID);

// identify character encoding of file
analyzer->encodingFromFile("...", encID);

// identify the single primary language of string in UTF-8 encoding
LanguageID langID;
analyzer->languageFromString("...", langID);

// identify the single primary language of file in UTF-8 encoding
analyzer->languageFromFile("...", langID);

// identify the list of multiple languages of string in UTF-8 encoding
vector<LanguageID> langIDVec;
analyzer->languageListFromString("...", langIDVec);

// identify the list of multiple languages of file in UTF-8 encoding
analyzer->languageListFromFile("...", langIDVec);

// segment the UTF-8 multi-lingual string into single-language regions
vector<LanguageRegion> regionVec;
analyzer->segmentString("...", regionVec);

// segment the UTF-8 multi-lingual document into single-language regions
analyzer->segmentFile("...", regionVec);

// get the length of the first sentence of string in UTF-8 encoding
int len = analyzer->sentenceLength("...");

delete knowledge;
delete analyzer;
```

Below are some guidelines which might be helpful in your using the interface.

5.1 How to get string representation of identification result

When you have got the identification result (EncodingID or LanguageID), you might need to get their string representation. It could be achieved using below Knowledge interface.

```
static const char* getEncodingNameFromID(EncodingID id);
static const char* getLanguageNameFromID(LanguageID id);
```

5.2 How to tokenize sentence

Given a string in UTF-8 encoding, you could use Analyzer::sentenceLength() to perform sentence tokenization like below.

```
const char* p = "...";
string sentStr;
while(int len = analyzer->sentenceLength(p))
{
    sentStr.assign(p, len);  // get each sentence
    cout << sentStr << endl; // print each sentence
    p += len;                // move to the begining of next sentence
}
```

5.3 How to print out language region

When you have got LanguageRegion using Analyzer::segmentString(), you might need to print out the i-th region content like below.

```
const char* str = "...";
vector<LanguageRegion> regionVec;
analyzer->segmentString(str, regionVec);

string regionStr(str + regionVec[i].start_, regionVec[i].length_);
cout << regionStr << endl;
```

Similarly, when you have got LanguageRegion using Analyzer::segmentFile(), you might need to print out the i-th region content like below.

```
const char* fileName = "...";
vector<LanguageRegion> regionVec;
analyzer->segmentFile(fileName, regionVec);

ifstream ifs(fileName);
ifs.seekg(regionVec[i].start_);

const unsigned int BUFFER_SIZE = 1024;
char buffer[BUFFER_SIZE];
unsigned int t;
for(unsigned int len = regionVec[i].length_; len; len-=t)
{
    t = min(len, BUFFER_SIZE);

    ifs.read(buffer, t);
    cout.write(buffer, t);
}
```

5.4 How to configure the identification process

Among the interface list in chapter 4, Analyzer::setOption(OptionType nOption, int nValue) could be used to do some configurations in identification process. OptionType is an enumeration type defined below.

```
enum OptionType
```

```
{
    OPTION_TYPE_LIMIT_ANALYZE_SIZE,
    OPTION_TYPE_BLOCK_SIZE_THRESHOLD,
    OPTION_TYPE_NO_CHINESE_TRADITIONAL,
    OPTION_TYPE_NUM
};
```

And you could set value to each option like below.

```
analyzer->setOption(Analyzer::OPTION_TYPE_LIMIT_ANALYZE_SIZE, 512);
analyzer->setOption(Analyzer::OPTION_TYPE_BLOCK_SIZE_THRESHOLD, 100);
analyzer->setOption(Analyzer::OPTION_TYPE_NO_CHINESE_TRADITIONAL, 1);
```

The meaning of each option is described below.

5.4.1 OPTION_TYPE_LIMIT_ANALYZE_SIZE

This option is used to limit input size. The default value is 1024.

If a non-zero value is configured, it is used as the maximum input size for below Analyzer interfaces. That is, only the input bytes within this range would be used in these interfaces, and the rest bytes are just ignored. When the input file size is large, the upper bound of time consumption could be ensured by using this option.

```
virtual bool encodingFromString(const char* str, EncodingID& id) = 0;
virtual bool encodingFromFile(const char* fileName, EncodingID& id) = 0;
virtual bool languageFromString(const char* str, LanguageID& id) = 0;
virtual bool languageFromFile(const char* fileName, LanguageID& id) = 0;
```

If a zero value is configured, it uses all the input bytes in these interfaces, so that the highest accuracy could be achieved.

5.4.2 OPTION_TYPE_BLOCK_SIZE_THRESHOLD

This option configures the threshold of language block size. The default value is zero.

If a non-zero value is configured, for below Analyzer interfaces, the language block (LanguageRegion), which size is not larger than this value, would be combined into adjacent larger block in a different language.

```
virtual bool segmentString(const char* str, std::vector<LanguageRegion>& regionVec) = 0;
virtual bool segmentFile(const char* fileName, std::vector<LanguageRegion>& regionVec) = 0;
```

If a zero value is configured, it disables combining blocks in different languages.

For example, a sample text “Ｍｒｓ．Ｈｏｕｓｅ負責圖書館初期規劃。” would be segmented into two blocks when it is configured as default value zero.

language	**block content**
English	Ｍｒｓ．
Chinese Traditional	Ｈｏｕｓｅ負責圖書館初期規劃。

While when it is set as value 100, any block which size is less than 100 bytes would be combined into adjacent larger block, so that the above segmented blocks would be combined into one block like below.

language	**block content**
Chinese Traditional	Ｍｒｓ．Ｈｏｕｓｅ負責圖書館初期規劃。

5.4.3 OPTION_TYPE_NO_CHINESE_TRADITIONAL

This option configures not to use Chinese Traditional language. The default value is zero.

If a non-zero value is configured, for below Analyzer interfaces, Chinese Traditional text would be identified as Chinese Simplified language.

```
virtual bool languageFromString(const char* str, LanguageID& id) = 0;
virtual bool languageFromFile(const char* fileName, LanguageID& id) = 0;
virtual bool languageListFromString(const char* str, std::vector<LanguageID>& idVec) = 0;
virtual bool languageListFromFile(const char* fileName, std::vector<LanguageID>& idVec) = 0;
virtual bool segmentString(const char* str, std::vector<LanguageRegion>& regionVec) = 0;
virtual bool segmentFile(const char* fileName, std::vector<LanguageRegion>& regionVec) = 0;
```

If a zero value is configured, Chinese Traditional text would be identified as Chinese Traditional language as original.

Below is an example when it is configured as different values.

option value	**language**	**source text**
0	Chinese Traditional	Ｈｏｕｓｅ負責圖書館初期規劃。
1	Chinese Simplified	Ｈｏｕｓｅ負責圖書館初期規劃。