Technical Report of *Language Identification*

Jun Jiang

February 8, 2010

Abstract

Language identification is a useful pre-processing step in information retrieval. For example, the indexing methods used in search engines are usually language dependent, because they require knowledge about the morphology of a language. If the document is written in Asian languages such as Chinese etc, as words are not white-space delimited in Chinese text, it would be necessary to perform word segmentation before further processing. Language identification is also closely related to the recognition of the character encoding. In this article, we aims to develop the system of language and encoding identification with high accuracy and performance.

Changes

Date	Author	Notes
2009-10-26	Jun	Initial version
2009-11-13	Jun	Add section 4.2 on N-gram frequency profile
2009-11-20	Jun	Add section 5.1 on Character Type in UTF-8 Encoding
2009-11-27	Jun	Add section 5.2 on Sentence Level Identification
2009-12-04	Jun	Add section 5.4 on Sentence Segmentation
2009-12-11	Jun	Update section 5.1 and revise the approach on sentence level Identification in section 5.2
2009-12-25	Jun	Update section 5.3 on combining small blocks in language segmentation
2009-12-31	Jun	Add chapter 6 on user guidance
2010-01-08	Jun	Add chapter 7 on experiment result
2010-02-08	Jun	Add chapter 8 on conclusion and some discussions.

Contents

Chapter 1

Introduction

Large amounts of documents have been created by all kinds of users, which documents include HTML pages, E-mails, newspapers etc. When gathering, processing and presenting these information to user, we often have to figure out what is the language these text are written in.

Moreover, given the text in unknown character encoding, it is not easy to identify even a character of a specific language. For example, one Chinese character occupies three bytes in UTF-8 encoding, while two bytes in GB-2312 encoding.

In this article, for encoding identification, we apply the n-gram language model as the text categorization learning method. Compared with those standard machine learning techniques, instead of explicitly pre-computing features, the language modeling approach simply considers all character subsequences occurring in the text as candidate features, and implicitly considers the contribution of every feature in the final model. Also, it could also avoids the word segmentation problems that arise in many Asian languages, and thereby achieves a language independent method for constructing accurate text categorizers.

For language identification, we focus on Asian languages (CJK) and English in UTF-8 encoding. As the scripts of each language are quite different, instead of using statistical approach, we utilize the script types such as ideograph, kana, hangul and Latin to identify each language.

Chapter 2

Project Requirement Overview

The goal of this project is to develop a language identification system. The main requirements are listed below.

2.1 Language and Character Encoding

The system could identify the pair of language and character encoding. It supports the languages and encodings in Table 2.1.

Table 2.1: Language and Encoding Support

Languages	**Encodings**
English	UTF-8
Chinese Simplified	UTF-8, GB18030
Chinese Traditional	UTF-8, BIG5
Japanese	UTF-8, EUC-JP, Shift_JIS
Korean	UTF-8, EUC-KR

2.2 Mono-lingual Document

Given a document comprised of single language, it could identify the single pair of language and encoding in Table 2.1.

2.3 Multi-lingual Document

Given a document comprised of multiple languages, it could identify each language, and also segment the document into blocks of individual languages.

Chapter 3

Related Works

In existing techniques for language identification, a variety of features have been used as discriminators. These include the presence of particular characters, the presence of particular words, and the presence of particular character n-grams.

As the words are not explicitly delimited by white-space in most Asian languages, we would focus on the character n-grams approaches. In this context, n-grams refer to n-character contiguous slices of a longer string.

3.1 Character N-gram

The n-gram method proposed by Canvar and Trenkle [1] is widely used in language identification. It constructs a ranking of the most frequent character n-grams for each language during the training phase. For classification, a ranking is constructed for the input document in the same fashion and is compared against each available language model. The closest model (in terms of ranking distances) wins and is returned as the identified language.

Dunning's method [2], which involves n-gram statistics and Markov models, is also reported to perform very well in discriminating two moderately-related languages, English and Spanish. It assumes that language can be modeled by a low order Markov process generating strings, and then uses Bayesian decision rules to select which of two phenomena caused a particular observation.

Another approach makes use of Shannon's information theory and compares language entropies [4]. Here Markov models are also estimated based on training data. In contrast to [2], all models of orders $0, \cdots, k$ are used and their relationship explicitly modeled. This allows the algorithm to fall back

to lower order models in case of insufficient data through mechanism called *escape probabilities*. Decision function views input as a stream of characters and in accordance with information theory tries to predict the next character in the stream. Success of these predictions is measured by cross-entropy and the model with the lowest cross-entropy after having processed the whole stream wins. Because of its ties to information theory and language compression, this technique is sometimes called the *compression* technique.

3.2 Language Segmentation

When the input document contains blocks from different languages, it is necessary to locate and identify all compact single-language blocks.

One attempt at language segmentation was made in [4]. It considers all possible combinations of language change at each character in the input text and measure the resulting entropy on such text blocks.

Another word-window approach is applied in [3]. It uses tri-grams and a windows size of eight words. For each window, the most likely language is determined based on the transition probability between tri-grams. For windows in which a different language occurs, a language change is assumed. The position is determined based on the position of the first window in which the new language is encountered. The change is assumed to occur at the first position of that window plus two words.

A recent approach [5] uses signal processing method to estimate the potential segment boundaries.

Chapter 4

Encoding Identification

As the value range in those encoding of Table 2.1 is overlapping, there is no explicit rule on how to distinguish the encoding type. While we could utilize the statistical approach to extract the different characteristics of each encoding. We use the N-gram frequency profiles. It is simple and efficient. The details are described below.

4.1 N-grams

An N-gram is an N-character slice of a longer string. Typically, one slices the string into a set of overlapping N-grams. In our system, we use N-grams of several different lengths simultaneously.

As all languages share the whitespace characters, such as space, new line, tab, etc, we would replace them into the underscore character "_" to unify their occurrences. Thus, the string "n grams" would be composed of the following N-grams when N is 5:

1-grams: n, _, g, r, a, m, s

2-grams: n_, _g, gr, ra, am, ms

3-grams: n_g, _gr, gra, ram, ams

4-grams: n_gr, _gra, gram, rams

5-grams: n_gra, _gram, grams

N-gram-based matching has had some success in dealing with noisy ASCII input in other problem domains, such as in text retrieval and in a wide

variety of other natural language processing applications. The key benefit that N-gram-based matching provides derives from its very nature: since every string is decomposed into small parts, any errors that are present tend to affect only a limited number of those parts, leaving the remainder intact. If we count N-grams that are common to two strings, we get a measure of their similarity that is resistant to a wide variety of textual errors.

4.2 Classification using N-grams Frequency Statistics

Human languages invariably have some words which occur more frequently than others. One of the most common ways of expressing this idea has become known as Zipf's Law, which we can re-state as follows:

> The nth most common word in a human language text occurs with a frequency inversely proportional to n.

The implication of this law is that there is always a set of words which dominates most of the other words of the language in terms of frequency of use. This same law holds, at least approximately, for the frequency of occurrence of N-grams, both as inflection forms and as morpheme-like word components which carry meaning.

Zipf's Law implies that if we are comparing documents from the same category they should have similar N-gram frequency distributions.

To use N-grams frequency statistics in classification, we start with a set of pre-existing text categories for which we have reasonably sized samples. From these, we would generate a set of N-gram frequency profiles to represent each of the categories. When a new document arrives for classification, the system first computes its N-gram frequency profile. It then compares this profile against the profiles for each of the categories using an easily calculated distance measure. The system classifies the document as belonging to the category having the smallest distance.

4.2.1 Generating N-gram Frequency Profiles

The frequency profile generation is simple, it merely reads incoming text, and counts the occurrences of all N-grams. To do this, the system performs the following steps:

- Scan down each token, generating all possible N-grams, for N=1 to 5. The whitespace characters are replaced with the underscore character "_" to unify their occurrences.
- Hash into a table to find the counter for the N-gram, and increment it.
- When done, sort those N-grams into reverse order by the number of occurrences. Keep just the top 400 N-grams, which are now in reverse order of frequency. The resulting file is then an N-gram frequency profile for the document.

Please note that the parameters in above steps are just for illustration, such as N=5, and 400 top N-grams. We would choose the optimal parameters in our experiment.

We can make the following informal observations from an inspection of those profiles: the top N-grams generated are almost always highly correlated to the pair of language and encoding. For example, on the same encoding of UTF-8, the profiles from Korean text and Chinese text are very different. And on the same language of Chinese, the profiles from UTF-8 and GB18030 encoding are also very different.

4.2.2 Comparing N-gram Frequency Profiles

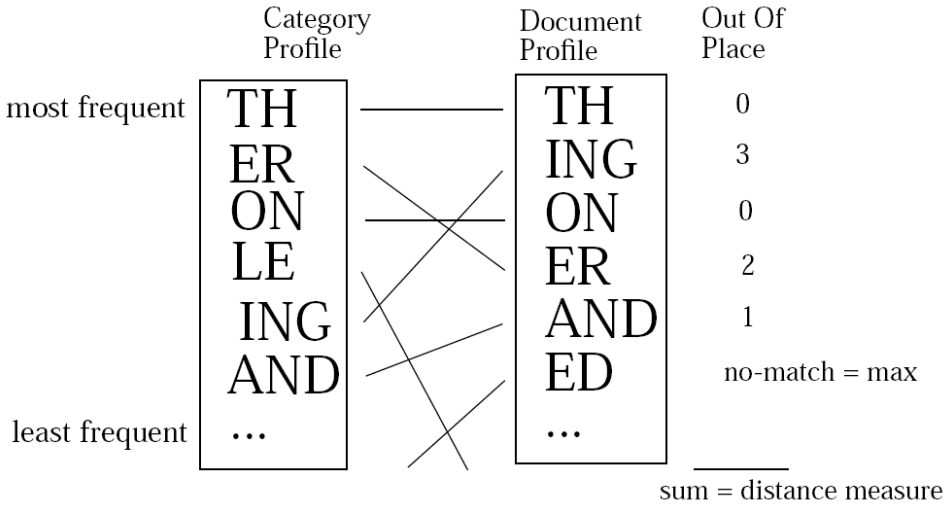
To measure the profile distance, it merely takes two N-gram profiles and calculates a simple rank-order statistic we call the "out-of-place" measure. This measure determines how far out of place an N-gram in one profile is from its place in the other profile.

Figure 4.1 gives a simple example of this calculation using a few N-grams. For each N-gram in the document profile, we find its counterpart in the category profile, and then calculate how far out of place it is. For example, in Figure 4.1, the N-gram "ING" is at rank 2 in the document, but at rank 5 in the category. Thus it is 3 ranks out of place. If an N-gram (such as "ED" in the figure) is not in the category profile, it takes some maximum out-of-place value. The sum of all of the out-of-place values for all N-grams is the distance measure for the document from the category.

The out-of-place score provides a simple and intuitive distance measure that seems to work well enough for these classification tasks.

Finally, to find the minimum distance, it simply takes the distance measures from all of the category profiles to the document profile, and picks the smallest one.

Figure 4.1: Calculating The Out-Of-Place Measure Between Two Profiles



4.2.3 Testing on language and encoding classification

This approach is tested on some texts in the language and encoding pairs listed in Table 2.1. The result shows that:

1. Encodings could be identified well. The reason could be that the characteristics of each encoding is very different. We also tried different parameters in n-gram profiles comparison. The result shows that, the profiles containing 2,000 bi-grams achieve a good balance of accuracy and performance.

2. Languages such as English and Korean in UTF-8 encoding could be identified well. The reason could be that the letter set of English and Korean is unique compared with other languages.

3. On small test size such as 10 bytes in UTF-8 encoding, the identification on languages such as Chinese Simplified, Chinese Traditional and Japanese is not so good compared with large test size. The reason could be that these languages share the Chinese characters, which could not be distinguished rigorously using profile comparison.

We could see that this approach is suffice in our encoding identification. While for language identification on small data size, especially on UTF-8 encoding, to overcome the limitation of N-grams profile comparison in

above point 3, other approach could be helpful, which would be described in next chapter.

Chapter 5

Language Identification

From Table 2.1, we could see that the language type on native encodings (that is, except UTF-8) could be determined by its encoding type naturally. For example, GB18030 encoding infers Chinese Simplified language, and EUC-KR encoding infers Korean language. As encoding identification could be achieved by using N-grams profiles in section 4.2, we would focus on the language identification in UTF-8 encoding in this section.

5.1 Script Type on UTF-8 Encoding

The basic idea would base on the script types in each languages. Concerning the five languages in Table 2.1, we could see that different script types could be a hint to what language it belongs to. The script types are listed in Table 5.1.

Table 5.1: Script and Language Type

Priority	**Script Type**	**Language Type**	**Example**
1	Hangul	Korean	Hangul letters
2	Hiragana and Katakana	Japanese	あ い う え お
3	Ideograph (Traditional)	Chinese Traditional	圖書館
4	Ideograph (Simplified)	Chinese Simplified	图书馆
5	Alphabet	English	ABC abc

Regarding CJK unified ideograph characters, as they could appear in Chinese Simplified, Chinese Traditional and also Japanese text, to distinguish

between Chinese Simplified and Chinese Traditional languages, we used a table to look up those Chinese Traditional characters such as “圖書館”, etc.

The process of identify the character type could be like below:

1. Tokenize UTF-8 string into each characters.
2. Convert each character to UCS2 value.
3. Look up the character type in Unicode table by the UCS2 value.

5.2 Identification on Sentence Level

Based on the script type of each character, in case of different script types existing in one sentence, to determine the main script type, we defined a priority order in the 1st column of Table 5.1.

Some examples are listed in Table 5.2. In the text containing alphabet, ideograph and Hiragana, it is identified as Japanese, as the priority of Hiragana is the highest.

Table 5.2: Examples of Sentence Level Identification

Raw Text	**Main Script Type**	**Language Type**
WTO	Alphabet	English
WTO世界	Ideograph (Simplified)	Chinese Simplified
WTO世界貿易機関	Ideograph (Traditional)	Chinese Traditional
ＷＴＯ世界貿易機関のルールでは	Hiragana and Katakana	Japanese

When the text contains no script type in Table 5.1, it is identified as Unknown language, such as text containing only digits or punctuations.

5.3 Identification on Document Level

To identify the primary single language type on document, it consists steps below:

1. Segment the document into sentence units without knowing the language type of the text.

2. Identify the language type of each sentence using the approach in section 5.2.

3. From the language types of those sentences, use the primary type as the document type.

The language segmentation would use a similar approach. That is:

1. Segment the document into sentence units without knowing the language type of the text.

2. Identify the language type of each sentence using the approach in section 5.2.

3. Group the sentences together with the same language type as a block, and return those blocks in different language type.

5.4 Sentence Boundary

From 5.3, we could see that document needs to be segmented into sentence units. According to Unicode Standard Annex #29 http://www.unicode.org/reports/tr29, we could define the sentence break property in Table 5.3.

Boundary determination is specified in terms of an ordered list of rules, indicating the status of a boundary position. They are summarized in Table 5.4.

Each rule consists of a left side, a *boundary symbol*, and a right side. Either of the sides can be empty. The left and right sides use the boundary property values in regular expressions. As the *boundary symbol*, ÷ means the sentence boundary, and × means no sentence boundary between characters. The two special identifiers *sot* and *eot* stand for start and end of text, respectively.

The rules are numbered for reference and are applied in sequence to determine whether there is a boundary at a given offset. That is, there is an implicit "otherwise" at the front of each rule following the first. The rules are processed from top to bottom. As soon as a rule matches and produces a boundary status (boundary or no boundary) for that offset, the process is terminated.

These rules are designed to forbid breaks within strings such as

Table 5.3: Sentence Break Property Values

Value	**Summary List of Characters**
CR	carriage return
LF	line feed
Extend	combining grave accent, etc
Sep	line separator, etc
Format	U+FEFF as zero width no-break space, etc
Sp	space, tab, ideographic space
Lower	small letter such as "a" in half and full width
Upper	capital letter such as "A" in half and full width
OLetter	CJK ideographic and Hangul letter, etc
Numeric	digit such as "0" in half and full width
ATerm	full stop such as "." in half and full width
STerm	exclamation and question mark such as "!?" in half and full width, etc
Close	quotation mark such as " in half and full width, etc
SContinue	comma such as "," in half and full width, etc
Any	*This is not a property value; it is used in the rules to represent any code point.*

Table 5.4: Sentence Break Rules

rule number	**rules in regular expression**
0.2	sot ÷
0.3	÷ eot
3.0	CR × LF
4.0	(Sep | CR | LF) ÷
5.0	× [Format Extend]
6.0	ATerm × Numeric
7.0	Upper ATerm × Upper
8.0	ATerm Close* Sp* × [¬ OLetter Upper Lower Sep CR LF STerm ATerm]* Lower
8.1	(STerm | ATerm) Close* Sp* × (SContinue | STerm | ATerm)
9.0	(STerm | ATerm) Close* × (Close | Sp | Sep | CR | LF)
10.0	(STerm | ATerm) Close* Sp* × (Sp | Sep | CR | LF)
11.0	(STerm | ATerm) Close* Sp* (Sep | CR | LF)? ÷
12.0	× Any

c.d
3.4
U.S.
... the resp. leaders are ...
... etc.)' '(the ...

While they permit breaks in strings such as

She said "See spot run."	John shook his head. ...
... etc.	它们指...
...理数字.	它们指...

While they cannot detect cases such as "...Mr. Jones...", more sophisticated tailoring would be required to detect such cases.

Chapter 6

User Guide

6.1 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
```

6.2 How to run the demo

To run the demo bin/test_langid_run, please make sure the project is built (see 6.1).

Below is the demo usage:

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

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

```
$ ./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
```

6.3 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.

6.4 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;
```

6.5 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.

6.5.1 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);
```

6.5.2 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
}
```

6.5.3 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);
}
```

6.5.4 How to configure the identification process

Among the interface list in section 6.4, 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.

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.

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。

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。

Chapter 7

Experiment

The approaches in chapter 4 and 5 are experimented on accuracy and performance below.

7.1 Encoding Identification

N-gram frequency profiles are used in encoding identification. In our experiment, we found the profiles containing 2,000 bi-grams achieve a good balance of accuracy and performance. The test result on these profiles is described below.

7.1.1 Accuracy

The test data is grouped by whether its length is over or under 30 bytes. The result of accuracy percent is listed in table 7.1. Each item in the table is tested on 2500 texts randomly chosen.

From the table, we could see that the accuracy on short text is slightly lower than long text. The reason is that for short text, there would be a smaller amount of text from which to compute N-gram frequencies. On the whole, the accuracy is only slightly sensitive to text length.

When the text contains multiple languages, the accuracy might be impacted. Below is an example illustrating how multi-language text could impact the result.

Table 7.1: Accuracy on Encoding Identification

Test Size	**≤ 30 bytes**	**> 30 bytes**
GB18030	95.52	100
BIG5	100	100
EUC-JP	98.8	100
Shift_JIS	99.72	100
EUC-KR	99.92	100
UTF-8	100	100
Overall	98.99	100

Raw Text	**Source Encoding**	**Identified Result**
上海世博会	GB18030	GB18030
Shanghai World Expo	UTF-8	UTF-8
上海世博会 Shanghai World Expo	GB18030	UTF-8

In above example, the first two lines could be identified correctly. While the third line containing two languages is identified as UTF-8 incorrectly, which source encoding is GB18030.

This was because the N-grams of both languages are generated into one profile. This "multi-language" profile is compared with each "mono-language" profile from training, so that the high accuracy could not be kept using the similarity measure approach in section 4.2.2.

Such situation is caused by the compatibility of English ASCII characters with each encoding type in table 2.1. To alleviate the impact by multi-language text, the input text length could be limited to avoid such situation using configuration type OPTION_TYPE_LIMIT_ANALYZE_SIZE in interface 6.5.4.

7.1.2 Performance

The test environment for performance evaluation is described in Table 7.2.

The execution time on encoding identification is evaluated on different text length. The statistics is listed in Table 7.3.

We could see that the time consumption is almost linear to input text length. Regarding the accuracy result in table 7.1, a mediate input length is suffice to ensure a high accuracy. For example, the input length is limited to 1K bytes by default in our system. You could also select other limitation size

Table 7.2: Test Environment

Platform	Ubuntu x86_32 Kernel Linux 2.6.28-14
Memory	3.7GB
CPU	Intel Core 2 Duo 2.33GHz

Table 7.3: Time Statistics (seconds) on Encoding Identification

1K bytes	1M bytes	10M bytes	100M bytes
0.036	0.356	3.210	30.892

using configuration type OPTION_TYPE_LIMIT_ANALYZE_SIZE in interface 6.5.4.

7.2 Language Identification

Unlike the statistical approach for encoding identification, the rule based approach in chapter 5 is used to identify language. So that if only the input text follows the rule we defined, it could give out the correct result.

Although in most cases it could meet our requirement, we would like to discuss some limitations and possible solutions regarding our approach used.

7.2.1 Limitation and Solutions

Language Segmentation

One problem is found in language segmentation. For example, a sample text from Chinese Traditional language source is like "Ｍｒｓ．Ｈｏｕｓｅ負責圖書館初期規劃。". Basing on our segmentation approach, it would be segmented into two blocks like below, while it should be one block in semantics.

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

We could see that, as our sentence tokenization approach relies only on sentence-final punctuations such as period(.) and question mark(?), etc, it

could not suffice for those sentences containing abbreviations such as “Mrs.”.

To avoid splitting above text into two sentences, we plan to enhance our sentence tokenization. That is, those common abbreviations are disallowed to be the second-to-last token in a sentence.

As we could not exhaust all abbreviations, to avoid such problem in language segmentation, a threshold value could be configured to limit the minimum block size using configuration type OPTION_TYPE_BLOCK_SIZE_THRESHOLD. The details could be found in section 6.5.4.

Japanese Identification

In current identification on Japanese language, it depends only on Hiragana and Katakana such as “あ い う え お”, etc. While it could not suffice for those sentences containing only ideographs like below. As we assumed all ideographs as Chinese language, it is identified as Chinese Traditional incorrectly.

大隅良典（自然科学研究機構基礎生物学研究所教授）

To overcome such problem, we plan to enhance Japanese identification by looking up Japanese words from dictionary. If each word could be found as a Japanese word, we would assume the whole sentence as Japanese language.

7.2.2 Performance

The execution time on language identification is evaluated on different text length. The statistics is listed in Table 7.4.

Table 7.4: Time Statistics (seconds) on Language Identification

1K bytes	1M bytes	10M bytes	100M bytes
0.034	0.062	0.317	2.851

We could see that the time consumption is linear to input text length. In executing language segmentation, a similar performance could be achieved. Just like encoding identification, you could also limit input size other than 1K bytes using configuration type OPTION_TYPE_LIMIT_ANALYZE_SIZE in interface 6.5.4.

Chapter 8

Conclusion

8.1 Quality and Performance

We evaluated identification on both encoding and language results.

For encoding identification, as it is statistical based approach, when the input text length is long enough (more than 30 bytes), the accuracy is close to 100 percent. Otherwise when the text is short (less than 30 bytes), the accuracy could be sustained higher than 95 percent.

For language identification, as it is rule based approach, the accuracy is 100 percent on the rules we defined.

Regarding the performance, the time consumption is linear to input text length. Our system also supplies the configuration on input size limitation in interface 6.5.4, so that the upper bound of time consumption could be ensured on any large file.

The evaluation result shows that our system is capable in realistic usage.

8.2 What Affects the Quality

For encoding identification, the longer the input text length, the higher accuracy could be achieved. When the text contains multiple languages, the accuracy on encoding result might also be impacted. So it would better to use mono-language input text on encoding identification.

For language identification, as it is rule based, when input text contains only ideographs with no Hiragana or Katakana, it would be identified as Chinese language. While this is not true for all Japanese text. So that some text

from Japanese source might be identified as Chinese language incorrectly as long as it contains no Hiragana or Katakana.

8.3 Future Work

We can improve our system to overcome current limitations, such as Japanese identification, sentence tokenization. The details of the improvement solutions are discussed in section 7.2.1.

Appendix A

Appendix - Project schedules and milestones

	Milestone	Start	Finish	In Charge	Description	Status
1	Survey on the project	2009-10-19	2009-10-23	Jun	1. Have a general overview of current language identification techniques and their differences. 2. Select a suitable one and do more research on it.	Finished
2	Analyze requirement specification, and study the potential solutions	2009-10-26	2009-10-30	Jun	Analyze requirements, and study the candidate approaches.	Finished
3	Implement the language identification system	2009-11-02	2009-12-11	Jun	Implement an initial version of this project.	Finished
4	Experiment and debugging	2009-12-14	2010-01-08	Jun	Experiment the system on training data and test data, and debugging the system.	Finished
5	Implement a web demo	2010-01-11	2010-01-22	Jun	Implement a web demo for encoding and language identification.	Finished
6	Wrap up the whole project.	2010-01-25	2010-01-29	Jun	Finish TR and doxygen documents.	Finished

Bibliography

[1] W. B. Cavnar and J. M. Trenkle. N-gram-based text categorization. *Proceedings of SDAIR-94, the 3rd Annual Symposium on Document Analysis and Information Retrieval*, pages 161–175, 1994.

[2] T. Dunning. Statistical identification of language. *Technical Report MCCS 94-273, New Mexico State University*, 1994.

[3] Thomas Mandl, Margaryta Shramko, Olga Tartakovski, and Christa Womser-hacker. Language identification in multi-lingual web-documents. *Applications of Natural Language to Data Bases - NLDB*, pages 153–163, 2006.

[4] W. Teahan. Text classification and segmentation using minimum cross-entropy. *Proceeding of RIAO 2000, 6th International Conference Recherche d'Information Assistee par Ordinateur, Paris, FR*, page 943–961, 2000.

[5] Radim Řehůřek and Milan Kolkus. Language identification on the web: Extending the dictionary method. *Computational Linguistics and Intelligent Text Processing, 10th International Conference, CICLing 2009, Proceedings*, pages 357–368, 2009.

Index