Diction Identifier

Developed By

Fouzia Latif
Farheen Hanif

Supervised By
Dr. M. Sikandar Hayat Khiyal

Department of Computer Science
Faculty of Applied Sciences
International Islamic University
Islamabad
2006

In
The
Name
Of
ALLAH
The Most Merciful
The Most Beneficial

A
Dissertation
Submitted as Partial Fulfillment
of Requirements for the Award of the Degree of
MS in Computer Science

DEDICATION

To
The Holiest man Ever Born,
Prophet Muhammad (Peace Be Upon Him)
&
Our Dear Parents
Who are an embodiment of diligence and honesty,
Without their prayers and support
This Dream could has never come true
&
Our
Precious Friendship that has made us
laugh, held us when we cried and always, always, always
be among us

DECLARTAION

We hereby declare that this software neither as a whole nor as a part has been copied out from any source. It is further declared that we have developed this software and accompanied thesis on the basis of our personal efforts, under the sincere guidance of our supervisor Dr. M. Sikandar Hayat Khiyal. If any part of this system is proved to be copied out from any source or found to be reproduction of someone else, we shall stand for the consequences.

Fouzia Latif
218 -CS/MS/04

Farheen Hanif
215-CS/MS/04

ACKNOWLEDGEMENT

Countless gratitude to the Almighty Allah, who is Omnipotent and He, who blesses us with the ability to read and write. He blessed us with a chance and choice, health and courage to achieve this goal.

Without the support and prayers of our families we could have never completed this project work. They always helped us in our times and boosted our morals. We would like to pay then the very special thanks for their best wishes, encouragement and support in not only this project but throughout our lives without that we would have not been able to achieve anything worthwhile.

We would like to pay special thanks to our supervisor Dr. M. Sikandar Hayat Khiyal, Head, Department of Computer Science, International Islamic University, Islamabad without his proper guidance we could never made this project a reality.
We also owe a great thanks to our dear brother Kashif Iftikhar who extended his help whatsoever we needed.

Fouzia Latif
218 -CS/MS/04

Farheen Hanif
215-CS/MS/04

Department of Computer Sciences
International Islamic University, Islamabad

Dated: 27-5-2016

Date:

Final Approval

It is certified that we have read the thesis submitted by Fouzia Latif (Reg # 218-CS/MS/04) and Farheen Hanif (Reg # 215-CS/MS/04) and it is our judgment that this thesis is of sufficient standard to warrant its acceptance by the International Islamic University, Islamabad for the MS Degree in Computer Science.

Committee:

External Examiner
Mr. Shaftab Ahmed
Engineer coordinator
House No. 460, Street No. 68,
G-11/2, Islamabad
BAHRIA UNIVERSITY

Internal Examiner
Mr. Asim Munir
Faculty Member
Department of Computer Science
IIUI

Supervisor
Dr. M. Sikandar Hayat Khiyal
Head, Department of Computer Science
International Islamic University,
Islamabad

PROJECT IN BRIEF

PROJECT TITLE	:	Diction identifier
UNDERTAKEN BY	:	Fouzia Latif Reg # 218 -CS/MS/04 Farheen Hanif Reg # 215 -CS/MS/04
SUPERVISED BY	:	Dr. M. Sikandar Hayat Khiyal Head, Department of Computer Science, International Islamic University Islamabad
STARTED ON	:	September 2005
COMPLETED ON	:	May 27, 2006
TOOLS	:	Python (Wing IDE 2.0 professional) PHP HTML Linux 2.4 (Slackware 10.2) Apache Web Server

ABSTRACT

The purpose of the described research is to develop a fast and accurate language detection system that is able to detect languages belonging to different scripts. The system is broadly divided into two general steps (*Training & Identification*). Two different techniques are used for language identification both having their pros and cons.

The system is meant to be integrated into many bigger systems, such as search engines, digital libraries, and discussion forums etc., where automated language detection is useful. The whole language detection process relies on the factors like: quality of training documents, type of input document, size of input document, multiple languages/scripts within the same document

The described research provides Unicode support and multiple script support. It also provides language detection for multilingual documents.
Multiple scripts within the same document are detected accurately however detecting languages belonging to the same script in a document is a much up-hill task.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1. Introduction

The fundamental purpose of "language" identifiers is to indicate distinctions related to linguistic properties and specifically distinctions that are relevant for information technology purposes. There are a wide variety of distinctions pertaining to several distinct linguistic parameters that have been suggested as potentially relevant for "language" identification: languages, language families, dialects, country variants, other regional-based variants, script variants, style variants, and modality variants, time based variants and typographic variants. Many different orthogonal parameters could be used in meta-data attributes, and the potential combinations and permutations are daunting. In actual practice many of the potential distinctions are not needed for most realistic usage scenarios.

Application areas can probably be divided into two general types: cataloging and retrieval of content and resources for localization and language enabling of software.
Diction identifier provides Unicode support and multi-script support (Roman, Chinese, Arabic). It also provides language detection for multilingual documents.

1.1 General Techniques

The system uses an array of different techniques for language identification. Some of the techniques are suited for one scenario while others are more suited for different scenarios.

1.1.1 Cavnar's and Trenkles algorithm

C & T algorithm is an n-gram based method. It is one of the most popular method used for language identification [1].It is the most effective method for identifying languages having same script. Detection using this algorithm does not only rely on alphabets present in a language rather it focuses on the most common combinations of these alphabets in different languages. For this reason this method is more effective than the primitive/traditional dictionary based method for language identification. As languages belonging to the same script (for example Roman) may have similar or same alphabets but the order in which they occur is not the same. For example, the 3-grams 'the' and 'ing' are the most common 3-grams in English but they are not found that frequently in other Roman script languages like German and French.

1.1.2 Dictionary based method

In this method a list of unique characters for each language is generated and compared against input list of unique characters in the document that is to be identified. This method is much faster than C&T algorithm and performs well when comparing languages belonging to different scripts.

1.1.3 Unicode based script/language identification

The Unicode standard defines unique code points for each alphabet in a language. These code points are categorized according to language and script. In this technique Unicode code points of characters in input document are analyzed and categorized according to the script range in which they occur.

1.1.4 Inter document language/script separation and identification

In this method Unicode code point script ranges are used to identify different scripts in input document. This method is very effective for detecting the presence of multiple scripts/languages in a document.

1.2 Purpose of System

The purpose of the described research is to develop a fast and accurate language detection system that is able to detect languages belonging to different scripts. The system is broadly divided into two general steps (Training & Identification). Two different techniques are used for language identification both having their pros and cons.

The system is meant to be integrated into many bigger systems, such as search engines, digital libraries, and discussion forums, where automated language detection is useful. Keeping this in mind

The whole language detection process relies on the factors mentioned below:

- Quality of training documents
- Type of input document
- Size of input document
- Multiple languages/scripts within the same document

Proper training of the system is absolutely critical for successful language identification. Quality of training documents matters a lot and these training documents have to be filtered to make sure that they only contain the language they are used to train for. Corrupted training documents can easily cause the system to give incorrect results.

Type of input document is also an important factor. The system accepts Unicode UTF-8 encoded documents. UTF-8 is a Unicode encoding scheme [9] [11]. All types of documents (text, doc, PDF, html, etc.) can be saved using UTF-8 encoding. On windows UTF-8 has been the default encoding scheme since windows 2000.

The system is known to perform badly on very short documents (10-30 characters). Very short documents present too few information that can be processed for language identification.Multiple scripts within the same document are detected accurately however detecting languages belonging to the same script in a document is a much up-hill task.

1.3 HISTORY

Automated language identification has been the focus of research in the past decades. However majority of the work has been done on Roman script languages only. In mid-80s Cavnar and Trenkle proposed their algorithm for language identification. After this algorithm, language identification was considered a solved problem in that era [1]. In those times all the data present on computer systems was either in English or other Roman script languages.

After the explosion of Internet, the world of computers quickly began expanding to multiple regions and with that, to multiple languages. The presence of multi-lingual data on the web has once again prompted the need for a generic language identifier that has the ability to handle multiple scripts.

Having multiple scripts and conversion to the Unicode standard has opened new possibilities for language detection. These possibilities were mostly un-explored till now. The latest known research by Katia Hayati [1] in Jun, 2004 also only considers Roman languages though it uses web specific information for language detection.

1.4 Main Features

Main features of the system include:

1.4.1 Unicode Based Script Separation within a Single Document

The system is not only able to read/write Unicode files; it also uses Unicode extensively for language identification. Use of Unicode makes language identification much more flexible, accurate and faster.

1.4.2 Unicode Based Script Separation within a Single Document

Unicode code point information is used to detect the presence of multiple scripts in a document. Data of each script is then processed separately and language identification results for each script chunk are returned.

1.4.3 Dictionary Based (Fast) Language Detection Method

Alphabets in the input document are compared against the alphabets of different languages and based on that the language of the input document is guessed. This method is the most used method for language identification. Although simple, the results returned from this method were found to be satisfactorily accurate and quite fast.

1.4.4 N-Gram Based (More Accurate) Modified Method

This is a modified (improved) version of Cavnar's and Trenkle's algorithm [1]. The modifications give higher weights to n-grams that are less common in other languages.

1.4.5 Automatic Removal of Document Formatting Information

The system handles input documents like web pages very well and automatically removes all tags and other formatting info from the page using only the page content for language identification.

1.4.6 Support for Easily Adding More Languages into the System

More languages can be easily trained using a couple of training programs to expand the system to support more languages.

1.5 Future Enhancements

With the ever changing face of information in the computer world, more possibilities for improving language detection are just around the corner. There are many standards and save meta-data along with documents. The increasing adoption of such standards (like XML) will allow use of document meta data for language identification.
Automatic generation of training documents without having the need to check them manually is also a field that requires more research.
Detection of multiple languages belonging to the same script within a document is another field that needs improvement.

CHAPTER 2

LITERATURE SURVEY

2. LITERATURE SURVEY

2.1 Research Study

Katia Hayati [1] improves on the existing widely used n-grams based technique for language identification proposed by Cavnar and Trenkle [1]. The focus is especially on language detection for web documents.

A sample of 1359 web pages spanning eleven languages (Danish, German, English, Spanish, Finnish, French, Italian, Dutch, Norwegian, Portuguese, and Swedish) are obtained and classified according to language. The encoding scheme used is in the Windows-1252 character set [12] (a superset of the ISO-8859-1 character set, also called Latin-1).
The feature selection stage and similarity measure mechanism for the algorithm are improved using Fisher discriminant function and the cosine similarity metric respectively. One observation mentioned in the paper is that the technique performs badly on very short documents.
The use of Web-specific information, namely in links, to improve the performance of the classifier on very short Web documents is also discussed. The new information more than doubles the accuracy of the classifier on pages with less than 25 characters, and slightly less than doubles the accuracy on pages with less than 50 characters.
Investigating top-level domain information for language identification is identified but not done in this research.

Zhong GU and Daniel Berleant [3] discuss n-grams technique for machine readable language identification. The main focus of the paper is one removing the shortcomings of n-grams technique, which are huge memory and processing requirements. The fact that most of the "all possible n-grams" for a language actually never occur in real documents presents room for improvement.
The paper discusses two possible solutions to this problem.

1. Using actually occurring and most common n-grams of a language.

2. Using hash tables by having one hash to represent more than one n-grams.
The author focuses his discussion on hash table technique to reduce the size of n-grams table by using representative hash tables instead.

Identifying good and bad hash table address this problem sizes over a wide range of sizes. It is also observed that English, French, and German n-grams behave similarly when hashed, and that this is unlike the behavior of randomly generated n-grams. Therefore the difference in behavior is due to properties of the languages themselves. Different table sizes are investigated and sizes that are particularly good when hashing n-grams during processing of these languages are identified.
Clive Souter, Gavin Churcher, Judith Hayes, John Hughes & Stephen Johnson [14] describes an experiment in the development and use of bigraph and trigraph models for automatically recognizing written natural languages. The models are extracted from corpora of different languages, and then employed to identify new texts probabilistically. It describes three approaches to the task of automatically identifying the language a text is

written in. Experiments are conducted to compare the success of each approach in identifying languages from a set of texts in Dutch/Friesian, English, French, Gaelic (Irish), German, Italian, Portuguese, Serbo-Croat and Spanish.

The three techniques we chose to investigate are:

- **Unique character string identification**

  This involved finding (empirically or using linguistic 'competence') short strings of characters which are unique to each language. The simplest identification technique might be to find a string of characters in the Latin alphabet which are unique to a particular language.

- **Frequent word recognition**

  Another method explored was to extract frequency ordered wordlists, and choose the top 100 words for each language. Unseen text would then be analyzed word by word, looking up each candidate in the list for each language, and adding to a running total or likelihood for each. At any time, or at the end of the text, it returns the most likely language. An alternative method is to extract a frequency ordered list of the words in each language from the training material. Then, the most frequent words in each language can be used as a test list against which the words in a new, unknown sentence can be matched. Some of the words in the unknown text are found in the test list.

- **Bigraph/trigraph based recognition**

  All possible two- and three-letter combinations are extracted from the training texts, along with their frequencies in each language. Unseen text is then analyzed by similarly splitting up the text into ordered bi/trigraph, and a running total probability for each language maintained. We can return the most likely language at any stage.

Each method was implemented by training the model on roughly 100 kilobytes of text and tested on text samples. The bigraph recognition was 88% successful, being surpassed by the 'most- common-word' approach, which correctly identified the language in 91% of the test samples. However, trigraph approach showed recognition with 94%accuracy.
Results showed that a trigraph model is the most successful for recognizing the languages. Bigraph and trigraph models can be used to classify languages along the lines of a historical linguistic family tree for Indo-European languages.

There are many Issues in sentence categorization according to language and it is a fundamental step in document processing. Emmanuel Giguet [5] proposed an approach to sentence categorization which has the originality to be based on natural properties of

languages with no training set dependency. The paper also aims to point that the more the linguistic properties of the object are used, more better results are observed.
The implementation is fast, small, robust and textual errors tolerant and tested for French, English, Spanish and german discrimination.The resolution power is based on grammatical words (not the most common words) and alphabet. Having the grammatical words and the alphabet of each language at its disposal, the system computes for each of them.
Categorization according to language is done with text. The goal of text categorization is to tag texts with the name of the language in which they are written. Information retrieval is the main application field. Grammatical Words are used as they are short, not numerous and we can easily build an exhaustive list. Grammatical words in sentences represent on average about 50% of words. They can't be omitted because they structure sentences and make them understandable.
To improve categorization of short sentences, alphabets are used because alphabets are Proper to each language
Mainly two ways can be explore to improve categorization, using natural languages properties:

- Syllabation: This gives the ideas to check the good syllabation of words in a language. It requires distinguishing first, middles and last syllabs. (Using only endings seems to be a possible way)

- Sequences of vowels or consonants: the idea is that these sequences are proper to each language.

Heuristical knowledge is also used to deal with texts. In a same paragraph, contiguous sentences are written in the same language. Titles of a paragraph are written in the same language as their body. Included blocks in a sentence (via parenthesis. . .) are written in the same language as the sentence.
The techniques are implemented by sentence categorization and language classification. This classification method is based on texts observation and understanding of their natural properties. It does not depend on training sets and converges fast enough to achieve very good results on sentences.

Gregory Grefenstette's paper [6] focuses on two techniques for automatic language identification. Machine readable text is given using easily calculable attributes.
The two techniques are

- Trigrams
- Short Words

The trigram technique calculates the frequency of sequences of three letters in a large language sample. The idea is to capture the intuition that, e-g, a word ending with –ing is more likely to be an English word whereas a word ending with –ez is more likely to belong to French language. Each text is tokenized, space is used as separator and underscore is added in the beginning and end of each token to indicate the initial and terminal mark. All

the sequences are counted and then probability of given trigram in a given language is approximated. A minimum probability is assigned to each unseen trigram. A language with highest probability is chosen.
Small word technique is based on the intuition of determiners, conjunctions and prepositions. The first million characters of the text of each language was tokenized and all tokens of five characters or less were extracted. These were counted for each language and words appearing more then three times were retained. The frequencies of these words were transformed into probabilities.

Peter G. Constable's paper [15] serves as a comprehensive backgrounder for language identification. It describes different components of a language and how it is represented. The proposed model involves four core category types: individual languages, writing systems, Orthographies, and domain-specific data sets.
This paper goes further, though, in also suggesting that these various category types stand in certain relationships to one another, and that these relationships motivate certain constraints on the way in which composite identifiers are formed.

This paper is intended as a starting point for discussion and development, not as a finished Proposal. It is expected that others will find many ways in which refinements can be made in the Model and comments to that effect are welcomed in the hope that such a dialog can soon lead to adequate solutions.

Muntsa Padr´o and Llu´ýs Padr´o [7] compared three different statistical language identification methods, and a detailed study of the influence on those systems of some basic parameters is performed. The analyzed parameters are the size of the train set, the amount of text that are to be classify and the languages the system is able to distinguish (it will be studied not only the influence of the number of languages but also the influence of which are the considered languages).figure 2.1 shows the general architecture of the system as discussed in the paper.

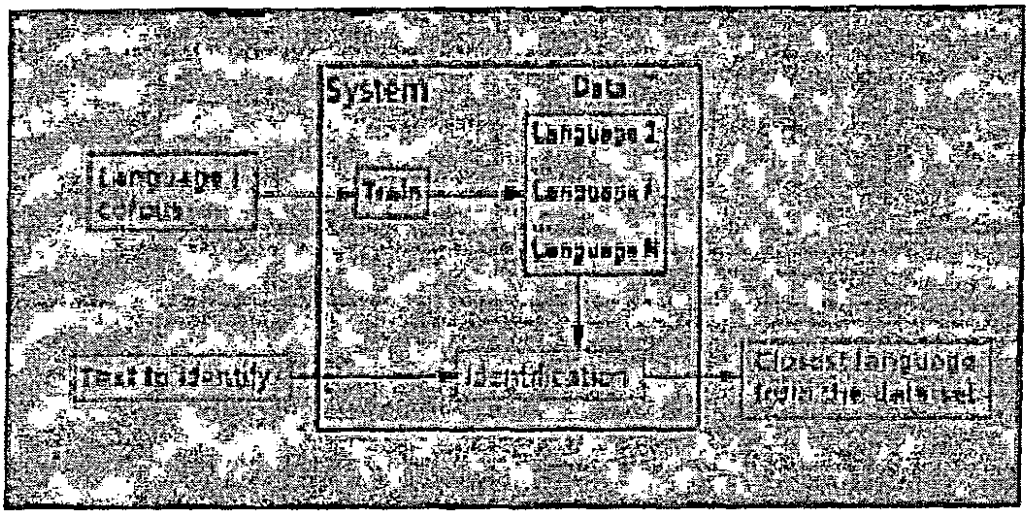


Figure 2.1 General Architecture of the System

Markov Models

Hidden Markov Models (HMM) are commonly used in spoken language identification. For each language that the system must know about, a model is trained from text corpora, and stored for later comparison with unidentified text. In these models each state represents a character trigram. Thus, the parameters of the MM are the transition probability and the initial probability.

Trigram Frequency Vectors

The trigram frequency vectors technique consists in comparing a vector of trigram frequencies for the text to classify with the vectors of known language, and select the closest one. Trigrams are formed by three consecutive characters of the text.

Gram Based Text Categorization

This technique is a text categorization method that can be applied to language identification, where each category is a language. The implementation of this technique is named TextCat. The system is based on comparing n-gram frequency profiles. A-gram frequency profile is a list of the occurring-grams sorted in decreasing frequency order. For each language we want to train the system, we create its -gram profile using all the -grams for all values of from 1to 5.

Martin Wechsler's, Paraic Sheridan's and Peter scauble's paper [16] is based on the SPIDER information retrieval system. Issues associated with indexing multilingual collection of information are addressed. The main focus is on the language identification and the use of stemming algorithm from the European countries. The search also focuses on multilingual intranet which contains documents in English, French, German and Italian. The work done include automatic identification of the language in which a particular text is written and the use of stemming algorithm for each individual language. The correct

identification of the language is crucial so that the correct stemming algorithm can be applied to each document. There is provision of user –friendly querying environment for a large multilingual collection of documents and special attention is made to language independent words.

The indexing features are based on the individual words and the size of the index is maintained on the final performance of the retrieval system. The first step in reducing the number of features to be included in the index usually involves discarding those words that have little or no value in representing the content of a document-referred to as *Stop Word.* The second step is taken to improve the set of features used to represent the content of documents, called N*ormalization.* In it the words from their surface are formed into a common base form. The emphasis is made on the presence of stop-words.

During indexing, language specific accumulators count the number of stop words that match against the stop-word list of each language. The language of the document or text passage is then assigned according to the accumulator with the maximum value.

This paper also give the idea of stemming to improve the performance and efficiency because the word reduction do not need to be linguistically meaningful since they are used only as indexing feature by retrieval system and not presented to the user at any stage. The crucial step to the performance of the stemming algorithm is the maintenance of the lexicon.

2.2 Our Work

In this project we have used three approaches for language detection. The description of these approaches is given below.

2.2.1 Dictionary Based Language Identification

Dictionary based identification (DBI) is one of the most tried and implemented methods for language identification. Though simple, its effectiveness in certain areas cannot be denied. One of its biggest advantages is its performance. DBI is much faster during training and detection than other algorithms used in this research. DBI gives good results when comparing languages belonging to different scripts. If used wisely this technique becomes more useful than it seems at first glance. Joined with Unicode based inter-language document identification, to detect and extract multiple languages data chunks from a document; dictionary based identification is applied on these chunks individually.

Like most techniques used in this research, dictionary based identification is also a two phase process. The first phase being training and the second identification. Before going on to identifying documents, the system must first be trained for the languages that we wish to detect. Remember, it only detects the languages it has been trained for. However, if a language belongs to a script for which another language has already been trained is found in an input document, that language is identified as the language that was trained for the same script. For example, say in the Arabic script, we have trained DBI for Arabic, Urdu and Persian languages. If we try to detect a Pushto, Punjabi or similar document, all Arabic, Urdu and Persian will come up as close matches giving a hint to the script the language belongs to.

The drawback in dictionary based implementation is that it doesn't perform well when comparing between languages belonging to the same script. Also document containing too few characters give much less information to DBI to detect languages effectively. Also, in case of documents like web pages containing multiple languages, the results are not able to clearly distinguish the language. However, if one language occupies the majority portion of the document, it has higher match weight-age hinting its identification.

Algorithm:

Training Phase:

Inputs: 1) Training Document
2) Dictionary File Path

Output: Unique characters list

- Open training document.
- Read all data from the document.
- Generate a list of unique characters present in the training document.
- Sort the unique characters list.

- Dump the list into the dictionary file specified.

Identification Phase:
Input: 1) Document to be identified

Output: 1) Language match count

- Open input document and read all data.
- Generate a list of unique characters in the document.
- Sort the generated list.
- Get a list of script folders in the lang_dicts folder
- for each script_folder in lang_dicts folder:
  - Get a list of language files in the script folder
  - For each language:
    - Load the language list of unique characters
    - Get match count of characters present in document list that are also present in language list.
- Display match count statistics for each language.

The language unique character lists are stored in a file folder hierarchy as displayed below:

- lang_dicts/ (root folder)
  - script folder 1/
    - language1 list file
    - language2 list file
  - script folder 2/
    - language 3 list file

2.3 Unicode Based Inter-Document Language/Script Identification

The use of Unicode in our research has proved to be very useful. The global acceptance of Unicode and the well thought out placement of different languages in the Unicode code-set has given us more opportunities to detect languages effectively. Each script/language in Unicode has defined code ranges. This information is provided in the form of Unicode Database giving type, language, code point and other info for every character of every language represented in Unicode. The use of this database enabled us to detect different scripts in a document by querying the Unicode database for info about any character.

Unicode based identification is the only technique in our research that does not require an explicit training phase. Only using updated versions of the Unicode database provided at www.unicode.org is sufficient for improving the technique.

This technique is the fastest technique for language detection that we have implemented. As added benefit, this technique also allows us to identify different script portions within a document. Like all, this technique also has its drawbacks. First, for those scripts that have

many languages in them (Roman, Arabic, etc.) this technique can only detect the script and not the language. Secondly for inter-document language identification, detection of different language chunks is effective if consecutive language chunks belong to different scripts. For example, it accurately separates Arabic and English language chunks repeating one after another but has problems if English and French or German language chunks start repeating after one another.

Algorithm:

Input: 1) Input Document

Output: List of languages and their byte ranges in the input document.

- Read all data from input document.
- set current_script = ""
- For each character in data:
    - Get Unicode category of the character (Letter, Digit, Punctuation, etc.)
    - If category = Letter
        - Get Unicode character name
        - Get the script name portion from the character name
        - if script_name != current_script:
            - Add last script to language chunks with start and end byte positions
            - set current_script = new script name
- Add last script to the language chunks list
- Remove first empty chunk from the list.

2.4 Cavnar's and Trenkle's Algorithm

C&T [1] algorithm is the most accurate algorithm around for language detection. This algorithm concentrates on alphabet-combination characteristics of languages. Because of this property, this algorithm excels where other algorithms fail. It more accurately identifies languages belonging to the same script.

This algorithm performs its calculations on n-grams. The value of n can be any digit (1,2,3, etc.) The number of n-grams in a document is equal to the number of characters in that document. For example, take the text "HELLO WORLD". 1-grams of this text are: 'H', 'E', 'L', 'L', 'O', ' ', 'W', 'O', 'R', 'L', 'D'. 2-grams for the same text are: 'HE', 'EL', 'LL', 'LO', 'O ', ' W', 'WO', 'OR', 'RL', 'LD', 'DH'. The advantage of using n-grams is that they highlight the language properties. Finding common alphabetic combinations of a language is the purpose of processing n-grams.
Over the years modifications and filtering techniques have been applied to further improve

the performance of this technique. The most recent successful addition to the technique was by Katia Hayati [1] in June, 2004, which included Fisher Discriminant function to give more importance to n-grams that were more unique across languages. We have further implemented another modification to the process by that serves as a replacement of Fisher discriminant function which was too time consuming for n-grams with n greater than 3.

The training phase for this technique spans across multiple steps that are:

- Document filtering to remove formatting information from documents.
- Document normalization for removing multiple white spaces.
- Training multiple documents per language so that sufficient training data per language is present to make the calculations more real.
- Sorting the resulting statistics and calculating top C n-grams according to weight for each language. In this research C=1000 was found to be a good value.

The identification phase involves generating n-grams for the input document and then comparing top C n-grams from the document again top C n-grams for each language to find out match value for that language.

By default the algorithm runs with our custom modifications (called FF modifications here) included. Since comparing the statistics against fisher discriminant implementation was also required to get validated results, so a fisher discriminant implementation of the algorithm is also provided.

Drawbacks of this method involve slower detection and more time required to train documents. Additionally adding a new language to the corpus of languages means doing all the calculations again for each language.

Algorithm (with FF modification):

Training (Step 1):

Inputs: 1) Language Name
2) Training Document

Output: NILL

- If the lang folder not exists inside the data folder
  - Load previously trained data for n-grams (n=1-4) from the data files present in the lang folder into n-gram lists.
- If lang folder does not exist:
  - Create lang folder
  - Initialize n-grams lists to blank
- Read all data from training document
- Normalize the data by replacing multiple white spaces with a single space.
- Generate n-grams from input data.

- Calculate weights (frequency) for each n-gram in input document.
- Update weight of each n-gram in the n-gram lists (loaded from previous training data files).
- Dump n-grams lists to lang folder inside the data folder.

Training (Step 2):

Inputs: 1) Source n-gram list
2) Target n-gram list
3) C (Top no. of n-grams to store)

Output: NILL

- Load Source n-gram list.
- Sort the list in descending order.
- Trim the list to top C items.
- Dump the new list to Target n-gram list path.

Training (Step 3) - n-gram Uniqueness Calculation:

Input: 1) Lang

Output: NILL

- Load n-gram lists from lang folder in the data_finalizer folder.
- Get list of languages in data_finalizer folder.
- For each n-gram in n-gram lists:
  - set u_ng = C * total_langs
  - For each lang:
    - if n-gram found in lang list:
      - idx = index of n-gram in current lang list.
      - u_ng = u_ng -idx
  - Add uniqueness value of n-gram to uniqueness list
- Dump n-gram uniqueness lists to lang folder.

Identification phase:

Input: 1) Input Document

Output: Language match count

- Read the data from document
- Normalize the data.
- Generate n-grams(1-4) for the data.
- Calculate n-gram weights (Frequencies) in data.

- Sort document n-grams list in descending order.
- Trim list to top C items.
- Get list of language folders in the data_finalizer folder.
- For each language:
    - Load lang n-gram lists.
    - Load n-gram uniqueness values.
    - For each document n-gram list (n= 1-4):
        - set diff = 0
        - For each n-gram in list:
            - If n-gram found in lang list:
                - set i1 = index of n-gram in doc list
                - set i2 = index of n-gram in lang list
                - diff += i1 - i2
                - uv = unique-ness value of n-gram
                - diff -= (uv/100)
            - if n-gram not found in lang list
                - diff += C+1
                - uv = unique-ness value of n-gram
                - diff -= (uv/100)
- return/display match statistics

Folder structure used in this algorithm is given below:

- data/
    - lang1/
        - 1grams.dat
        - 2grams.dat
        - 3grams.dat
        - 4grams.dat
    - lang2/
        - 1grams.dat
        - 2grams.dat
        - 3grams.dat
        - 4grams.dat

    .
    .
    .
- data_finalizer/
    - lang1/
        - 1grams.dat
        - 2grams.dat
        - 3grams.dat
        - 4grams.dat
        - u_1grams.dat

    - u_2grams.dat
    - u_3grams.dat
    - u_4grams.dat
- lang2/
    - 1grams.dat
    - 2grams.dat
    - 3grams.dat
    - 4grams.dat
    - u_1grams.dat
    - u_2grams.dat
    - u_3grams.dat
    - u_4grams.dat

Algorithm (Fisher Discriminant):

Training Phase:

Pre-condition: All training documents are present in training_documents folder.

- Generate a list all unique n-grams found in all training documents of each language. Lets call this list A.
- Get a list of language in the training_documents folder.
- For each lang:
    - Get a list of training docs in that lang.
    - For each doc:
        - For each n-gram in A:
            - Find frequency of n-gram in doc.
            - Find normalized frequency of n-gram in doc.
- For each n-gram in A:
    - Calculate mean frequency of n-gram
- Sort the list of mean frequencies in *descending order of frequency.*
- Trim this list to top 1000 items. Lets call this list R.
- For each lang:
    - Calculate lang fisher values
        - For each n-gram in A:
            - Calculate lang mean frequency.
        - For each n-gram in R:
            - fR = mean frequency of n-gram in R.
            - fL = mean frequency of n-gram in current lang.
            - n-gram fisher value = FL / fR
- Save lang fisher values.

Identification Phase:

Input: Input Document

- Read document data.
- Normalize data.
- Get a list of unique n-grams present in the document.
- For each n-gram in document n-grams:
    - calculate frequency of n-gram in document.
    - Load R (Top thousand n-grams by mean frequency, generated during training phase).
    - For each n-gram in R:
        - If n-gram present in document n-grams:
            - calculate n-gram normalized frequency in document
        - else:
            - set normalized frequency of n-gram in doc to 0.00.
- Get a list of Languages available for identification
- For each lang:
    - Load lang fisher values.
    - set diff = 0.00
    - For each n-gram in document normalized frequencies list:
        - set nFV = lang fisher value for ngram
        - set nNF = document normalized frequency for ngram
        - diff += nNF * nFV
    - Display diff as lang match value for document.

CHAPTER 3

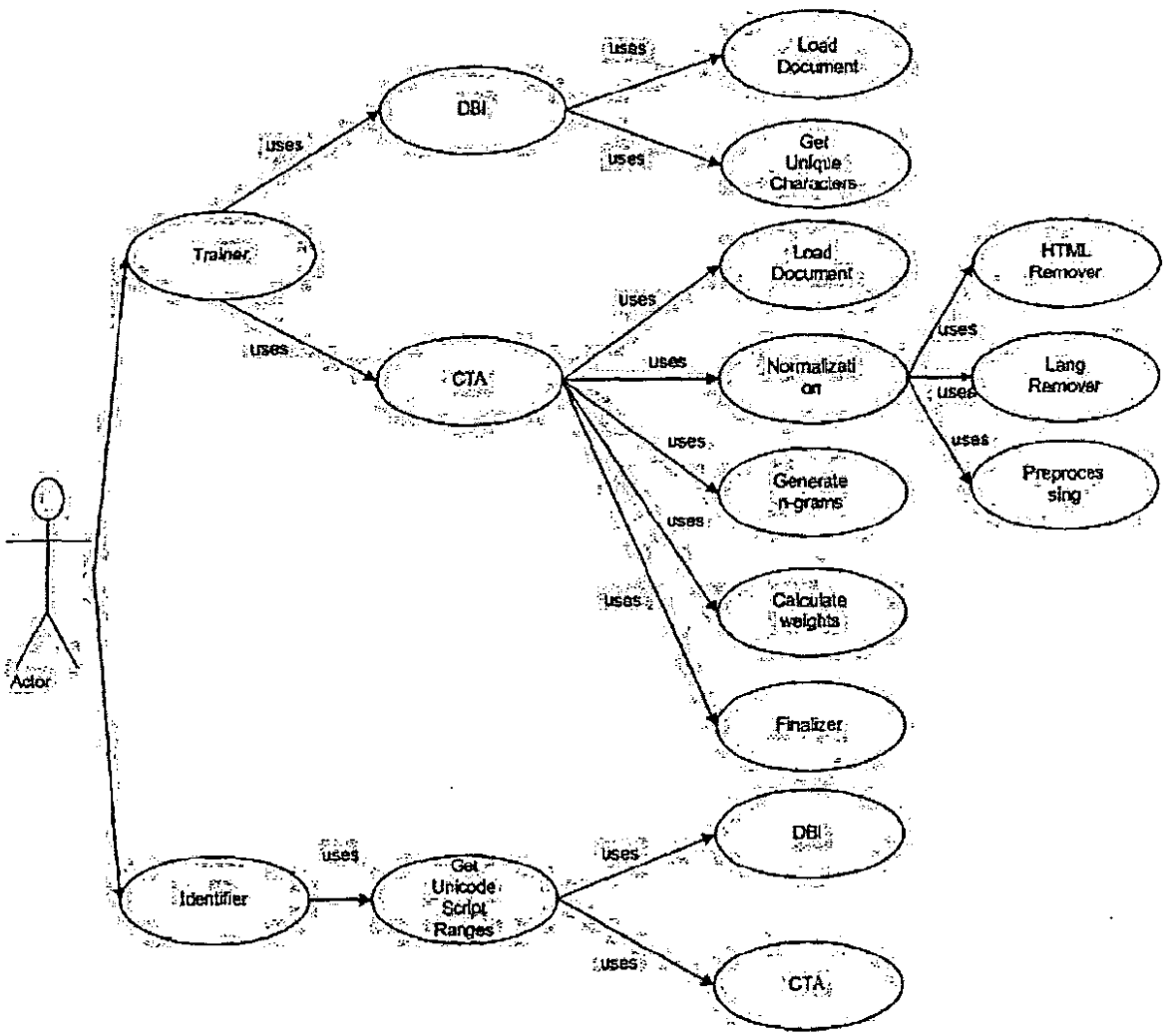
REQUIRENMENT ANALYSIS

3. Requirement Analysis

The requirement analysis is the first step towards developing software. Analysis must be performed in a systematic and correct manner so as to have as few mistakes as possible in the software and to have an end product completely fulfilling the expectations of the client. The main objective of this phase is to identify all possible requirements and expectation of software.

3.1 Use Case Analysis

Analysis of the project is represented in terms of Use Case diagrams indicating the actors and use case in expanded format. Use Case may be related to the other Use Cases by the Extended, Include, and Generalization relationships. The use case model describes the proposed functionality of the new system.



3.2 Use Case in Expanded Format

In this section we discuss the use cases by describing their related actors, pre and post conditions, typical course of actions and alternative course of actions.
The use cases have been written in expanded format as follows.

3.2.1 Trainer

Uc1 Dictionary Based Method
Uc2 Load Document
Uc3 Get Unique Characters
Uc4 Cavnar & Trenkle's Algorithm
Uc5 Load Document
Uc6 Normalization
Uc7 HTML Removal
Uc8 Language Removal
Uc9 Preprocessing
Uc10 Generate n-grams
Uc11 Calculate Weights
Uc12 Finalizer

3.2.2 Identifier

Uc13 Get Unicode Script ranges
Uc14 DBI
Uc15 CTA

3.2.1.1 Trainer Use Case

a) Name: Trainer
b) Actor: Admin
c) Pre-Condition: None
d) Post-Condition: Runs the selected method for training
e) Typical Course of Action:

Actor Action	System Response
1) User Selects the method for training.	2) Call and runs the specified method.

f) Alternate Course of Action:

Actor Action	System Response
1a) If user doesn't select any method for training	2a) Default method is called.

3.2.1.2 DBI Use Case

a) Name: DBI
b) Actor: Admin
c) Pre-Condition: Input document must be specified and the target file path where the dictionaries are to be saved. Input Document must be in the pdf, HTML, txt, doc and in UTF-8 encoding
d) Post-Condition: Dictionaries are saved in specified path.
e) Typical Course of Action:

Actor Action	System Response
1) Input document is loaded(pdf,txt,doc,HTML format,UTF-8 encoding)	2) List of unique characters are generated for that document. 3) Dictionaries are saved in the specified path.

f) Alternate Course of Action:

Actor Action	System Response
1a) If the document is not in specified format or UTF-8 encoding. 4a) Repeat step 1 to 3	2a) Document is not loaded. 3a) Error message is displayed.

3.2.1.3 CTA Use Case

a) Name: CTA
b) Actor: Admin
c) Pre-Condition: Input document must be specified. Input Document must be in the pdf, HTML, txt, doc and in UTF-8 encoding
d) Post-Condition: Saves list of n_grams and update their weights, if no list exist creates a list.
e) Typical Course of Action:

Actor Action	System Response
1) Input document is loaded (pdf, txt,doc,HTML format,UTF-8 encoding)	2) Removes HTML tags. 3) Removes the multiple spaces, \t,\n and any other special characters. 4) Removes the specified characters (may belong to other scripts) from the document. 5) Returns only the textual data after

	normalizing. 6) Generate all the n-grams (1 to 4 gram). 7) Calculates the weights for n-grams. 8) Reduce n-grams up to the value of C(1000).

f) Alternate Course of Action:

Actor Action	System Response
1a) If the document is not in specified format or UTF-8 encoding. 4a) Repeat step 1 to 3	2a) Document is not loaded. 3a) Error message is displayed.

3.2.2.1 Identifier Use Case

(A)

a) Name: Identifier
b) Actor: User/Admin
c) Pre-Condition: Input document in specified format and in UTF-8 encoding,
d) Post-Condition: Language identification results displayed using CTA
e) Typical Course of Action:

Actor Action	System Response
1) User Selects the CTA method for identification.	2) Generates n-gram list for the input document. 3) Loads n-grams list from data Finalizer folder. 4) Compares the two lists. 5) Display language identified and its percentage count.

f) Alternate Course of Action:

Actor Action	System Response
1a) If user doesn't select any method for identification	2a) Default method is called.

3.2.2.2 Identifier Use Case

(B)

a) Name: Identifier
b) Actor: User/Admin
c) Pre-Condition: Input document in specified format and in UTF-8 encoding,
d) Post-Condition: Language identification results displayed using DBI

e) Typical Course of Action:

Actor Action	System Response
1) User Selects the DBI method for identification.	2) Generates list of unique characters for the input document. 3) Loads Dictionaries from data Finalizer folder. 4) Compares dictionaries with list of unique characters. 5) Display script, language identified and percentage count.

f) Alternate Course of Action:

Actor Action	System Response
1a) If user doesn't select any method for identification.	2a) Default method is called.

CHAPTER 4

SYSTEM DESIGN

4. DESIGN

System design is the specification or construction of a technical, computer based solution for business requirements identified in the system analysis. It is evaluation of alternative solutions and the specification of a detailed computer based solution.

4.1 Modular Approach

The whole system is implemented as a set of python modules. This approach allows both use of individual functions or the whole application. Most of the modules when run individually either allow the user to perform one specific task or run some test cases to test the code they contain. When imported into other python programs they allow usage of all the functionality the system provides.

4.1.1 User Interfaces

Apart from being a code library that can be used in bigger systems requiring language identification two user interfaces have been built using this library.

4.1.1.1 Command Line Interface (CLI)

The command line interface allows language detection from the command line. This interface is ideally suited for quick language detection tasks, automated batch jobs for doing language detection on a bulk of documents etc. The results returned are machine parse-able (GREP-able). This is in-line with the traditional Unix philosophy of allowing programs to be easily glued together for accomplishing bigger tasks.

4.1.1.2 Web Interface

To maximize the availability of the system, a web based interface for the system has also been implemented allowing the user to upload a document and have its language detected. This interface is built upon the idea of parsing the output of the CLI and displaying them in a more user friendly manner.

4.2 Sequence Diagram

Sequence Diagrams are used to show the flow of functionality through a use case. It illustrates the entire flow of processing object and actor interaction with respect to time. Sequence Diagrams of some use cases are discussed below:

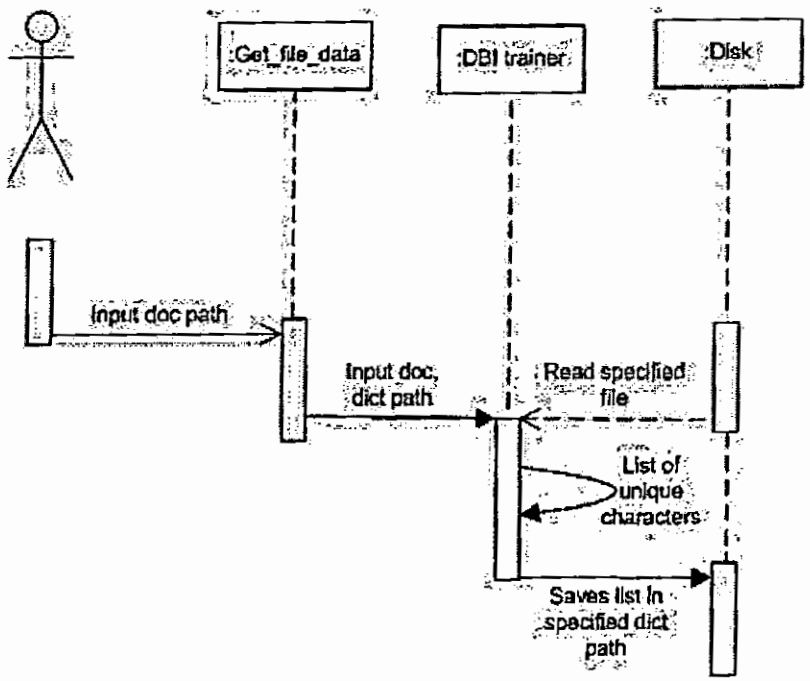


Figure 4.1 Sequence Diagram of DBI (Trainer)

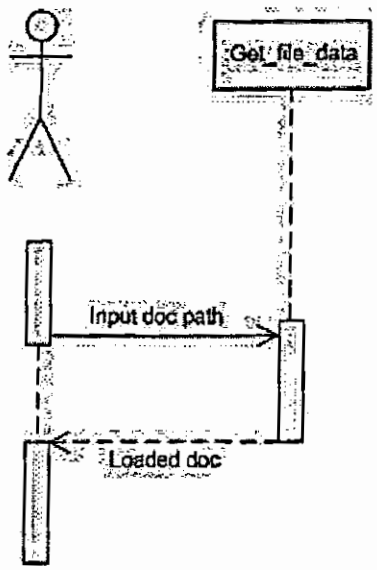
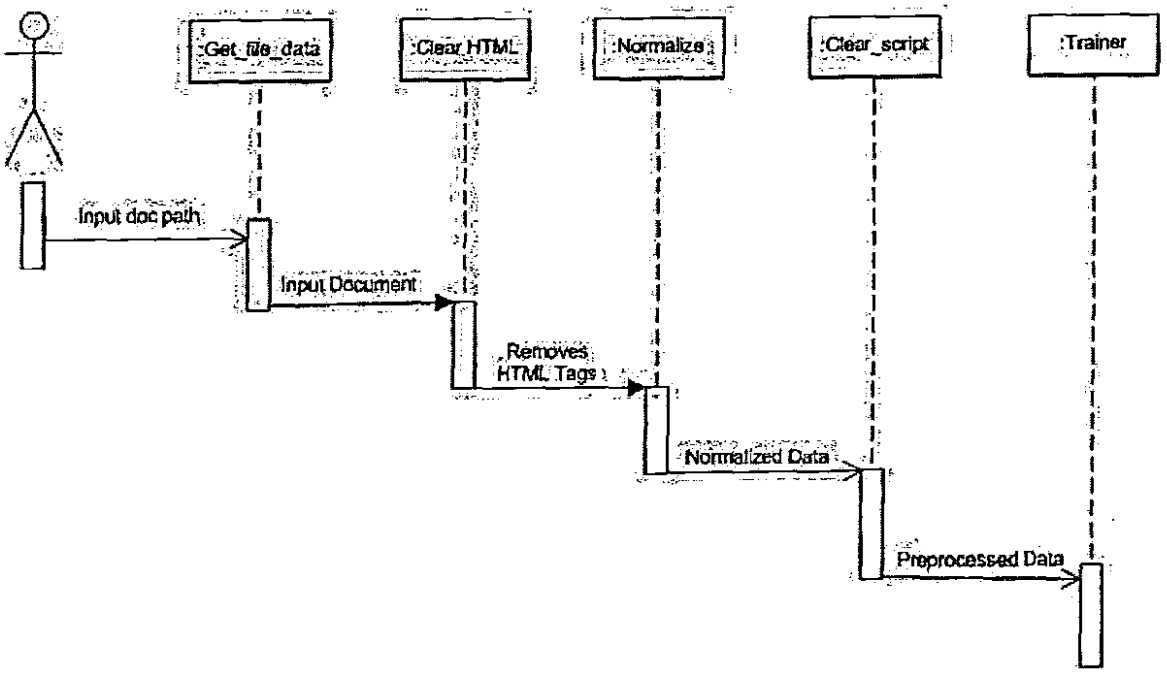


Figure 4.2 Sequence Diagram of Load Document (CTA-Trainer)



4.3 Sequence Diagram of Normalization (CTA-Trainer)

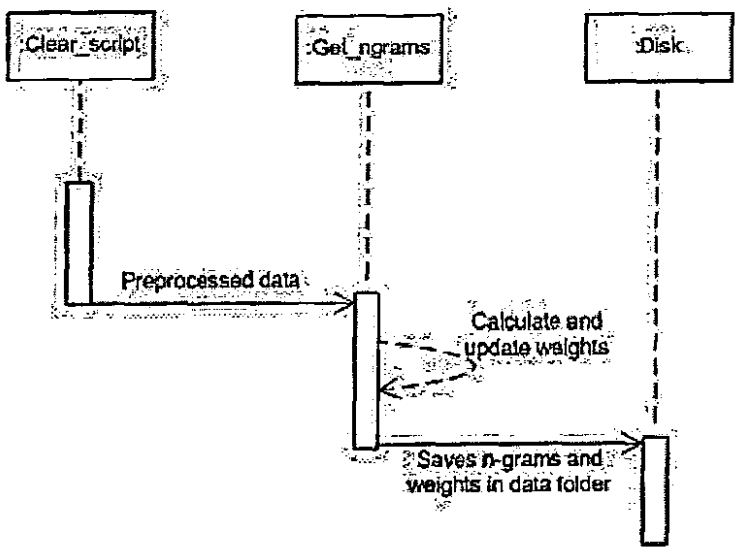


Figure 4.4 Sequence Diagram of Generate n-grams(CTA-Trainer)

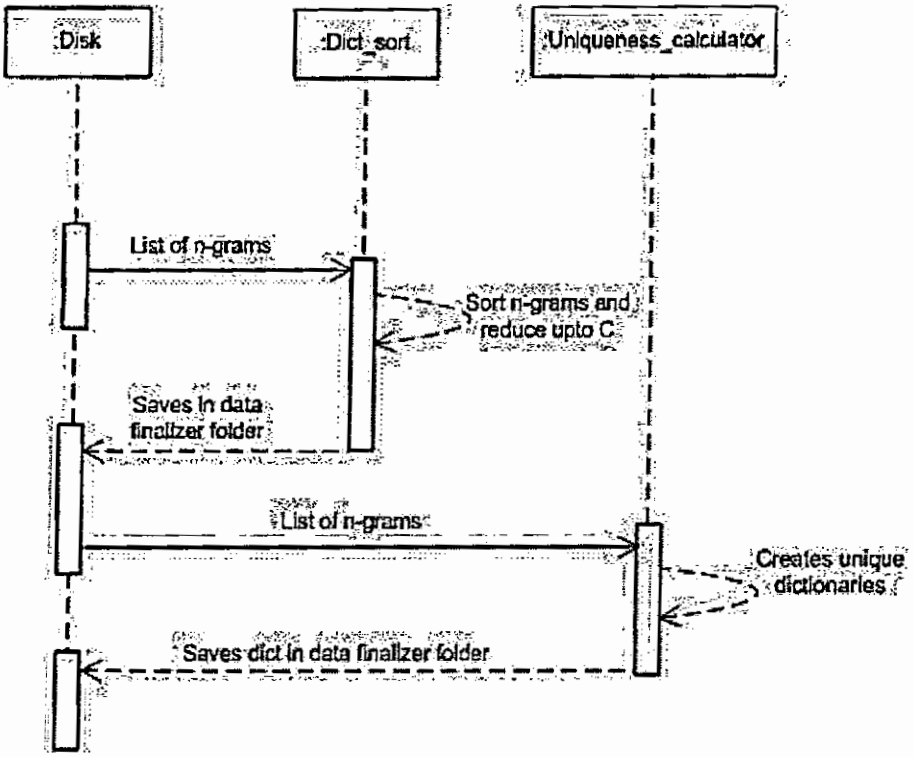


Figure 4.5 Sequence Diagram of Finalizer (CTA-Trainer)

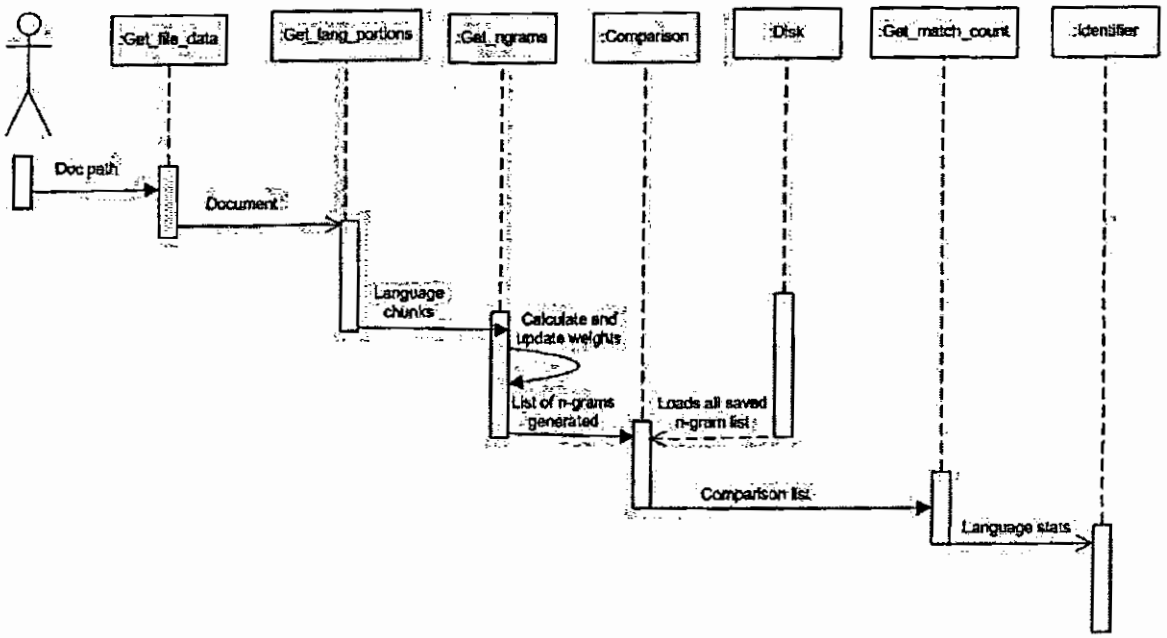


Figure 4.6 Sequence Diagram of CTA Identifier

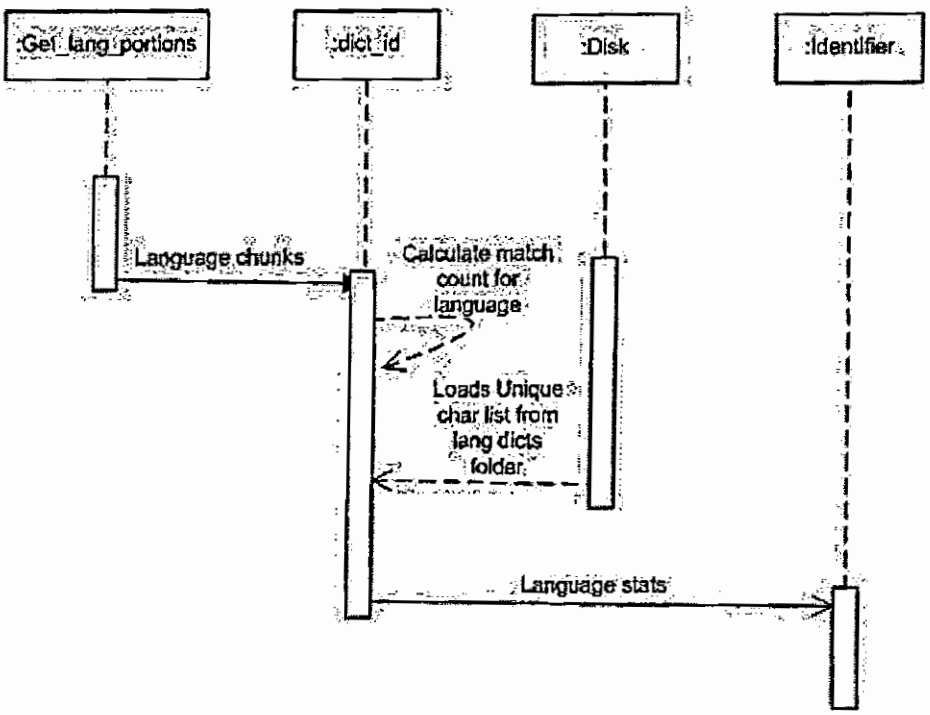


Figure 4.7 Sequence Diagram of DBI Identifier

CHAPTER 5

IMPLEMENTATION

5. IMPLEMENTATION

Web interface for the system has been developed using PHP & HTML. All other components of the system have been developed using Python.

The modules in the system can be divided into the following categories:

5.1 Unicode and Utility Modules
5.2 Dictionary Based Modules
- Training Modules
- Identification Modules

5.3. Cavnar's and Trenkle's Algorithm modules
- Training Modules
- Identification Modules

5.1 Unicode and Utility Modules

5.1.1 Inter-Document Multiple Script Identification (inter_lang_id.py)

This module is used to detect multiple scripts within a single document. Given the data it returns chunks of each script.

Detecting and Returning multiple scripts within the same data:

```
def get_lang_portions(data):

  big_list = []

  L = []
  i=0
  Li=0
  current_script=''
  Si=0
  script = ''

  #print repr(data)

  for ch in data:
    cat=unicodedata.category(ch)
    #print "cat: %s" % (cat)


    if cat[:1] == 'L':

      Li= i
      #print "ch Name: " + unicodedata.name(ch)
      script = unicodedata.name(ch)
      script=script.split()
```

```
        script = script[0]
        #print "Script: %s, i: %i, Li: %i" % (script, i, Li)
        if current_script != script:

      #print "IN IF: Current Script: %s, script: %s" %
(current_script, script)

          L.append(current_script)
          L.append(Si)
          L.append(Li)


          big_list.append(L)

       L=[]
       Si=Li
       current_script=script

     i += 1

  L.append(current_script)
  L.append(Si)
  L.append(Li)

  big_list.append(L)

  L=[]
  Si=Li
  current_script=script
  del big_list[0]
  return (big_list)
```

5.1.2 Document Conversion (data_conversion.py)

This module is used to read various forms of input documents and return their data, ignoring any formatting and other info they may contain. Type of the document is detected using its file extension and data from the document is ready accordingly.

Reading data from different types of documents:

```
def get_file_data(filename):
     fileparts = filename.rsplit(".",1)
     file_ext = fileparts[1]
     #print repr(file_ext)
     file_ext = file_ext.lower()
```

```
    if "htm" == file_ext or "html" == file_ext:
            fd = codecs.open(filename,'r',encoding = 'utf-8')
            data = fd.read()
            fd.close()
            data = html_cleaner.clear_html(data)

    if "txt" == file_ext or "text" == file_ext:
            fd = codecs.open(filename,'r',encoding = 'utf-8')
            data = fd.read()
            fd.close()
    if "pdf" == file_ext:
        cmd = "pdftotext %s %s" % (filename, filename + '.txt')
         #print cmd
         os.system(cmd)

         fd = codecs.open(filename + '.txt','r',encoding =
'utf-8')
         data = fd.read()
         fd.close()
         os.system ("rm %s" % (filename + '.txt'))
    return data
```

5.1.3 HTML Cleaner (html_cleaner.py)

This module is used to clear html tags, comments etc from html documents. Regular expressions are used to clear off any html tags from the document. Any javascript and style information present in the document is also removed

Clearing HTML formatting information from a document:

```
def clear_html(d):
  #define the regular expression to be matched
  r =
re.compile("<script.*?>.*<\/script>|<style.*?>.*<\/style>|<\
/?\w.*?>|<\!--|-->| ", re.M + re.S + re.I + re.U)
  d = r.sub("",d)
  return d
```

5.1.4 Document Cleaner (doc_cleaner.py)

This module is used to remove alphabets of one language from documents of another language. Its used in "purifying" the training documents before using them for training. For example, using this module Roman alphabets can be removed from non-Roman languages like Urdu, Arabic, Chinese etc.

The data and a list of alphabets that need to be removed are passed to the function.

Removing alphabets of other scripts from a document:

```
def clear_script(d,list):
    d2 = ""
    for ch in d:

        if ch in list:
            #print repr(ch)
            d2 = d2 + ch
    return d2
```

5.1.5 File Folders Utilities (myutils.py)

This module is used for misc functions like getting a list of files and folders in a given path.

Function to get a list of files and folder in given path:

```
def get_folders_files(folder_name):
     directories = []
     files = []
     for r,d,f in os.walk(folder_name):
          if r == folder_name:
               directories = d
               files = f
     return (directories,files)
```

5.1.6 Normalization (Pre_Processing.py)

This module is used for normalizing input data by removing tab and new line characters etc. and converting multiple spaces to single space.

Function to normalize given data:

```
def normalize(d):
    d = d.replace("\n"," ")
    d = d.replace("\t"," ")
    d = d.replace("\r"," ")
    while -1 != d.find("  "):
         d = d.replace("  "," ")

    return (d)
```

5.2 Dictionary Based Modules

5.2.1 Dictionary Based Identification - Trainer (DBI_Trainer.py)

This module generates unique character lists from given input files. These lists are later used for dictionary based language identification.

unique list generation from input document :

```
dllist= []
fd = codecs.open(sys.argv[1],'r',encoding = 'utf-8')
unicode_string = fd.read()
for unicode_char in unicode_string:
      if unicode_char not in dllist:
            dllist.append(unicode_char)

dllist.sort()
```

5.2.2 Dictionary Based Identification - Identifier (DBI_Identifier.py)

This module performs dictionary based language identification. A List of unique characters in the input document is generated and is compared against each list of unique characters for supported languages.

Dictionary based language identification :

```
def dict_id(data):
      DL = Uni_char_doc(data)
      big_list=[]
      (folders,f) = myutils.get_folders_files('./lang_dicts')
      #print repr(folders)
      script_dict = {}
      for script_folder in folders:
            script_dict= {'name':script_folder}
            script_dict['langs'] = []
            (f,files) =
myutils.get_folders_files('./lang_dicts/' + script_folder)
           L=[]
           #print repr(files)
           for lang_file in files:
                 Li = cPickle.load(codecs.open('./lang_dicts/'
+ script_folder + '/' + lang_file,'r'))

                 L = [lang_file]
                 match_count = 0

                 for ch in DL:
```

```
                    if ch in Li:
                        match_count +=1

                L.append(match_count)
                script_dict['langs'].append(L)

            big_list.append(script_dict)

        return    big_list
```

5.3 Cavnar's and Trenkle's Algorithm Modules

5.3.1 N-Grams Generator (ngrams.py)

This module serves for generating specified n-grams (1-grams, 2-grams, 3-grams, etc.) from given data. This module is used at many occassions by modules of this category.

The n-gram generator function :

```
def getngrams(d,n=3):
    l=len(d)
    n = int(n)
    d=d+d[:n-1]
    list = []
    for i in range(l):
        list.append(d[i:i+n])
    return list
```

5.3.2 Trainer (trainer.py)

This trainer module is used to train the system for more language or to further improve the learning of the system for already trained languages. It takes two arguments, a language name and an input document. If the language has previously been trained then previous stats are loaded and after processing the new document stats are updated and stored again. If the language has not been previously trained, new stats are generated and stored.

1-gram training code snippet :

```
if os.path.isdir("./data/" + sys.argv[1] ):
          dict_1gram = cPickle.load(codecs.open("./data/" + sys.argv[1] +
"/1grams.dat",'r'))

            dict_2gram = cPickle.load(codecs.open("./data/" + sys.argv[1]
+ "/2grams.dat",'r'))

          dict_3gram = cPickle.load(codecs.open("./data/" + sys.argv[1] +
"/3grams.dat",'r'))

          dict_4gram = cPickle.load(codecs.open("./data/" + sys.argv[1] +
"/4grams.dat",'r'))

     else:
          os.mkdir("./data/" + sys.argv[1] )
          dict_1gram = {}
          dict_2gram = {}
          dict_3gram = {}
          dict_4gram = {}

       for filename in sys.argv[2:]:
           print "Reading File: " + filename
           starttime = time.time()
           filename = codecs.open(sys.argv[2],'r', encoding = 'utf-8')
           d = filename.read()
           filename.close()
         endtime = time.time()
           difference = endtime - starttime
           print "Read Time : " + str(difference)

           print "Normalizing: "
           starttime = time.time()
           d = Pre_Processing.normalize(d)
         endtime = time.time()
           difference = endtime - starttime
           print " time: " + str(difference)


           print "Getting ngrams: "
           starttime = time.time()
           ng = ngrams.getngrams(d,1)
         endtime = time.time()
           difference = endtime - starttime
           print " time: " + str(difference)
```

```
            one_grams = {}
            print "Calculating Weights: "
            starttime = time.time()
            for n in ng:
                if one_grams.has_key(n):
                    one_grams[n]+=1
                else:
                    one_grams[n]=1

            endtime = time.time()
            difference = endtime - starttime
            print " time: " + str(difference)

          for k in one_grams:
            if dict_1gram.has_key(k):
                dict_1gram[k]=one_grams[k] + dict_1gram[k]
            else:
                dict_1gram[k]=one_grams[k]

            cPickle.dump(dict_1gram,codecs.open("./data/" + sys.argv[1] +
"/1grams.dat","w"))
```

5.3.3 Training Finalizer (training_finalizer.py)

After enough training documents have been used to train for languages, the training finalizer module is used to sort the stats and trim them to the top C (c=1000 used) most frequent n-grams for each language. The results of this module are used for language identification by the identifier module.

The training finalization phase :

```
dict = cPickle.load(codecs.open(sys.argv[1],'r'))
#        print repr(dict)
         list = dict_sort(dict,'d')
         list = list[:int(sys.argv[3])]
#         print repr(list)
         cPickle.dump(list, codecs.open(sys.argv[2], "w"))
```

5.3.4 Uniqueness Calculator (uniqueness_calculator.py)

This module trains the system for an alternate improvement to Fisher Discriminant function. For each n-gram in every language, its uniqueness value is calculated. N-grams that are more unique across languages are given higher weight-age during language detection.

Calculating uniqueness of 4-grams :

```
lang_folders = myutils.get_folders_files("data_finalizer")
```

```
lang_folders = lang_folders[0]
folder_count = len(lang_folders)

for ng in L_4gram:
      Ung = C * folder_count

for l in lang_folders:
      l_4gramlist = cPickle.load(codecs.open('./data_finalizer/' + l +
'/4grams.dat','r'))

      if ng in l_4gramlist:
            index = l_4gramlist.index(ng)

            Ung = Ung - index


      unique_dict[ng] = Ung

cPickle.dump(unique_dict,codecs.open("./data_finalizer/" + lang +
"/u_4grams.dat","w"))
```

5.3.5 Stats Comparison Module (caomparision.py)

This module compares input document stats against uniqueness and frequency stats of a language and returns the results. It is used by the language identified modules for comparing languages for a given doc.

Comparing n-gram stats :

```
def compare_ngrams(G,Gi,Ui,C):
  similarity_list = [0,0,0,0]

  for cn in range(0,4):
    list1 = G[cn]
    list2 = Gi[cn]
    U_dict = Ui[cn]
    current_difference = 0
    for ngrams in list1:
      if ngrams in list2:
        index1 = list1.index(ngrams)
        index2 = list2.index(ngrams)
        current_difference = current_difference + (index1 - index2)
        if U_dict.has_key(ngrams):
          u_value = U_dict[ngrams]
          current_difference = current_difference - (u_value/100)

      else:
        current_difference = current_difference + (C+1)
        if U_dict.has_key(ngrams):
          u_value = U_dict[ngrams]
          current_difference = current_difference - (u_value/100)
```

```
    #current_difference = current_difference / 100
    similarity_list[cn] = current_difference

  return(similarity_list)
```

5.3.6 CTA Identifier (CTAlgo_detector.py)

This module performs language detection for the C&T algorithm. It takes an input document and prints language match percentages for that document.

code snippet of matching by one_grams:

```
def get_match_count(unicode_string):
      C = 1000
      #return normalized data
      #print "Time For Normalizing :"
      #start_time = time.time()
      normalized_doc = Pre_Processing.normalize(unicode_string)
      #endtime = time.time()
      #difference = endtime - start_time
      #print " Time for Normalization: " + str(difference)

      #Return All the n-grams of normalized data
      G = []
      #print "Getting 1grams: "
      #starttime = time.time()
      ng1_doc = ngrams.getngrams(normalized_doc,1)
      #endtime = time.time()
      #difference = endtime - starttime
      #print " Time for all 1-grams: " + str(difference)

      one_grams = {}
      #print "Calculating Weights: "
      #starttime = time.time()
      for n in ng1_doc:
          if one_grams.has_key(n):
            one_grams[n]+=1
          else:
            one_grams[n]=1
```

5.3.7 Fisher Discriminant (fisher_discriminant.py)

This module performs language detection by using C&T algorithm along with the Fisher Discriminant Improvement. It takes an input document and prints language match percentages for that document.

This module is implemented as one class that performs all operations for fisher discriminant based C&T algo language detection. Every thing from training to detection is performed by the same module.

calculating mean frequencies:

```
 def calculate_mean_freqs(self):
      self.load_all_ngrams()
      self.load_all_normalized_freqs()
      total_ngrams = len(self.all_ngrams)
      print "\nCalculating mean frequencies for all ngrams"
      print "Total N-grams: %i" % total_ngrams
      current_count = 0
      st = time.time()

      mean_freqs = {}
      for ngram in self.all_ngrams:
         current_count += 1
         f = self.get_mean_freq(ngram)
         mean_freqs[ngram] = f
         if 0 == current_count % 100:
            ct = time.time()
            elapsed_time = ct - st
            remaining_ngrams = total_ngrams - current_count
            remaining_time = (elapsed_time/current_count) *
remaining_ngrams
            print "calculating %i/%i, time elapsed: %i, estimated time
remaing: %i" % (current_count, total_ngrams, elapsed_time, remaining_time)

      return(mean_freqs)
```

CHAPTER 6

RESULTS

6. RESULTS

In this chapter various statistics generated and observations made during the system development process are presented.

6.1 Training Documents

Training documents are the very basis of successful language identification. Picking suitable training documents and cleaning them correctly before feeding to the system for learning is the most important part to ensure the accuracy of the system. While picking training documents following precautions were taken:

- Topic specific technical documents containing the same words or phrases again and again were not picked.
- All formatting information and languages other than the language for which the document would be used for training were removed.
- Total data size of training documents for each language was kept roughly equal to avoid any bias toward an heavily trained language.
- All documents were save in UTF-8 encoding.

Given below is the list of training documents used for each language along with their sizes

```
root@fouzia:/Project/training_documents# ls -lhR
.:
total 5.5K
drwxr-xr-x  2 root root 528 2005-12-26 14:07 arabic/
drwxr-xr-x  2 root root 208 2006-01-01 01:46 chineese/
drwxr-xr-x  2 root root  96 2005-12-26 13:42 english/
drwxr-xr-x  2 root root  80 2006-01-01 01:47 french/
drwxr-xr-x  2 root root 144 2006-01-01 01:49 german/
drwxr-xr-x  2 root root 208 2005-12-25 15:15 italian/
drwxr-xr-x  2 root root 176 2006-01-01 01:50 japaneese/
drwxr-xr-x  2 root root 488 2006-01-01 01:51 persian/
drwxr-xr-x  2 root root 208 2006-01-01 01:52 spanish/
drwxr-xr-x  2 root root 400 2005-12-26 13:57 urdu/

./arabic:
total 260K
-rw-r--r--  1 root root 1.7K 2005-12-26 14:05 arabic1.txt
-rw-r--r--  1 root root  25K 2005-12-26 14:06 arabic10.txt
-rw-r--r--  1 root root  31K 2005-12-26 14:06 arabic11.txt
-rw-r--r--  1 root root  51K 2005-12-26 14:06 arabic12.txt
-rw-r--r--  1 root root  26K 2005-12-26 14:07 arabic13.txt
-rw-r--r--  1 root root 5.8K 2005-12-26 14:07 arabic14.txt
-rw-r--r--  1 root root  18K 2005-12-26 14:07 arabic15.txt
-rw-r--r--  1 root root 3.2K 2005-12-26 14:05 arabic2.txt
-rw-r--r--  1 root root  30K 2005-12-26 14:05 arabic3.txt
-rw-r--r--  1 root root 5.8K 2005-12-26 14:05 arabic4.txt
-rw-r--r--  1 root root 4.0K 2005-12-26 14:06 arabic5.txt
-rw-r--r--  1 root root 9.1K 2005-12-26 14:06 arabic6.txt
-rw-r--r--  1 root root 6.1K 2005-12-26 14:06 arabic7.txt
```

```
-rw-r--r--  1 root root 9.9K 2005-12-26 14:06 arabic8.txt
-rw-r--r--  1 root root 7.2K 2005-12-26 14:06 arabic9.txt

./chineese:
total 312K
-rw-r--r--  1 root root  67K 2005-12-26 13:32 chineese2_clean.html
-rw-r--r--  1 root root  30K 2005-12-26 13:32 chineese3_clean.html
-rw-r--r--  1 root root  72K 2005-12-26 13:33 chineese5_clean.html
-rw-r--r--  1 root root 140K 2005-12-26 13:33 chineese6_clean.html

./english:
total 340K
-rw-rw-r--  1 root root 336K 2005-01-03 03:09 132.txt
-rw-r--r--  1 root root 2.4K 2005-11-11 06:08 new.txt

./french:
total 260K
-rwxr-xr-x  1 root root 258K 2005-12-25 15:15 french2.html*

./german:
total 304K
-rwxr-xr-x  1 root root  44K 2005-12-25 15:15 german2.txt*
-rwxr-xr-x  1 root root  83K 2005-12-25 15:15 german4.txt*
-rwxr-xr-x  1 root root 175K 2005-12-25 15:15 german5.txt*

./italian:
total 276K
-rwxr-xr-x  1 root root  45K 2005-12-25 15:15 italian1.txt*
-rwxr-xr-x  1 root root  59K 2005-12-25 15:15 italian2.txt*
-rwxr-xr-x  1 root root 114K 2005-12-25 15:15 italian3.txt*
-rwxr-xr-x  1 root root  17K 2005-12-25 15:15 italian4.txt*
-rwxr-xr-x  1 root root  29K 2005-12-25 15:15 italian5.txt*

./japaneese:
total 260K
-rw-r--r--  1 root root  35K 2005-12-26 13:54 japaneese1.txt
-rw-r--r--  1 root root  79K 2005-12-26 13:55 japaneese2.txt
-rw-r--r--  1 root root 106K 2005-12-26 13:55 japaneese3.html
-rw-r--r--  1 root root  36K 2005-12-26 13:55 japaneese5.html

./persian:
total 396K
-rw-r--r--  1 root root 28K 2005-12-26 13:46 persian_clean.txt
-rw-r--r--  1 root root 20K 2005-12-26 13:28 persian_clean1.txt
-rw-r--r--  1 root root 45K 2005-12-26 13:47 persian_clean10.html
-rw-r--r--  1 root root 32K 2005-12-26 13:29 persian_clean2.txt
-rw-r--r--  1 root root 41K 2005-12-26 13:29 persian_clean3.txt
-rw-r--r--  1 root root 49K 2005-12-26 13:29 persian_clean4.txt
-rw-r--r--  1 root root 33K 2005-12-26 13:30 persian_clean5.txt
-rw-r--r--  1 root root 34K 2005-12-26 13:30 persian_clean6.txt
-rw-r--r--  1 root root 17K 2005-12-26 13:30 persian_clean7.txt
-rw-r--r--  1 root root 51K 2005-12-26 13:45 persian_clean8.txt
-rw-r--r--  1 root root 28K 2005-12-26 13:45 persian_clean9.txt

./spanish:
total 228K
-rwxr-xr-x  1 root root  62K 2005-12-25 15:15 spanish1.html*
```

```
-rwxr-xr-x  1 root root  53K 2005-12-25 15:15 spanish2.html*
-rwxr-xr-x  1 root root  35K 2005-12-25 15:15 spanish3.html*
-rwxr-xr-x  1 root root  63K 2005-12-25 15:15 spanish4.html*
-rwxr-xr-x  1 root root 4.3K 2005-12-25 15:15 spanish5.txt*

./urdu:
total 212K
-rw-r--r--  1 root root  19K 2005-12-26 13:56 urdu1.txt
-rw-r--r--  1 root root  41K 2005-12-26 13:57 urdu10.txt
-rw-r--r--  1 root root  32K 2005-12-26 13:57 urdu11.txt
-rw-r--r--  1 root root 9.3K 2005-12-26 13:56 urdu2.txt
-rw-r--r--  1 root root 6.3K 2005-12-26 13:57 urdu3.txt
-rw-r--r--  1 root root  15K 2005-12-26 13:57 urdu4.txt
-rw-r--r--  1 root root 9.9K 2005-12-26 13:57 urdu5.txt
-rw-r--r--  1 root root  15K 2005-12-26 13:57 urdu6.txt
-rw-r--r--  1 root root  14K 2005-12-26 13:57 urdu7.txt
-rw-r--r--  1 root root  14K 2005-12-26 13:57 urdu8.txt
-rw-r--r--  1 root root  18K 2005-12-26 13:57 urdu9.txt
```

6.1.1 Training Languages and Scripts

10 languages belonging to 4 different scripts have been used to test out the system training and identification phases. The scripts are Arabic, Chinese, Roman/Latin, Japanese. The languages used are Arabic, Chinese, English, German, French, Italian, Japanese, Persian, Spanish and Urdu.

6.2 Test Environment

All statistics were gather by running the system on a 2.4 GHz P-IV system with 256 MB RAM. Operating system used is GNU Linux 2.4.31#6 (Distribution: Slackware 10.2). No processor intensive tasks were performed during the statistics gathering process.

6.3 Algorithm Wise Timing and Accuracy Statistics

Given below is are timing and accuracy statistics grouped by each algorithm used in the language. These serve as the facts that support our conclusions.

6.3.1 Dictionary Based Identification

Training for Dictionary Based Identification is very fast and new languages can be trained in a matter of seconds (even milliseconds).
A snippet of a training session of training a few languages for DBI is given below:

```
root@fouzia:/Project/src# ls -lh ../training_documents/english/132.txt
-rw-rw-r--  1 root root 336K 2005-01-03
03:09 ../training_documents/english/132.txt

root@fouzia:/Project/src#
time ./DBI_Trainer.py ../training_documents/english/132.txt
```

```
lang_dicts/roman/en.dict [u'\n', u'\r', u' ', u'!', u'"', u'#', u'$', u'%',
u'&', u"'", u'(', u')', u'*', u',', u'-', u'.', u'/', u'0', u'1', u'2', u'3',
u'4', u'5', u'6', u'7', u'8', u'9', u':', u';', u'?', u'@', u'A', u'B', u'C',
u'D', u'E', u'F', u'G', u'H', u'I', u'J', u'K', u'L', u'M', u'N', u'O', u'P',
u'Q', u'R', u'S', u'T', u'U', u'V', u'W', u'X', u'Y', u'Z', u'[', u']', u'_',
u'`', u'a', u'b', u'c', u'd', u'e', u'f', u'g', u'h', u'i', u'j', u'k', u'l',
u'm', u'n', u'o', u'p', u'q', u'r', u's', u't', u'u', u'v', u'w', u'x', u'y',
u'z', u'|'] [u'\n', u'\r', u' ', u'!', u'"', u'#', u'$', u'%', u'&', u"'",
u'(', u')', u'*', u',', u'-', u'.', u'/', u'0', u'1', u'2', u'3', u'4', u'5',
u'6', u'7', u'8', u'9', u':', u';', u'?', u'@', u'A', u'B', u'C', u'D', u'E',
u'F', u'G', u'H', u'I', u'J', u'K', u'L', u'M', u'N', u'O', u'P', u'Q', u'R',
u'S', u'T', u'U', u'V', u'W', u'X', u'Y', u'Z', u'[', u']', u'_', u'`', u'a',
u'b', u'c', u'd', u'e', u'f', u'g', u'h', u'i', u'j', u'k', u'l', u'm', u'n',
u'o', u'p', u'q', u'r', u's', u't', u'u', u'v', u'w', u'x', u'y', u'z', u'|']
real    0m0.758s user    0m0.730s sys     0m0.000s

root@fouzia:/Project/src# ls -lh ../training_documents/german/german2.txt
-rwxr-xr-x  1 root root 44K 2005-12-25
15:15 ../training_documents/german/german2.txt* root@fouzia:/Project/src#
time ./DBI_Trainer.py ../training_documents/german/german2.txt
lang_dicts/roman/de.dict [u'\n', u' ', u'!', u'"', u"'", u'(', u')', u',',
u'-', u'.', u'/', u'0', u'1', u'2', u'3', u'4', u'5', u'6', u'7', u'8', u'9',
u':', u';', u'>', u'?', u'A', u'B', u'C', u'D', u'E', u'F', u'G', u'H', u'I',
u'J', u'K', u'L', u'M', u'N', u'O', u'P', u'R', u'S', u'T', u'U', u'V', u'W',
u'Z', u'a', u'b', u'c', u'd', u'e', u'f', u'g', u'h', u'i', u'j', u'k', u'l',
u'm', u'n', u'o', u'p', u'q', u'r', u's', u't', u'u', u'v', u'w', u'x', u'y',
u'z', u'\xa9', u'\xc4', u'\xdc', u'\xdf', u'\xe4', u'\xf6', u'\xfc']
[u'\n', u' ', u'!', u'"', u"'", u'(', u')', u',', u'-', u'.', u'/', u'0',
u'1', u'2', u'3', u'4', u'5', u'6', u'7', u'8', u'9', u':', u';', u'>', u'?',
u'A', u'B', u'C', u'D', u'E', u'F', u'G', u'H', u'I', u'J', u'K', u'L', u'M',
u'N', u'O', u'P', u'R', u'S', u'T', u'U', u'V', u'W', u'Z', u'a', u'b', u'c',
u'd', u'e', u'f', u'g', u'h', u'i', u'j', u'k', u'l', u'm', u'n', u'o', u'p',
u'q', u'r', u's', u't', u'u', u'v', u'w', u'x', u'y', u'z', u'\xa9', u'\xc4',
u'\xdc', u'\xdf', u'\xe4', u'\xf6', u'\xfc']

real    0m0.117s
user    0m0.090s
sys     0m0.010s
```

As can be seen, training DBI for a new language takes under a second in most cases.

Language detection with DBI is equally fast. Here is a sample run session in which a multi-lingual document contain Arabic, Urdu, English, German language paragraphs (chunks) is processes.

```
root@fouzia:/Project/src# time ./lang_identifier.py inter_doc.txt UDBI
Scripts Found: ARABIC,  LATIN,  FEMININE,

Chunk Data From Starting Byte:0 To Ending Byte:22
   Top 3 Languages Identified By Dictionary Based Method :
      1. Arabic [13]
      2. pr [11]
      3. Urdu [10]
```

```
Chunk Data From Starting Byte:22 To Ending Byte:2185
   Top 3 Languages Identified By Dictionary Based Method :
      1. German [56]
      2. sp [55]
      3. French [53]


Chunk Data From Starting Byte:2185 To Ending Byte:2186
   Top 3 Languages Identified By Dictionary Based Method :
      1. Urdu [0]
      2. sp [0]
      3. pr [0]


Chunk Data From Starting Byte:2186 To Ending Byte:2518
   Top 3 Languages Identified By Dictionary Based Method :
      1. sp [35]
      2. English [33]
      3. German [33]


Chunk Data From Starting Byte:2518 To Ending Byte:3721
   Top 3 Languages Identified By Dictionary Based Method :
      1. Arabic [38]
      2. Urdu [32]
      3. pr [32]


Chunk Data From Starting Byte:3721 To Ending Byte:4123
   Top 3 Languages Identified By Dictionary Based Method :
      1. sp [34]
      2. English [34]
      3. French [33]


Chunk Data From Starting Byte:4123 To Ending Byte:6451
   Top 3 Languages Identified By Dictionary Based Method :
      1. Urdu [45]
      2. pr [36]
      3. Arabic [32]


Chunk Data From Starting Byte:6451 To Ending Byte:9409
   Top 3 Languages Identified By Dictionary Based Method :
      1. sp [66]
      2. English [63]
      3. German [61]



real    0m0.334s
user    0m0.300s
sys     0m0.030s
```

This shows the effectiveness of DBI as quick and quite reliable detection mechanism for

high performance requirements.

6.3.2 CTA without FF modification

This algorithm is the more accurate but more slow too. Training phase for this algorithm requires hours and if n-grams for n>4 are to be used (5-grams, 6-grams etc.) it can span to days and even months and would require huge storage requirements. However it was noticed that results upto 4-grams are most satisfactory. Using n > 4 results in degradation of accuracy.

Training the algorithm is a 3 step process. First documents are fed against a language for training. Once enough documents for each language have been fed to generate accurate n-gram frequencies, the generated lists are sorted and trimmed to top 1000 elements.

Sample of training sessions for a few Arabic and Urdu documents is given below. For larger documents, generating 4-grams can take hours.

```
root@fouzia:/Project/CT_Algo# ./trainer.py ar
/Project/documents/cleaned_documents/arabic/arabic10.txt
Reading File: /Project/documents/cleaned_documents/arabic/arabic10.txt
Read Time : 0.0292570590973
Normalizing:
 time: 0.00265288352966
Getting ngrams:
 time: 0.031350851059
Calculating Weights:
 time: 0.0387530326843
Generating 2grams:
 time: 0.100138187408
generating 3 grams:
 time: 0.205411195755
generating 4 grams:
 time: 0.455098152161
root@fouzia:/Project/CT_Algo# ./trainer.py ar
/Project/documents/cleaned_documents/arabic/arabic10.txt
Reading File: /Project/documents/cleaned_documents/arabic/arabic10.txt
Read Time : 0.0292570590973
Normalizing:
 time: 0.00265288352966
Getting ngrams:
 time: 0.031350851059
Calculating Weights:
 time: 0.0387530326843
Generating 2grams:
 time: 0.100138187408
generating 3 grams:
 time: 0.205411195755
generating 4 grams:
 time: 0.455098152161


root@fouzia:/Project/CT_Algo# ./trainer.py ur
```

```
/Project/documents/cleaned_documents/urdu/urdu6.txt
Reading File: /Project/documents/cleaned_documents/urdu/urdu6.txt
Read Time : 0.0267460346222
Normalizing:
 time: 0.00266599655151
Getting ngrams:
 time: 0.00911808013916
Calculating Weights:
 time: 0.0296800136566
Generating 2grams:
 time: 0.0721571445465
generating 3 grams:
 time: 0.154779195786
generating 4 grams:
 time: 0.324899196625
root@fouzia:/Project/CT_Algo# ./trainer.py ur
/Project/documents/cleaned_documents/urdu/urdu7.txt
Reading File: /Project/documents/cleaned_documents/urdu/urdu7.txt
Read Time : 0.0137910842896
Normalizing:
 time: 0.00247097015381
Getting ngrams:
 time: 0.00782513618469
Calculating Weights:
 time: 0.0287661552429
Generating 2grams:
 time: 0.0689010620117
generating 3 grams:
 time: 0.153650045395
generating 4 grams:
 time: 0.325874090195
```

After all languages have been sufficiently trained, the whole set is sorted, trimmed to top 1000 n-grams (1-grams to 4-grams) for each language and uniqueness value for each n-gram is calculated.

A session of this process on our test documents is given below:

```
root@fouzia:/Project/src# time ./finalize_CTA_training.sh
Finalizing Training for Arabic
Finalizing Training for Chinese
Finalizing Training for German
Finalizing Training for English
Finalizing Training for French
Finalizing Training for Italian
Finalizing Training for Japanese
Finalizing Training for Persian
Finalizing Training for Spanish
Finalizing Training for Urdu
Calculating uniqueness wieghts for Arabic
Calculating uniqueness wieghts for Chinese
Calculating uniqueness wieghts for German
Calculating uniqueness wieghts for English
Calculating uniqueness wieghts for French
```

```
Calculating uniqueness wieghts for Italian
Calculating uniqueness wieghts for Japanese
Calculating uniqueness wieghts for Persian
Calculating uniqueness wieghts for Spanish
Calculating uniqueness wieghts for Urdu

real    160m19.854s
user    159m3.190s
sys     0m47.720s
```

This shows that for approximately 2.7 MB of training documents the process took 2 hours, 40 minutes and 19 seconds. It must be noted that this is still faster than the fisher discriminant version of CTA.

Language Identification using this method is also somewhat slow but can be used in most usage scenarios. Here is a sample run on the same document which was used for DBI.

```
root@fouzia:/Project/src# time ./lang_identifier.py inter_doc.txt CTA
Scripts Found: ARABIC,  LATIN,  FEMININE,

Chunk Data From Starting Byte:0 To Ending Byte:22
   Top 3 Languages Identified By FF_CT_Algo Method :
      1. Arabic [111.51, 121.67, 137.5, 118.23]
      2. pr [101.57, 92.27, 17.85, 0.0]
      3. French [0.0, 0.0, 0.0, 0.0]


Chunk Data From Starting Byte:22 To Ending Byte:2185
   Top 3 Languages Identified By FF_CT_Algo Method :
      1. German [101.06, 101.61, 51.11, 27.2]
      2. pr [0.0, 0.0, 0.0, 0.0]
      3. French [95.99, 92.05, 34.48, 10.17]


Chunk Data From Starting Byte:2185 To Ending Byte:2186
   Top 3 Languages Identified By FF_CT_Algo Method :
      1. pr [0.0, 0.0, 0.0, 0.0]
      2. French [0.0, 0.0, 0.0, 0.0]
      3. English [0.0, 0.0, 0.0, 0.0]


Chunk Data From Starting Byte:2186 To Ending Byte:2518
   Top 3 Languages Identified By FF_CT_Algo Method :
      1. English [103.88, 114.31, 83.72, 52.16]
      2. sp [110.65, 112.98, 75.32, 30.24]
      3. pr [0.0, 0.0, 0.0, 0.0]


Chunk Data From Starting Byte:2518 To Ending Byte:3721
   Top 3 Languages Identified By FF_CT_Algo Method :
      1. Arabic [107.2, 115.8, 56.81, 28.14]
      2. pr [95.44, 95.58, 24.13, 3.54]
```

```
      3. French [0.0, 0.0, 0.0, 0.0]


Chunk Data From Starting Byte:3721 To Ending Byte:4123
   Top 3 Languages Identified By FF_CT_Algo Method :
      1. English [110.43, 111.85, 90.62, 51.64]
      2. German [107.27, 114.39, 48.82, 10.65]
      3. sp [110.76, 113.65, 77.53, 28.24]


Chunk Data From Starting Byte:4123 To Ending Byte:6451
   Top 3 Languages Identified By FF_CT_Algo Method :
      1. Urdu [110.18, 116.23, 62.39, 39.77]
      2. pr [84.96, 91.09, 27.79, 6.21]
      3. French [0.0, 0.0, 0.0, 0.0]


Chunk Data From Starting Byte:6451 To Ending Byte:9409
   Top 3 Languages Identified By FF_CT_Algo Method :
      1. English [91.46, 91.26, 52.92, 23.24]
      2. sp [99.86, 90.59, 47.06, 18.1]
      3. pr [3.05, 0.35, 0.0, 0.0]



real    0m17.303s
user    0m16.420s
sys     0m0.130s
```

As can be seen it is around 52 times slower than DBI. However language detection is more accurate than DBI.

6.3.3 CTA with FF modification

Fisher discriminant function uses another approach proposed and used in the past for finding n-gram uniqueness across languages.

The first step in training is to generate a list containing all unique n-grams found in all training documents available.

```
root@fouzia:/Project/src# ./fisher_discriminant.py
Language: urdu
      File: urdu1.txt
      File: urdu2.txt
      File: urdu3.txt
      File: urdu4.txt
      File: urdu5.txt
      File: urdu6.txt
      File: urdu7.txt
      File: urdu8.txt
      File: urdu9.txt
```

```
      File: urdu10.txt
      File: urdu11.txt
Language: japaneese
      File: japaneese1.txt
            .
            .
            .

Total Ngrams: 47372

 time: 3926.53322601
```

The process took 1 hour and 5 minutes.
Second step is of calculating frequencies of all n-grams in each training document.

```
root@fouzia:/Project/src# ./fisher_discriminant.py
arabic; ./fisher_discriminant.py chineese; ./fisher_discriminant.py
english; ./fisher_discriminant.py french; ./fisher_discriminant.py
german; ./fisher_discriminant.py italian; ./fisher_discriminant.py
japaneese; ./fisher_discriminant.py persian; ./fisher_discriminant.py
spanish; ./fisher_discriminant.py urdu
      File: arabic10.txt
      File: arabic11.txt
      File: arabic12.txt
      File: arabic13.txt
      File: arabic14.txt
      File: arabic15.txt
      File: arabic1.txt
      File: arabic2.txt
      File: arabic3.txt
      File: arabic4.txt
      File: arabic5.txt
      File: arabic6.txt
      File: arabic7.txt
      File: arabic8.txt
      File: arabic9.txt
time: 16.9870109558
     File: chineese2_clean.html
     File: chineese3_clean.html
     File: chineese5_clean.html
     File: chineese6_clean.html
time: 188.355343103
     File: new.txt
     File: 132.txt
time: 242.632512093
     File: french2.html
time: 120.97786808
     File: german2.txt
     File: german4.txt
     File: german5.txt
time: 124.435526133
     File: italian1.txt
     File: italian2.txt
     File: italian3.txt
     File: italian4.txt
```

```
      File: italian5.txt
 time: 75.2099030018
      File: japaneese1.txt
      File: japaneese2.txt
      File: japaneese3.html
      File: japaneese5.html
 time: 106.28076911
      File: persian_clean10.html
      File: persian_clean.txt
      File: persian_clean1.txt
      File: persian_clean2.txt
      File: persian_clean3.txt
      File: persian_clean4.txt
      File: persian_clean5.txt
      File: persian_clean6.txt
      File: persian_clean7.txt
      File: persian_clean8.txt
      File: persian_clean9.txt
 time: 28.6899340153
      File: spanish1.html
      File: spanish2.html
      File: spanish3.html
      File: spanish4.html
      File: spanish5.txt
 time: 91.8341488838
      File: urdu1.txt
      File: urdu2.txt
      File: urdu3.txt
      File: urdu4.txt
      File: urdu5.txt
      File: urdu6.txt
      File: urdu7.txt
      File: urdu8.txt
      File: urdu9.txt
      File: urdu10.txt
      File: urdu11.txt
 time: 14.1865110397
```

This process took 16 minutes.

The next stop of calculating normalized frequencies takes around 10 minutes.

The step of calculating mean frequencies take around 3-4 minutes.

Around 38 minutes are required to sort this list and trim it to top 1000 items.

The training phase concluded by generating language frequencies from the list generated in the previous step. A session doing this is given below:

```
root@fouzia:/Project/src# ./fisher_discriminant.py arabic
Total N-grams: 47372
   Calculating mean frequencies of all ngrams for language: arabic

   Calculating fisher values of all ngrams for language: arabic
```

```
 time: 21.7646238804
root@fouzia:/Project/src# ./fisher_discriminant.py chineese
Total N-grams: 47372

   Calculating mean frequencies of all ngrams for language: chineese

   Calculating fisher values of all ngrams for language: chineese
 time: 10.7011699677
root@fouzia:/Project/src# ./fisher_discriminant.py english
Total N-grams: 47372

   Calculating mean frequencies of all ngrams for language: english

   Calculating fisher values of all ngrams for language: english
 time: 8.76469802856
root@fouzia:/Project/src# ./fisher_discriminant.py german
Total N-grams: 47372

   Calculating mean frequencies of all ngrams for language: german

   Calculating fisher values of all ngrams for language: german
 time: 10.0287070274
root@fouzia:/Project/src# ./fisher_discriminant.py french
Total N-grams: 47372

   Calculating mean frequencies of all ngrams for language: french

   Calculating fisher values of all ngrams for language: french
 time: 7.99114203453
root@fouzia:/Project/src# ./fisher_discriminant.py italian
Total N-grams: 47372

   Calculating mean frequencies of all ngrams for language: italian

   Calculating fisher values of all ngrams for language: italian
 time: 11.8043589592
root@fouzia:/Project/src# ./fisher_discriminant.py japaneese
Total N-grams: 47372

   Calculating mean frequencies of all ngrams for language: japaneese

   Calculating fisher values of all ngrams for language: japaneese
 time: 12.897441864
root@fouzia:/Project/src# ./fisher_discriminant.py persian
Total N-grams: 47372

   Calculating mean frequencies of all ngrams for language: persian

   Calculating fisher values of all ngrams for language: persian
 time: 17.6784720421
root@fouzia:/Project/src# ./fisher_discriminant.py spanish
Total N-grams: 47372

   Calculating mean frequencies of all ngrams for language: spanish

   Calculating fisher values of all ngrams for language: spanish
 time: 11.4607839584
```

```
root@fouzia:/Project/src# ./fisher_discriminant.py urdu
Total N-grams: 47372

   Calculating mean frequencies of all ngrams for language: urdu

   Calculating fisher values of all ngrams for language: urdu
 time: 17.3237810135
```

This process takes around 96 seconds to complete.
Identification times for fisher discriminant are similar to FF modification. Here is a sample run on a 36K document.

```
root@fouzia:/Project/src# ./fisher_discriminant.py identify
/Project/sample.txt
spanish 47.7521386072
english 38.8319851158
french  31.7983005046

 time: 11.6977980137
```

CHAPTER 7

CONCLUSION AND FUTURE ENHANCEMENT

7. Conclusion and Future Enhancements

7.1 Conclusion

Detection of multiple languages within a single script using Unicode meta data is most useful for multi-lingual documents. Language chunks belonging to different scripts were always identified and extracted correctly by this method. However, detecting language chunks belonging to same script is not possible with this technique and requires further work.

Dictionary based identification serves well for script identification. Language identification results give close match values for languages belonging to the same script as of the original documents.

The N-gram based implementations (CTA with FF modifications and Fisher Discriminant Function) were found to be most accurate giving above 90% accuracy. Training phase of CTA with FF was observed to take much less time than Fisher Discriminant. Identification times and results for both variants were observed to be similar.

For very short documents (less that 30 characters) all algorithms were observed to perform badly for language detection. However, accurate script detection of these documents using Unicode meta info or Dictionary Based Identification was still accurate.

7.2 Future Enhancements

With the ever changing face of information in the computer world, more possibilities for improving language detection are just around the corner. There are many standards and save meta-data along with documents. The increasing adoption of such standards (like XML) will allow use of document meta data for language identification.
Automatic generation of training documents without having the need to check them manually is also a field that requires more research.
Detection of multiple languages belonging to the same script within a document is another field that needs improvement.

REFERENCES

[1] Katia Hayati. *Language Identification on the World Wide Web.* UNIVERSITY of CALIFORNIA, SANTA CRUZ. June, 2004.

[2] Andras Kornai and J. Michael Richards. *Linear Discriminant Text Classification in High Dimension.* (http://www.kornai.com/Papers/his01.pdf)

[3] Zhong GU and Daniel Berleant. *Hash Table Sizes for Storing N-Grams for Text Processing.* Electrical and Computer Engineering, 2215 Coover Hall, Iowa State University, Ames, Iowa 50011.

[4] Clive Souter, Gavin Churcher, Judith Hayes, John Hughes & Stephen Johnson. Hermes, Journal of Linguistics no. 13 – 1994
School of Computer Studies, University of Leeds, Leeds LS2 9JT (UK).

[5] Emmanuel Giguet. *Multilingual Sentence Categorization according to Language.* GREYC — CNRS URA 1526 — Universit de Caen, Esplanade de la Paix, 14032 Caen cedex — France. 10th March 1995.

[6] Gregory Grefenstette. *Comparing two Language Identification Schemes.* Xerox research Centre Europe. 3rd International conference on Statistical Analysis of Textual data, Rome. Dec 11-13, 1995.

[7] Muntsa Padr'o and Llu'ýs Padr'o. *Comparing Methods for Language Identification.* TALP Research Center.Universitat Polit`ecnica de Catalunya Jordi Girona Salgado 1-3, 08034 Barcelona, Spain.

[8] Markus Kuhn. *UTF-8 and Unicode FAQ for Unix/Linux.*
(http://www.cl.cam.ac.uk/~mgk25/unicode.html)

[9] *The UTF-8 names Unicode Encoding Form.* (http://tbray.org/tag/utf-8+names.html)

[10] *Multi-lingual text on Linux.* (http://www.jw-stumpel.nl/stestu.html)

[11] *Recommendations for Creating New Orthographies.* (http://www.unicode.org/notes/tn19/)

[12] Peter Constable and Gary Simons. *Ananalysis of ISO639 Preparing the way for advancements in language identification standards. SIL international 2002.*
(http://www.unicode.org/notes/tn8/SILEWP2002-004.pdf)

[13] *Python Unicode data module.*
(http://www.python.org/doc/2.4.2/lib/module-unicodedata.html)

[14] Clive Souter, Gavin Churcher, Judith Hayes, John Hughes & Stephen Johnson *Natural Language Identification using Corpus-Based Models.* Hermes, Journal of Linguistics no. 13 – 1994,School of Computer Studies, University of Leeds, Leeds LS2 9JT (UK)
http://www.comp.leeds.ac.uk/nti-kbs/ai5/research2.html

[15] Peter G. Constable, *Toward a Model for Language Identification*Defining ontology of language-related categories, SIL International 2002, http://www.sil.org/silewp/2002/SILEWP2002-003.pdf

[16] Martin Wechsler, Paraic Sheridan, Peter Scauble.*Multi-language Text indexing for internet Retrieval,* Swiss federal institute of technology (ETH), CH-8092 Zurich, Switzerland

PUBLICATION

6-045
)

06-045 (v1) unicode aided language identification across multiple scripts and hetrogenous data (download)

by malik Sikander Hayat Khiyal, fareheen hanif, and fouzia latif (e-mail authors)

submitted: Sat, 11 Feb 2006

decision: *none yet*

notes:

(all) A E (auto) Sat, 11 Feb 2006 manuscript submitted by Sikandar

(add note)

Unicode Aided Language Identification Across Multiple Scripts and Heterogenous Data

Farheen Hanif, Fouzia Latif
and
M. Sikandar Hayat Khiyal
Department of Computer Science
International Islamic University
Sector H/10, Islamabad, PAKISTAN.
Email: farheen1112@hotmail.com
fouzia_latif@yahoo.com
hdcs@iiu.edu.pk

Abstract

With growing explosion of multi-lingual data on the Internet and other informational and communicational fields, the requirement of having effective automated language identifiers has increased further. More information finds its way into the computer systems and the web and using manual methods to categorize the information is becoming increasingly in-feasible. In this paper we discuss improvements we have achieved in existing language identification methods. Couple of new areas that were not explored before is the inclusion of non-Roman scripts and active usage of Unicode information about scripts to enhance the language detection process.

Keywords: language identification; unicode; multi-lingual documents; n-grams; internationalization; AI; language script.

1. Introduction

The fundamental purpose of "language" identifiers is to indicate distinctions related to linguistic properties and specifically distinctions that are relevant for IT purposes. There are a wide variety of distinctions pertaining to several distinct linguistic parameters that have been suggested as potentially relevant for "language" identification: languages, language families, dialects, country variants, other regional-based variants, script variants, style variants, and modality variants, time based variants, typographic variants, etc. Many different orthogonal parameters could be used in meta-data attributes, and the potential combinations and permutations are daunting. In actual practice many of the potential distinctions are not needed for most realistic usage scenarios.

Application areas can probably be divided into two general types: cataloging and retrieval of content and resources for localization and language enabling of software.

Our research provides Unicode support and multi-script support (e-g Roman, Chinese, Arabic). It also provides language detection for multilingual documents.

2. Identification Techniques

Dictionary based identification methods had been the most used methods in the early stages. In the 80's Cavnar and Trenkle presented their n-gram based algorithm [2] for language detection that solved many problems that persisted with previous language identification techniques.

In our research we have tried to improve on what already has been done. We have implement improved variations of dictionary based and n-gram based algorithms.

2.1 Dictionary Based Identification:

Dictionary based identification (DBI) is one of the most tried and implemented methods for language identification[8]. Though simple, its effectiveness in certain areas cannot be denied. One of its biggest advantages is its performance. DBI is much faster during training and detection than other algorithms used in this research. DBI gives good results when comparing languages belonging to different scripts. If used wisely this technique becomes more useful than it seems at first glance. Joined with Unicode based inter-language document identification, to detect and extract multiple languages data chunks from a document; dictionary based identification is applied on these chunks individually.

Dictionary based method has been previously used but in different ways[14]. Dictionary based method used in our work is more efficient as compared to previously implemented methods. E.g. as compared to [14] our method (DBI) is more efficient due to the following reasons:

- DBI uses N-grams (1 grams) as compared to the compounds used in[14].1-grams covers all the linguistic features.
- DBI requires less memory
- DBI is not a lengthy process as compared to [14].In [14] the compounds are first split into components, normalized and then query structuring for compounds and components. In DBI n-grams are not normalized. They are generated and stored in the dictionary.
- DBi requires no knowledge about the language to be identified.

Like most techniques used in this research, dictionary based identification is also a two phase process. The first phase being training and the second identification. Before going on to identifying documents, the system must first be trained for the languages that we wish to detect. Remember, it only detects the languages it has been trained for. However, if a language belong to a script for which another language has already been trained is found in an input document, that language is identified as the language that was trained for the same script. For example, say in the Arabic script, we have trained DBI for Arabic, Urdu and Persian languages. If we try to detect a Pushto, Punjabi or similar document, all Arabic, Urdu and Persian will come up as close matches giving a hint to the script the language belongs to.

The drawback in dictionary based implementation is that it doesn't perform well when comparing between languages belonging to the same script. Also document containing too few characters give much less information to DBI to detect languages effectively. Also, in case of documents like web pages etc containing multiple languages, the results are not able to clearly distinguish the language. However, if one language occupies the majority portion of the document, it has higher match weight-age hinting its identification.

Algorithm:

Training Phase:

Inputs: 1) Training Document
2) Dictionary File Path

Output: Unique characters list

- Open training document.
- Read all data from the document.

- Generate a list of unique characters present in the training document.
- Sort the unique characters list.
- Dump the list into the dictionary file specified.

Identification Phase:
Input: 1) Document to be identified

Output: 1) Language match count

- Open input document and read all data.
- Generate a list of unique characters in the document.
- Sort the generated list.
- Get a list of script folders in the lang_dicts folder
- for each script_folder in lang_dicts folder:
  - Get a list of language files in the script folder
  - For each language:
    - Load the language list of unique characters
    - Get match count of characters present in document list that are also present in language list.
- Display match count statistics for each language.

The language unique character lists are stored in a file folder hierarchy as displayed below:

- lang_dicts/ (root folder)
  - script folder 1/
    - language1 list file
    - language2 list file
  - script folder 2/
    - language 3 list file

2.2 Unicode Based Inter-Document Language/Script Identification:

The use of Unicode in our research has proved to be very useful. The global acceptance of Unicode and the well thought out placement of different languages in the Unicode code-set has given us more opportunities to detect languages effectively. Each script/language in Unicode has defined code ranges. This information is provided in the form of Unicode Database giving type, language, code point and other info for every character of every language represented in Unicode [10, 14]. The use of this database enabled us to detect different scripts in a document by querying the Unicode database for info about any character.

Unicode based identification is the only technique in our research that does not require an explicit training phase. Only using updated versions of the Unicode database provided at www.unicode.org is sufficient for improving the technique.

This technique is the fastest technique for language detection that we have implemented. As added benefit, this technique also allows us to identify different script portions within a document.

Like all, this technique also has its drawbacks. First, for those scripts that have many languages in them (Roman, Arabic, etc.) this technique can only detect the script and not the language. Secondly for inter-document language identification, detection of different language chunks is effective if consecutive language chunks belong to different scripts. For example, it accurately separates Arabic and English language chunks repeating one after another but has problems if English and French or German language chunks start repeating after one another.

Algorithm:

Input: 1) Input Document

Output: List of languages and their byte ranges in the input document.

- Read all data from input document.
- set current_script = ""
- For each character in data:
  - Get Unicode category of the character (Letter, Digit, Punctuation, etc.)
  - If category = Letter
    - Get Unicode character name
    - Get the script name portion from the character name

- if script_name != current_script:
  - Add last script to language chunks with start and end byte positions
  - set current_script = new script name
- Add last script to the language chunks list
- Remove first empty chunk from the list.

2.3 Cavnar's and Trenkle's Algorithm

C&T algorithm [2] is the most accurate algorithm around for language detection. This algorithm concentrates on aphabet-combination characteristics of languages. Because of this property, this algorithm excels where other algorithms fail. It more accurately identifies languages belonging to the same script.

This algorithm performs its calculations on n-grams. The value of n can be any digit (1,2,3, etc.) The number of n-grams in a document is equal to the number of characters in that document. For example, take the text "HELLO WORLD". 1-grams of this text are: 'H', 'E', 'L', 'L', 'O', ' ', 'W', 'O', 'R', 'L', 'D'. 2-grams for the same text are: 'HE', 'EL', 'LL', 'LO', 'O ', ' W', 'WO', 'OR', 'RL', 'LD', 'DH'. The advantage of using n-grams is that they highlight the language properties. Finding common alphabetic combinations of a language is the purpose of processing n-grams.

Over the years modifications and filtering techniques have been applied to further improve the performance of this technique. The most recent successful addition to the technique was by Katiya Hayati[1] in June, 2004, which included Fisher Discriminant function to give more importance to n-grams that were more unique across languages. We have further implemented another modification to the process by that serves as a replacement of Fisher discriminant function which was too time consuming for n-grams with n greater than 3.

The training phase for this technique spans across multiple steps that are:

- Document filtering to remove formatting information from documents.
- Document normalization for removing multiple white spaces.
- Training multiple documents per language so that sufficient training data per language is present to make the calculations more real.
- Sorting the resulting statistics and calculating top C n-grams according to weight for each language. In this research C=1000 was found to be a good value.

The identification phase involves generating n-grams for the input document and then comparing top C n-grams from the document again top C n-grams for each language to find out match value for that language.

By default the algorithm runs with our custom modifications (called FF modifications here) included. Since comparing the statistics against fisher discriminant implementation was also required to get validated results, so a fisher discriminant implementation of the algorithm is also provided.

Drawbacks of this method involve slower detection and more time required to train documents. Additionally adding a new language to the corpus of languages means doing all the calculations again for each language.

Algorithm (with FF modification):

Training (Step 1):

Inputs: 1) Language Name
2) Training Document

Output: NILL

- If the lang folder exists inside the data folder
  - Load previously trained data for n-grams (n=1-4) from the data files present in the lang folder into n-gram lists.
- If lang folder does not exist:

- Create lang folder
- Initialize n-grams lists to blank
- Read all data from training document
- Normalize the data by replacing multiple white spaces with a single space.
- Generate n-grams from input data.
- Calculate weights (frequency) for each n-gram in input document.
- Update weight of each n-gram in the n-gram lists (loaded from previous training data files).
- Dump n-grams lists to lang folder inside the data folder.

Training (Step 2):

Inputs: 1) Source n-gram list
2) Target n-gram list
3) C (Top no. of n-grams to store)

Output: NILL

- Load Source n-gram list.
- Sort the list in descending order.
- Trim the list to top C items.
- Dump the new list to Target n-gram list path.

Training (Step 3) - n-gram Uniqueness Calculation:

Input: 1) Lang

Output: NILL

- Load n-gram lists from lang folder in the data_finalizer folder.
- Get list of languages in data_finalizer folder.
- For each n-gram in n-gram lists:
  - set u_ng = C * total_langs
  - For each lang:
    - if n-gram found in lang list:
      - idx = index of n-gram in current lang list.
      - u_ng = u_ng -idx
  - Add uniqueness value of n-gram to uniqueness list
- Dump n-gram uniqueness lists to lang folder.

Identification phase:

Input: 1) Input Document

Output: Language match count

- Read the data from document
- Normalize the data.
- Generate n-grams(1-4) for the data.
- Calculate n-gram weights (Frequencies) in data.
- Sort document n-grams list in descending order.
- Trim list to top C items.
- Get list of language folders in the data_finalizer folder.
- For each language:
  - Load lang n-gram lists.
  - Load n-gram uniqueness values.
  - For each document n-gram list (n= 1-4):
    - set diff = 0
    - For each n-gram in list:
      - If n-gram found in lang list:
        - set i1 = index of n-gram in doc list
        - set i2 = index of n-gram in lang list
        - diff += i1- i2
        - uv = unique-ness value of n-gram
        - diff -= (uv/100)
      - if n-gram not found in lang list
        - diff += C+1
        - uv = unique-ness value of n-gram
        - diff -= (uv/100)
- return/display match statistics

Folder structure used in this algorithm is given below:

- data/
  - lang1/
    - 1grams.dat
    - 2grams.dat
    - 3grams.dat
    - 4grams.dat
  - lang2/
    - 1grams.dat
    - 2grams.dat
    - 3grams.dat
    - 4grams.dat

- data_finalizer/

- lang1/
  - 1grams.dat
  - 2grams.dat
  - 3grams.dat
  - 4grams.dat
  - u_1grams.dat
  - u_2grams.dat
  - u_3grams.dat
  - u_4grams.dat
- lang2/
  - 1grams.dat
  - 2grams.dat
  - 3grams.dat
  - 4grams.dat
  - u_1grams.dat
  - u_2grams.dat
  - u_3grams.dat
  - u_4grams.dat

.
.
.

Algorithm (Fisher Discriminant):

Training Phase:

Pre-condition: All training documents are present in training_documents folder.

- Generate a list all unique n-grams found in all training documents of each language. Lets call this list A.
- Get a list of languages in the training_documents folder.
- For each lang:
- Get a list of training docs in that lang.
- For each doc:
  - *For each n-gram in A:*
    - Find frequency of n-gram in doc.
    - Find normalized frequency of n-gram in doc.
- For each n-gram in A:
  - Calculate mean frequency of n-gram
- Sort the list of mean frequencies in descending order of frequency.
- Trim this list to top 1000 items. Lets call this list R.
- *For each lang:*
  - Calculate lang fisher values
    - For each n-gram in A:
      - Calculate lang mean frequency.
      - For each n-gram in R:
        - fR = mean frequency of n-gram in R.
        - fL = mean frequency of n-gram in current lang.
        - n-gram fisher value = *FL / fR*
- Save lang fisher values.

Identification Phase:

Input: Input Document

- Read document data.
- Normalize data.
- Get a list of unique n-grams present in the document.
- For each n-gram in document n-grams:
  - calculate frequency of n-gram in document.
  - Load R (Top thousand n-grams by mean frequency, generated during *training* phase).
  - For each n-gram in R:
    - If n-gram present in document n-grams:
      - calculate n-gram normalized frequency in document
    - else:
      - set normalized frequency of n-gram in doc to 0.00.
- Get a list of Languages available for identification
- *For each lang:*
  - Load lang fisher values.
  - set diff = 0.00
  - For each n-gram in *document* normalized frequencies list:
    - set nFV = lang fisher value for ngram
    - set nNF = document normalized frequency for ngram
    - diff += nNF * nFV
  - Display diff as lang match value for document.

3. Implementation Details

All the *algorithms* in our research were tested on 4 scripts containing a total of 10 language.

The scripts used are:

- Arabic
- Chinese
- Japanese

- Roman

The languages used are:

- Arabic
- Chinese
- English
- French
- German
- Italian
- Japanese
- Persian
- Spanish
- Urdu

Training documents of roughly equal data size were used for training each language to ensure that all languages get equal share of training. All training documents were manually cleaned. Portions of languages other than the desired one were removed from each training document and all the documents were saved in UTF-8 encoding.

Training documents were mostly gathered from the web primarily in HTML format. The other formats used were UTF-8 encoded text, MS. WORD DOC format and PDF.

All the code except for the web interface is coded in Python programming language. The web interface is coded in PHP and HTML running on Apache web server. Operating system used for development and testing is Linux Kernel 2.4 (Slackware 10.2) on a Pentium IV (2.4 GHz) machine with 256 MB of RAM.

4. Features

Main features of the system include:

4.1 Unicode based script/language identification:

The system is not only able to read/write Unicode files, it also uses Unicode extensively for language identification. Use of Unicode makes language identification much more flexible, accurate and faster.

4.2 Unicode based script separation within a single document:

Unicode codepoint information is used to detect the presence of multiple scripts in a document. Data of each script is then processed separately and language identification results for each script chunk are returned.

4.3 Dictionary based (fast) language detection method:

Alphabets in the input document are compared against the alphabets of different languages and based on that the language of the input document is guessed. This method is the most used method for language identification. Although simple, the results returned from this method were found to be satisfactorily accurate and quite fast.

4.4 N-gram based (more accurate) modified method:

This is a modified (improved) version of Cavnar's and Trenkle's algorithm. The modifications give higher weights to n-grams that are less common in other languages.

4.5 Automatic removal of document formatting information:

The system handles input documents like web pages very well and automatically removes all tags and other formatting info from the page using only the page content for language identification.

4.6 Support for easily adding more languages into the system:

More languages can be easily trained using a couple of training programs to expand the system to support more languages.

5. Results

Following are the results, notes and observations we formulated during our research and implementation of the system.

Detection of multiple languages within a single script using Unicode meta data is most useful for multi-lingual documents. Language chunks belonging to different scripts were always identified and extracted correctly by this method. However, detecting language chunks belonging to same script is not possible with this technique and requires further work.

Dictionary based identification serves well for script identification. Language identification results give close match values for languages belonging to the same script as of the original documents.

The N-gram based implementations (CTA with FF modifications and Fisher Discriminant Function) were found to be most accurate giving above 90% accuracy. Training phase of CTA with FF was observed to take much less time than Fisher Discriminant. Identification times and results for both variants were observed to be similar.

For very short documents (less that 30 characters) all algorithms were observed to perform badly for language detection. However, accurate script detection of these documents using Unicode meta info or Dictionary Based Identification was still accurate.

6. Statistics

Total Size of Cleaned Training Documents: 2.7 MB.

Technique	Operation	Time
DBI	Training	0m0.117s
	Identification	0m0.334s
CTA With FF Modification	Training (Generating n-grams[1 to 4])	80m1.002s (1h:20m)
	Training (Finalizing)	160m19.854s (2h:40m:90s)
	Identification	0m17.303s
CTA With Fisher Discriminant (for 3-grams)	Training (Generating n-grams[3 grams])	3926.533226011 (1h:5m)
	Training (Calculating Frequencies)	30m1.119s
	Training (Calculating Fisher Values)	96m9.879s
	Identification	0m18.134s

7. Future Enhancements

With the ever changing face of information in the computer world, more possibilities for improving language detection are just around the corner. There are many standards that save meta-data along with documents. The increasing adoption of such standards (like XML) will allow use of document meta data for language identification. To do so detailed study of different document formats needs to be done along with widespread use of open file formats instead of closed specifications formats.

Automatic generation of training documents without having the need to check them manually is also a field that requires more research. Currently the trainer requires good knowledge of the languages that he/she wishes to train for. Training documents need to contain only the language that they will be used to train. This process involves manual cleaning of documents. Some technique can be developed to use online digital archives like digital libraries to

automatically obtain books or other documents for different languages, clean them and then use them for training the system.

Detection of multiple languages belonging to the same script within a document is another field that needs improvement. Currently this is being done using Unicode script ranges that allows only script detection. This poses problems detecting multiple languages belonging to the same script occurring in tandem. New ways need to be developed based language properties to detect this.

8. References

[1] Katia Hayati. Language Identification on the World Wide Web. UNIVERSITY of CALIFORNIA, SANTA CRUZ. June, 2004.

[2] Andras Kornai and J. Michael Richards. Linear Discriminant Text Classification in High Dimension. (http://www.kornai.com/Papers/his01.pdf)

[3] Zhong GU and Daniel Berleant. Hash Table Sizes for Storing N-Grams for Text Processing. Electrical and Computer Engineering, 2215 Coover Hall, Iowa State University, Ames, Iowa 50011.

[4] Clive Souter, Gavin Churcher, Judith Hayes, John Hughes & Stephen Johnson. Hermes,School of Computer Studies, University of Leeds, Leeds LS2 9JT (UK). Journal of Linguistics no. 13 - 1994

[5] Emmanuel Giguet, Multilingual Sentence Categorization according to Language. GREYC — CNRS URA 1526 — Universit de Caen, Esplanade de la Paix, 14032 Caen cedex — France. 10th March 1995.

[6] Gregory Grefenstette. Comparing two Language Identification Schemes. Xerox research Centre Europe. 3rd International conference on Statistical Analysis of Textual data, Rome. Dec 11-13, 1995.

[7] Muntsa Padr´o and Llu´ýs Padr´o. C omparing Methods for Language Identification. TALP Research Center. Universitat Polit'ecnica de Catalunya Jordi Girona Salgado 1-3, 08034 Barcelona, Spain.

[8] Markus Kuhn. UTF-8 and Unicode FAQ for Unix/Linux. (http://www.cl.cam.ac.uk/~mgk25/unicode.html)

[9] The UTF-8 names Unicode Encoding Form. (http://tbray.org/tag/utf-8+names.html)

[10] Multi-lingual text on Linux. (http://www.jw-stumpel.nl/stestu.html)

[11] Recommendations for Creating New Orthographies. (http://www.unicode.org/notes/tn19/)

[12] Peter Constable and Gary Simons. Ananalysis of ISO639 Preparing the way for advancements in language identification standards. SIL international 2002. (http://www.unicode.org/notes/tn8/SILEWP2002-004.pdf)

[13] Python Unicode data module. (http://www.python.org/doc/2.4.2/lib/module-unicodedata.html)

[14] Turid Hedlund. Compunds in Dictionary Based Cross Language Information Retrieval. Department of Information Studies university of Tampere Finland. Information Research, Vol. 7 No. 2,January 2002. (http://informationr.net/ir/7-2/paper128.html)