Automatic language recognition using phonetics

How phonetic transitions can be used to determine spoken languages

A Bachelors thesis

Monday, May 31st, 2010

Author: A.Z.N. van Pelt
Issued by: Rotterdam University of Applied Sciences
In cooperation with: NCIM Group
Preface

This document describes the theories and implementation of the NCIM language recognition system as was designed and programmed from February 8th to June 7th. During this period the author was an employee at the NCIM group, location Leidschendam. The internship and subsequent thesis were issued by the Rotterdam University of Applied Sciences, also know in Dutch as the Hogeschool Rotterdam. This document is the thesis that was issued by the Rotterdam University of Applied Sciences, as a test of skill and knowledge gained after four years of education at their Computer Science department. The goal of this document is to demonstrate the skill of the student, the author of this document.

I would like to thank the following persons who made this thesis possible:

Lec. H. Manni, (first lector to this thesis)
Ing. J.W. Renema, (company coach)
Lec. Ir. P.J. den Brok, (second lector to this thesis)

I would also like to thank the personnel of the NCIM Group, for supporting and advising on the designing and programming of the product associated with this thesis. Special thanks also go out to the lectors of the computer science department of the Rotterdam University of Applied Sciences for four years of valuable education. Finally I would like to thank all other persons who directly or indirectly helped to make this thesis possible.

Leidschendam, 23rd April 2010

A.Z.N. van Pelt
# Table of contents

1: Summary 4

2: Introduction 5
   2.1 The company 5
   2.2 The Project 5

3: The Assignment 7
   3.1 Description 7
   3.2 Definition of the syllabus 7
   3.3 Quality criteria 7

4: Analysis 8
   4.1 Problem analysis 8
   4.2 Definitions 8
   4.3 Theories 9
   4.4 Chosen theories 9
   4.5 Final plan 12

5: Design 13
   5.1 Task definitions 13
   5.2 Task execution 13
   5.3 Package description 14
   5.4 Design decisions 15
   5.5 Programming and software decisions 16

6: Functional design 17
   6.1 Evolution of the design 17
   6.2 Package responsibilities 19

7: Program flow 25
   7.1 Training 25
   7.2 Importing training 27
   7.3 Recognition of a file 29
   7.4 Audio processing 31

8: Problems and solutions 33
   8.1 Short file recognition rates 33
   8.2 Dutch corpus 33
   8.3 Obtaining transition databases 34
   8.4 Phonetic recognition 35
   8.5 Speed 36
   8.6 The database 36
   8.7 Data structure 38

9: Recommendations 39
   9.1 Research recommendations 39
   9.2 Programming recommendations 39

10 Conclusion 41

11: References 42

Appendices 43
   Appendix A 43
   Appendix B 44
   Appendix C 45
   Appendix D 46
1: Summary

The central research question which is answered in this thesis is:

*How can a language be identified using only sound and how can we use this knowledge to classify audio files?*

This research question can be split into two separate research questions, the first being:

*How can a language be identified using only sound?*

The answer to this question is currently impossible, because the definition 'sound' is too general. Humans can create several different kinds of 'sound', therefore to solve this problem 'sound' had to be redefined to phonemes. Phonemes are common in linguistics and have their own standards. By using the standards of phonetics, sound can be classified and processed linguistically.

This however does not solve the question of how a language can be identified, it just changes the question to:

*How can a language be identified using phonemes?*

Phonemes themselves do not tell anything about what language is spoken by a person. Phonemes are just a classification of sound a human can produce using its mouth. Therefore to solve this problem one needs to focus on the phenomenon language. When focusing on how phonemes are used in a language one can conclude that each language uses phonemes differently. The frequency, order in which phonemes are arranged, and the absence or presence of phonemes can be used to classify a language. Rating the presence or absence of phonemes is a very dangerous thing to do. An error in recognizing a random phoneme can exclude the correct language and cause incorrect language recognition. Therefore the frequency of phonemes and the way they are ordered are reliable enough to be used to identify a language.

The solution of this first research question is therefore:

*Yes, a language can be identified using only the sound which a speaker uses. This requires the usage of phonetic frequency and phonetic composition of the sound a speaker produces.*

The second question is:

*How can we use this knowledge to classify audio files?*

The first part of the answer is quite obvious. Phonemes need to be recognized in an audio file. This however is not recognizing a language. It is only the first step in the recognition process. Before a language can be identified the system needs to know how a language is composed in a phonetic level. This is done by looking at the frequency of phones and by measuring their transitions. When the frequency and composition has been recorded a statistical representation of a language can be made to which a sample can be scored.

The solution of this problem is therefore:

*By creating a statistical model of a language, we can use frequency and composition (to a certain extend) and compare it to an audio file. This information can then be used to classify it.*
2: Introduction

2.1: The company

The NCIM Group specializes in seconding highly educated professionals to manage projects, consult, develop software, and manage systems in technical environments.

More than 200 skilled professionals work on IT solutions in the defense and security, energy and utilities, telecommunications, transportation and media, and technical automation sectors.

Passion for technology – that is what every NCIM employee has. We are all software developers or system engineers, or have started as such, and are proud of it. We work on technical and innovative assignments for renowned companies.

Our employees are constantly evolving. We offer them education programs and training courses that enable them to broaden their knowledge, and ensure they are assigned suitable and challenging jobs. They are also coached by experienced colleagues.

2.2 The project

The core business of the NCIM Group is to assist in projects which are executed mostly by government branches and confidential branches. Because the work by the professionals of the NCIM Group is mostly confidential, it is not possible to display the talents of these professionals to the outside world. To display the skill of the professionals at NCIM to potential customers, the NCIM Group host several projects. These public projects are aimed at displaying the skill of the professionals, because these are not confidential, the NCIM Group can use them as demonstration software.

These projects are being executed by the personnel who are in between jobs, not seconding on a project. The problem with this system is that it is very difficult to actually produce a viable product. It is very hard to estimate if there is enough personnel available to finish these projects, so the company hires graduation students to help with these projects. The advantage of this system is that the project never actually shuts down due to a lack of personnel.

One of these projects is the Speech and Sound recognition project. The Speech and Sound recognition project is a project that is completely done by grad students and is divided in 5 sub projects:

- Signal Improvement
- Sex recognition
- Unique voice recognition
- Word recognition
- Language recognition

The sub project the author of this document was engaged in, was the Language recognition Automatic language recognition using phonetics
This sub project is aimed at recognizing what language is spoken in an audio file. This is a very interesting project if you look at it from a cultural and a security standpoint. It is a culturally interesting project because most countries are slowly changing from a single culture society to a multi-cultural society. The problem with a multicultural society versus a single culture society is the fact that the people in multicultural societies tend to have a more diverse spectrum of languages they can use.

This brings us to the problem of translating. Translating can be easily done by a translator, but for a translator to be effective, the translator needs to know what language is spoken. The recognition of languages by a human being is difficult because there are a lot of languages. A human usually only knows a handful, so languages that sound alike cannot be identified by most people. An application that can recognize a language could be very useful in determining the language that is spoken so a correct translator can be called to assist.
3: The assignment

3.1 Description

In the Netherlands alone, there were approximately 25 million telephone connections in the year 2007. There are 16 million Dutchmen, so that means that every Dutchmen has at least 1,5 telephones available. If we assume half of these telephones are used on a daily basis then we can say that in the Netherlands alone 12,5 million phone calls are being conducted. This in turn means that there is a lot of information is transferred by the means of a telephone. For national security agencies this is a huge problem. To find a (potential) dangerous call amongst a lot of other calls, security agencies need a set of tools to automatically analyze this big heap of data. The Speech and Sound recognition is aimed to show these agencies that the professionals at NCIM have the necessary knowledge to help in these projects.

The sub project language recognition is aimed at analyzing audio files so the language that is used by subjects in the audio file can be identified. The central question in this subproject is:

How can a language be identified using only sound and how can we use this knowledge to classify audio files?

3.2 Definitions

The assignment delimitations have been set at:

- The final product can recognize English and Dutch spoken text files.
- The final product can determine the language in any file that is compliant with the .wav file format.
- The system must have a feature that allows for more languages to be added.
- For the best result the wav format will be 16bit/16khz compliant

3.3 Quality criteria

The program should be able to recognize a language in an audio file, which is a recorded telephone conversation. Therefore the program should be able to perform its functions when audio quality is poor. Also for the program to be useful, it should achieve a recognition error rate of 20% maximum, over audio files longer than 30 seconds.
4: Analysis

4.1 Definitions

In this chapter the following concepts are used:

Sound

In this thesis sound is described as vibrations, causing vibrations which are directional molecular collisions. The word air is not used because sound can also be transferred through metal or water, in other words, environments where there are sufficient molecules to transfer vibrations. On the other hand the definition excludes the notion that there is sound outside of any atmosphere, which is correct, because sound does not exist in space.

Phonemes

Phonemes are the most basic sounds a human can produce using parts of its mouth, lungs, and throat. By combining these phonemes one can pronounce several different words. Phonemes can be identified, arranged and classified in several different ways.

Phonetic recognizer

A phonetic recognizer is a device that receives input in the form of audio and matches this audio to all the phonemes it can recognize. The output is a stream of recognized phonemes which corresponds to the sound that was inputted. Technically, language recognition is limited to the accuracy of a phonetic recognizer.

All other concepts are explained during the remainder of the thesis.

4.2 Problem analysis

In the previous chapter, a practical case was given to show how this project can be used in a real life situation, so this chapter will restrict itself to only analyze the technical problems of language recognition.

A speech recognizer is a different kind of program. Speech recognition programs use statistics in combination with a phonetic recognizer. Speech recognizers actually “guess” the next phoneme in an audio stream. This way a misheard phoneme can be replaced by a phoneme which statistically fits the words of a language better. The replacing of phonemes is why a speech recognizer is accurate, without it, it would achieve very bad recognition rates.

Language recognition software cannot boost its accuracy replacing phonemes, because what is a better statistical substitute in one language can be a poor substitute in another language. Substitution of
recognized phonemes is not a viable option. Therefore language recognition software tends to have very poor accuracy.

4.3 Theories

Automatic language recognition is a field that supports two main theories, phoneme based language identification and acoustic language identification.

Phoneme based language identification tries to recognize a language by using the frequency and absence of phonemes. The theory is that a language has a fixed vocabulary of phonemes that may be used. For instance the Dutch have a distinct difference between their “G” (used in the Dutch word “Groen”) phoneme and their “H” (used is the Dutch word “Hallo”) phoneme. In contrast, a native Spanish speaker cannot distinguish both phonemes, because the Spanish language only contains a “G” phoneme. By identifying a vocabulary of a language, it is possible to exclude a certain language if a phoneme is recognized that is not native to a language.

Acoustic language identification theory is based on the assumption that a phoneme can be stressed or non-stressed. This stressing of phonemes has several rules that differ by language. By identifying these stressed phonemes, and their locations it can be possible to identify a language. Also small differences in the pronunciation of phonemes can be used to discriminate between languages.

4.4 Chosen theory

The theory that was chosen for this project was the phoneme based language identification theory. This theory was chosen because it appears to be a method that is easy to train and because it uses more variables to recognize a language. Therefore phoneme based language recognition should be a more accurate method of recognizing a language.

There are two main pillars of the phoneme based language recognition theory:

− the absence or appearance of phonemes
− the frequency of phonemes

The absence or appearance of phonemes is used to exclude possible languages beforehand. For instance there is a language family known in central Africa know as the “Tuu” family languages. A family language is a collection of languages that share the same origins. The Tuu family languages use so called “click” phonemes. These phonemes are specific for these languages, so if one of these phonemes is recognized one can exclude languages outside of this language family beforehand.

The frequency of phonemes is a statistical way of measuring a language. The first step in using this method to identify a language is to analyze how a language uses phonetics. From this analysis a statistical model can be produced representing a language. Using this statistical representation of a language a sample can be broken down into phonemes and be compared to a language statistically.
Within this project two supplementary theories are used:

− Phonologic barbarism theory
− Phonetic transition theory

The phonologic barbarism theory supports the fact that languages influence each other, even on a phonetic level. Barbarisms can be described as the “copying” of words from one language to the other. A very good example is the English word “apartheid” which has its roots in the Dutch language. By allowing foreign words and their pronunciation to be added to a language, a language actually becomes less “pure”.

The process of barbarism is not only limited to the adding and changing of words, it is also influencing the phonetic vocabulary of languages. For instance the Dutch language contains a “R” phoneme that is changing, due to barbarisms, from an alveolar trill to a voiced uvular fricative (so called French “R”) or a retroflex approximant (so called English “R”). For readers which are not acquainted to these definitions appendix C can be referenced for a better explanation. To account for these changes in pronunciation and adding of phonemes, the program cannot exclude languages beforehand. This is not necessarily a contradiction to "the absence or appearance of phonemes" that is used as one of the main principles of phonetic language identification. This principle is just changed from "excluding a language" to "making it less likely that is this language".

The second theory that is added to this project is the phonetic transition theory. This theory is founded upon the assumption that languages have non-written rules about the order in which phonemes may be arranged. If these rules are not followed a word will sounds "alien" to native speakers. This is mostly the case with words that are derived from barbarisms. Also, words derived from a barbarism will almost always be slightly adjusted to sound "natural". A good example is the English word “computer”, this word is "copied" into the Dutch language, but its pronunciation is adjusted to sound more like other Dutch words.

To determine if a word "sounds like a language" statistically, the sequence of phonemes is important. The first step of finding the sequence of phonemes is to find and measure phonetic transitions in a language. These transitions can then be scored and based on these scores a language can be chosen as the best statistical candidate. The phonetic transition theory is vital to the program because it increases the amount of statistical parameters to which a language can be compared. This can best be described with an example.

Assume we have a phonetic recognition engine. This phonetic recognition engine has only the ability to recognize 2 different phonemes, a 'E' phoneme and an 'N' phoneme. All the sound which is heard by this phonetic recognizer is recognized as either an 'E' or an 'N'. This means that all other phonemes will not be recognized for what they are, but will be recognized as either an 'E' or a 'N'. The notation of the phoneme 'E' will be #A# and notation of the 'N' phoneme will be #B#. Using this information we can create a representation of any language. By simply recognizing phonemes in a sound file which is associated with one language we can get an idea of how the phonetic engine would recognize a language. Let's say we have a random sentence from the Dutch language we want to train.
The training sentence is “Jantje houdt van vissen”. Our phonetic recognizer decodes this sound file to:

<table>
<thead>
<tr>
<th>Jantje</th>
<th>houdt</th>
<th>van</th>
<th>vissen</th>
</tr>
</thead>
<tbody>
<tr>
<td>#A# #A# #B# #A#</td>
<td>#A# #B#</td>
<td>#A#</td>
<td>#A# #A# #B#</td>
</tr>
</tbody>
</table>

If we measure the frequency of the recognized phonemes we can create the following table:

- #A# occurs 8 times
- #B# occurs 4 times

In other words, the #A# phone was registered eight times while the #B# phone was recognized four times. We can use this information to recognize the Dutch language. If we recognize a file which has twice as much #A# phones as #B# phones we can state that Dutch is spoken. This is not a very accurate representation of the Dutch language, and another language can quite easily be recognized as Dutch. For instance the following random sentence from the English language would also qualify as Dutch: “James went fishing”. This sentence is decoded by the phonetic recognizer to:

<table>
<thead>
<tr>
<th>James</th>
<th>went</th>
<th>fishing</th>
</tr>
</thead>
<tbody>
<tr>
<td>#A# #A# #A# #A#</td>
<td>#B# #B# #A#</td>
<td>#A# #A# #A# #B# #B#</td>
</tr>
</tbody>
</table>

This sentence qualifies as Dutch because it has twice as much phoneme #A# as phoneme #B#. So in order to differ between Dutch and English more parameters are needed. To increase the amount of parameters we cannot use the frequency of phonemes, but we have to turn to their transitions.

If we measure both languages phonetic transitions we can say that the representation of Dutch becomes:

- #A# to #A# occurs 4 times
- #A# to #B# occurs 4 times
- #B# to #A# occurs 3 times
- #B# to #B# occurs 0 times

While the sentence from the English language would look like:

- #A# to #A# occurs 6 times
- #A# to #B# occurs 2 times
- #B# to #A# occurs 1 time
- #B# to #B# occurs 2 times

The difference between these two languages is now measurable. In conclusion we can say, by increasing the size of the samples from one phoneme, to two phonemes we can distinguish both language.
4.5 Final Plan

To implement all the theories mentioned in the previous chapter, there are two distinct parts which are required:

-A device or subprogram that is able to recognize phonemes.
-A system that is able to process the data from the first program and rate sound files.

The first part of the program is the most critical part of the system. If this part of the system is not able to read and decode sound files with a reasonable accuracy the system cannot conclude which language is being spoken. To recognize phonemes a speech engine was converted to a phonetic recognition engine. This was done by creating an artificial language that consists out of words that are only one phoneme in length. The speech engine was then configured to accept this language and thus return all the phonemes which were recognized.

The choice to convert a speech engine to a phonetic recognition engine was based upon the fact that phonetic recognition engines are very hard to come by. On the other hand, speech recognition engines are more common and they have a phonetic recognizer embedded. The retrieving of phonemes, before they are used to recognize words appeared an easy solution, but in the end, the creation of the phonetic language appears much easier and faster. This is mostly due to the fact that a speech recognition engine is a very complex piece of software. If parts of it were to be excluded the software would need to be partially rewritten.

The second part of the system is a combination of systems that cooperate to analyze and store data that comes out of the speech engine. The storing and analyzing part of the system needs to be trained for each language separately. This is necessary because it also creates statistical models of all known languages.
5: Design

Readers of this chapter are advised to keep appendix A nearby so it can be referenced quickly.

5.1 Task definitions

After reviewing the theories we can state that the system has four tasks, the phonetic engine task, the database task, the analysis task and the training task.

The phonetic engine task is a part of the system that is responsible for processing sound files and decodes them to such an extent that the engine returns a series of phonemes.

The database task is responsible for saving data to the hard drive and for the creation of the statistical language model.

The third task, the analysis task is responsible for scoring a sound file and determines which language is being spoken.

The last task, the training task is to create a statistical representation of a language.

5.2 task execution

The analyzing tool receives input from the phonetic recognition engine and scores them against a statistical language model stored in the database. By rating each phonetic transition and adding the scores to each language, one can calculate a score for each known language.

During the design phase research revealed, that there is no high quality phonetic engine available. This problem was solved by modifying the sphinx 4 engine. The engine was adjusted in such a way that it currently only recognizes phonemes. How this engine was modified can be read in chapter 8.4 in which the phonetic recognizer is explained.

To facilitate all the task of the system the final program has been divided into six separate Java packages. Each package has its own clear purpose, but work together to accomplish the three given tasks.

The phonetic engine task has been split into two packages, a phonetic output package and a loader package. The phonetic output package (sphinx output) serves as a control class for the phonetic engine, while the other package (loader) controls the output package and handles its file input and output queues.

The database task has been placed into one package, the CSV package. The CSV type of database was chosen for this project, but this kind of database can be replaced by a different kind of database. The system was designed with this flexibility in mind.
This leaves us with two packages in appendix A that appear not to help the other packages. These packages house all the necessary methods to execute two different programs. The programs are the products the users will see and administrators can use to improve the system. The two programs are the language analyzer, with a GUI, and a trainer program, without a GUI. Each of these programs has their own separate packages.

The complete system would look like illustration 1.

![Illustration 1: Block diagram](image)

### 5.3 Package description

To get a better impression of these packages and their sub-tasks, they will be quickly described.

The first package of the system is the speech recognition engine, called sphinx in the class diagram. This speech recognition engine has been tuned to recognize phonemes instead of words. How the Sphinx engine exactly has been tuned is not important to understand the system architecture. The speech recognition engine has been tuned in terms of speed and accuracy by the SphinxOutput package. This class is used to allocate the speech engine and to receive the phonetic transcription of a sound file.

The second package is the Loader package. The loader package is essentially a big interface for the sphinx engine. It is designed so changing the phonetic engine, currently sphinx, can be achieved very fast. In non technical terms, it is the connection between the sphinx engine and the other parts of
the system. The Loader package also houses an internal queue system so sound files can be queued before they are passed through the Sphinx engine.

The database task of the system has been placed in the CSV package. CSV stands for Comma Separated Value and is an old data storage system. This format was chosen for this project because it is a simple format to use. The CSV package is a package that is used to create and read databases of the Comma Separated Value format. This type of database is used to display the database to users. The system is currently still in use by the trainer to store its output to the hard drive. This is done because it will make sure a trained language can be used by several different instances of the language analyzer. Also, it guarantees the mobility of the system because the CSV format can be used on any system that can open and read ASCII encoded files.

The forth package, and first package containing a program, is the Trainer package. The trainer package consists out of two classes, which can be run as one program. These programs have the ability to create a statistical model of a language. When the programs finish they will store a statistical model of a language in the CSV file format to the hard drive.

The fifth package is the language analyzer package. This package is responsible for comparing a sound file to the language model stored in the language database. The language database is a different database then the one described in the CSV package. It is the object oriented database as described in chapter eight paragraph six.

The last package is the Gui package. The GUI package main function is to display information to the user, and to handle user input. It currently only supports the language recognition package, the training package cannot be addressed from this package. The GUI is conceived in such a way that each panel completes one task of the recognition package.

5.4 Design decisions

The first important design choice is to split the sphinx engine in the SphinxOutput package and a loader package. The loader package does not add any functionality to the system, so it appears to be redundant. The loader package was created intentionally because sphinx 4 needed to be separated from the system as much as possible.

The choice to separate sphinx 4 from the rest of the system was made because the recognition speed of the sphinx engine to recognize phonemes is relatively low. It also was very difficult to perform research into the phonetic recognition speed of phonetic and speech recognition engines. Therefore it is very likely a different engine exists that can decode an audio file much faster, and therefore achieve better recognition speeds.

One note tough, training has been done using the same phonetic engine as is used to recognize files. Therefore replacing the phonetic engine also implies it needs to be re-trained. The system has been made as generic as possible, so a new scheme of phonemes is very much a possibility.
The second design choice is the usage of the CSV file as a database. This type of database is still being used by the trainer package to output results and to add a new language into the language analyzer.

The third design choice that was made was the choice to separate the trainer and the language analyzer. This choice was made because both functions are very different from each other. The first is to consult the database; the second is to add a language to the database or to update a language in the database.

The difference is also that in the language analyzer the spoken language is not known, in contrary to the trainer package. In the training package the language that is being trained is known, so a language can be updated.

The language analyzer is not allowed to change the language statistics, so the languages cannot be contaminated with sentences from other languages. If this would happen this would render the program useless within a very short time frame.

5.5 Programming and software decisions

Ubuntu

The Ubuntu Linux platform was chosen for this project because the sphinx engine was written using the Linux platform. To ensure the maximum compatibility with the sphinx engine a distribution of Linux was advised. Being the only programmer on this project my personal preferences in Linux distributions could be applied. My personal favorite Linux distribution is Ubuntu Linux, therefore Ubuntu Linux was chosen.

Programming language and environment

The programming language that was chosen was the JAVA 1.6 programming language. This language was chosen because Sphinx 4 was written in JAVA. The sphinx four engine was chosen because if was recommended by the Carnegie Mellon University. They recommend Sphinx 4, the Java version of sphinx and not the C++ version, Sphinx 3, because Sphinx 4 is faster and achieves better recognition rates.

DB4O

The DB4O database is an object database. An object database is a database that stores instances of classes, not their variables. This saves time when loading instances of classes which are used several times. For more information readers can reference chapter 8.6 in which databases are discussed.
6: Functional design

6.1 Evolution of the design

The plan to create this application was at first very different. During the programming phase features were added, and some were removed. In some cases the execution of the plan was altered, so the program would work better or faster.

To understand the differences between the plans as was used at the beginning of the project, the reader is advised to look at appendices A and B. Appendix A is the first design, non UML compliant, and appendix B is the design that is used in the final program. The initial design is still visible, but has radically changed. In this chapter an explanation is given why these differences were introduced, and what the impact is on the final product.

The first change to the system is merging the Sphinx Decoder packages, which is present two times on the initial design. The idea was that these two decoders would do mostly the same, except for the Sphinx Decoder in the trainer program. The Sphinx Decoder in the trainer program had one extra feature, it could write its output to a file. Also, the second Sphinx Decoder was fitted with a queue so multiple sound files could be loaded and then in turn be decoded. The other Sphinx Decoder was only able to process one file at a time.

Analysis showed that both decoders were almost similar to each other. So both decoders were joined to become one decoder, with an option to write the output to a buffer. The actual writing to the buffer however was given to a support class called sphinx_queue. The queue was removed from the decoder and given to the sphinx_queue class. The decoder and its configuration file are currently located in the sphinx package.

The sphinx_queue support class, as described in the previous change, was upgraded into its own package. This package is currently called the loader package. The loader package was fitted with an interface, a factory and a support class called Engine Loader. In the sphinx package a class was added called Sphinx Output. This class implemented the interface provided by the loader package, so the loader would know what methods could be used to control the phonetic engine. It was essential to have a separate output class so the phonetic engine could be exchanged easily by creating an output file that implements the correct interface.

Within the loader package is also a interface called the Phoneme Factory. This Phoneme factory is responsible for starting the phonetic engine by calling its constructor. The last class, the engine loader was the only one with a reference within the application to the speech engine. A different engine can be used by changing the imported resources the Phoneme Factory uses to start a phonetic engine.

The trainer program was placed in its own package for a better overview of the packages. The program received its own main method, which was used to issue commands to the CSV package and to handle the output of the loader class. The buffer in the old design is still present, so a new statistical...
language file can be produced should the old one gets lost or damaged. The buffer is a plain text file in which the output of the phonetic engine is stored. This buffer file can be used to learn languages to the system.

All of the CSV related classes in the old design were moved to one package. This was done because the reader and writer used a lot of the same methods. For instance both had a method to open files. To make debugging and programming new shared features easier, both the reader and writer were placed in one package. The writer was renamed to CSV_creator because this is a better description of what the class does. The entire package has been re-factored from being primary array based to becoming hashmap, a Java native data structure, based. This increases read speed by an impressive 150%.

The language analyzer program in the initial design received more features and classes. The first was a class called loader input. This loader input class takes control of the loader package and stores its output. The loader input class also has several functions to search the output of the loader and to return transitions to the language analyzer class.

The language analyzer class is the same as in the initial design (Appendix A). But instead of consulting the CSV files through the CSV reader another class has been made. This class is the language class. The language class is designed to have a data structure that can be consulted. This data structure corresponds to the CSV File it was loaded from or a previously loaded language from the language database.

The language database class was a new feature used to minimize loading times. It is a feature which was not in the initial designs, but it is added later. The language database is an object database used to store language objects created by the language class. It is also possible to replace, update and delete languages using the language database class. The advantages of an object database have been addressed in chapter 8.6 of this document. It effectively makes sure a language does not need to be reloaded from a CSV file.

The last change is the addition of the graphical user interface. The GUI was not mentioned in the first designs (Appendix A), it was only implied by the lowest dashed arrow. The inputs for the system were also not modeled. This has all changed by adding the GUI package which takes care of this both the input and output to the user.
6.2 Package responsibilities

The sphinx package is responsible for the communication with the phonetic engine. This package is also responsible to store and apply the necessary configuration options to the sphinx engine. The configuration and the communication functionalities were placed into one package because during the project it was not obvious if the phonetic engine used was the best phonetic engine for this task. So in order to make sure the phonetic engine was replaceable by another phonetic engine, the sphinx phonetic engine was placed in its own package. Also tuning some settings is a lot easier if all the options can be found in one package.

The sphinx package has the following parts:

- The sphinx output, a class that interfaces with the engine
- The sphinx configuration settings this is not displayed in the UML diagram due to the fact that is part of the sphinx engine.
- A grammar file, to make sure the sphinx engine recognizes phonemes, this is also part of the sphinx engine.
The loader package is a package that connects the system to a phonetic recognition engine. Its responsibility is not only to maintain connection to the phonetic engine but also to feed the phonetic engine with sound files. Placing the connection to the phonetic engine in one package was very straightforward, especially if the engines need to be swappable. On the other hand, the feeding of the phonetic engine with sound files was placed in this package so all the calls to the phonetic engine could be handled in one package. By placing the feeding of the phonetic engine in another, or even separate packages, would mean that the phonetic engine would be inherited by other packages, thus eliminating the relevance of a loader package.

The loader has the following parts:
- An engine interface, any phonetic output class should implement this interface to make the output class compatible with the loader class.
- A phoneme factory, to act as a constructor to the phonetic engine
- The engine loader, a class in which is stated what engine must be started.
- The loader class, a class that regulates input/output to the phonetic engine
The trainer package manages the training stage of the language recognizer. The training phase has been broken down into two steps: The first is to create a buffer and the second step is to create a statistical representation of this buffer.

The first step, the creation of the buffer, is taken so that a different kind of database can be created from this data. If for instance, the database changes from a database that rates transitions between two phonemes, as is the case now, to a database that rated transitions between 3 or 4 phonemes, the system would not need a new training period.

The second stage, to create a CSV database, is applied due to the fact that the system currently only imports new languages from CSV files. Why this was done can be read in chapter 5.2, therefore there will be no further elaboration on why the CSV file format was chosen. One possibly powerful feature, is that the CSV database type can be changed in the future because the CSV database is placed in its own package.

The trainer has two distinct programs, the Train_a_language program, and the Create_database_from_buffer program. When the first program has finished, it will automatically start the second program to create a database. These two programs are separated, so it is relatively easy to rebuild a database, or to upgrade a database to a different version.
The CSV package is a package that is responsible for the comma separated value file input and output. The package also has a tool to create a statistical model from a buffer file. This tool has been integrated into the CSV package. The design choice was made to put the writer and the reader into one package because both the trainer and the language analyzer may use both the CSV reader and CSV creator. Also, the reader and writer share a lot of methods, this is due to the fact that both of these programs are build upon the standard Java I/O modules, also they both have a device that creates a data structure based on standard input. The system also has a class responsible for matching phonemes to a number, and vice versa. This is used within the reading and writing of files, so a loop can be constructed to convert a buffer to a file for instance.

The CSV Package contains:
- CsvArray, an array class which stores information before it is converted to the correct sound format
- CsvCreator, a class that is able to create a CSV file from a CsvArray object.
- CsvReaderHash, a class that can read a CSV File and convert in to a hashmap data structure.
- Phone_Array, a class that keeps track of what phoneme has what index, and vice versa.
- IOUtils, a class containing all the necessary I/O functions which are shared.
The language analyzer package is a package that has the responsibility to recognize what language is being spoken in a sound file. The way it works is described in the next chapter in which the program flow is explained.

The analyzer has a language object instead of hash maps containing only the data because the language needs to be more flexible than just a hash map. It needs to handle scores, store a score for a language and return transition scores.

In the package is also the loader input class, a class that handles input and output from the loader. This class is necessary to find transitions in a string of phonemes. These transitions are rated and the resulting score is an estimate of what language is being spoken.

The language analyzer package has 4 important classes:

- Language, to store transition values
- The language analyzer class, the manager class
- The language database class, to store a language object to the hard drive.
- The loader input class, to manage in and output from the loader package.
The GUI package is a package that is responsible for the displaying and receiving of information to the user. This means it only contains a user interface, and no further features. The choice to place the GUI into a separate package was based on the assumption that the program might get new features in the future. This means that the GUI should be flexible enough to support new elements. New features can be added easily if there is no need to also create a GUI element before the language analyzer works. This is done best if the GUI is separated from the rest of the program. Also, the GUI is divided into panels. Each panel has a distinct task, and new functionalities should be added to new panels. There are currently two panels, which will be described in the next paragraph.

The package also has a main program which starts the recognition task, with the GUI enabled. The language analyzer package used to also have a main method to start the recognition task without a GUI, but this feature has been canceled.

The GUI package has four classes and one executable file. The four classes are:

- The ControlPanel, a class constructing a panel that is responsible for displaying output to the user and to receive the signal that the recognition process should start.
- The FilePanel, a class constructing a panel that has as its main responsibility to manage selected files. It also has a file selector to make file selection easy.
- The package has a Commands class in which all the commands are recorded the GUI can perform out of its package.
- The GuiFrame class is the class which manages the GUI creation and regulates the execution of commands.
7: Program flow

Within this chapter the execution of some tasks within the system are explained. All readers are recommended to keep the appendices close in which the overall design is being explained.

7.1 Training from a buffer file

- UML diagram

This UML diagram describes the program flow starting at the buffer phase.
Task description

Training is used to get reference material to compare languages to the samples. Training is done by decoding an audio stream to a string of phonemes. This string of phonemes can then be analyzed and all the phonetic transitions can be recorded. The recorded transitions can then be stored into a database which can be referenced when the language analyzer is analyzing. The training database can be seen as a statistical phonetic transition database, in which the chance of one phoneme following another is recorded. The phonetic transition database can therefore become statistically more significant if it contains more trained data.

Task execution

The task is executed by a separate program in the trainer package. The task starts by allocating the phonetic recognition engine. After the phonetic engine has started the trainer package feeds the engine with sound files. Sound files enter the loader package and are stored in a queue so each of them can be decoded separately. After entering the phonetic engine sound files are processed individually, so each file will be represented by one string of phonemes.

When the output of a file is known the output is then placed into a buffer. This buffer is saved to the hard drive and so training more complex statistical phonetic databases is possible. The current phonetic database is of complexity $x^N$ in which $x$ represents the amount of phonemes, and the $N$ the length of a transition. At this point the database knows a transition length of 2 phonemes, but it is possible to define a transition the size of 3 or more phonemes. To eliminate the entire training process one can just consult the buffer and adjust the trainer so it creates a transition database with a transition length of 3 phonemes. After the recognition has finished the trainer then proceeds by reading this buffer file and export it to an CSV File.

When all the sound files are analyzed, the program enters the export phase. During the export phase the buffer is read and all the transitions are counted. The counting is done based upon the assumption that each transition has a “from” and a “to” phoneme. If the “from” and “to” phoneme are combined they form a unique key which is used to store a value. This value corresponds to the amount of times a transition has been seen. Also, the total amount of first phonemes is recorded using a separate key. This saves time when a transition needs to be scored. Transactions are scored by dividing them over the total amount of phonemes. So by adding a column that has the total amount of phonemes the program does not need to recalculate this every time a score needs to be calculated.
7.2 importing training file

- **UML Diagram**

This diagram shows how a language gets added to the list of known languages.

- **Task description**

The training file as was created by the process in the previous paragraph (7.1) cannot be immediately used by the program. The CSV file representing a language needs to be converted to something the system can use a lot faster. This is the language database in the DB4O format. The task of importing a training file has deliberately been simplified, so the average user should not encounter issues with it.

- **Task execution**

The task of importing a training file is a simple task, which is done by running the language analyzer. The language analyzer specifies in the code what file needs to be added. This should be added to a database management tool, at a later stage of the project. This tool has not been build yet due to the fact that there was no time to create this tool. For now the LanguageAnalyserClass takes care of database management. Adding a language for instance can be done by sending a location of a CSV file and a name for the language to the LanguageAnalyserClass which takes care of the processing of this call.
If a language with the same name already exists in the database the program will overwrite this entry. This can be prevented, but the philosophy is that it would be very unwise to keep multiple versions of the same language in the language database. Keeping multiple versions of one language will only confuse the operators and make the necessary maintenance only harder. This policy can be changed though, as the policy is enabled or disabled using a single boolean value.

When all the transitions are stored into a hashmap the hashmap can be added to a language object. The language object keeps this information secure, and makes sure no one can write in this information and thus contaminate the database. The language object also has one variable used to keep the score of a language. This variable is not used yet, but will be in the recognition phase of the project. When the language object has been created it is essentially ready to be used to recognize a language in an audio stream. The recognizer uses a list of languages to compare which language is the most likely candidate. To avoid loading all the languages every time the application starts the language objects are stored into the language database.
7.3 Recognition of a file

- UML Diagram

This diagram shows the program flow for the scoring task.

---

Task description

The most important task of the system is without a doubt the recognition task. The goal of this task is to find out which language is being spoken in an audio file. This is a very complex task due to the fact that it requires some, very computational intensive, calculations. The first intensive calculation is file processing, a process in which an audio file gets converted and the phonetic recognizer analyzes the audio data.
the data. The second aspect to the recognition job is the calculation process of the output from the phonetic analyzer. The output from the phonetic analyzer then needs to be analyzed. The third calculation is necessary to determine the language that has been spoken. When all of these calculations are completed the output needs to be send to a user interface of some kind. The task is complete when the user who inputted the file, receives an output in which it can read which language is being spoken.

- Task execution

When a file is added to the analyzer it is passed into the loader package which puts the file in the associated phonetic engine. The phonetic engine then decodes the file to the correct format and recognizes the phonemes in the file. This is the most difficult task and also the slowest. It is the slowest because conversion of a sound file is a very long and tedious process. After the conversion step the recognition of phonemes is also slow, because the system cannot exclude phonemes, and therefore needs to compare each sound in the sound file to a phoneme.

After a sound file has been decoded, the output can be analyzed. Analyzing is done based on a scoring system. Each transition can be detected and scored using each known language. To get a score the language objects are consulted. The language objects contain lists of the number of occurrences of a transition. These can be divided against the total amount of transitions from the first phoneme to another phoneme. This gives a low score therefore the result is multiplied by one hundred so the transition can have a value between one hundred and zero.
7.4 Audio processing

− UML diagram

![UML Diagram]

− Task description

The audio processing task can be separated into two sub tasks. The first task is allocating the phonetic engine, the second task is recognizing of phonemes. The phonetic engine gets allocated when the program starts up. Although the phonetic engine is not required at startup, it is allocated at startup. The engine is allocated during the startup phase because the engine is necessary later during the recognition process. Given the fact that it can take some time to startup the phonetic recognizer it is better to start the engine at the start of the program.
The recognizing of phonemes is done within the phonetic engine and is therefore different within every phonetic engine. This is however not important for the analysis or training parts of the system. They both use the same methods within the loader package to receive the sequence of phonemes. The analysis and training tasks require no more but a stream of phonemes to work with.

- Task execution
  The phonetic engine allocation can only be requested by the loader package. This is done deliberately so the engine can be controlled and secured by the loader package. For instance the trainer package cannot see how the phonetic engine works, which would slow down the phonetic recognizer. The loader package acts as broker to the phonetic recognition engine. This is to prevent multiple access to the phonetic engine which would slow it down. The loader package guarantees that there is no interference in the input and output of the recognition engine.

  When the loader package has received the call to allocate the phonetic engine, it passes the call to the engine loader class. The loader class has a reference to the phonetic engine and can allocate the engine. When the phonetic engine is started, the engine loader returns an instance of the Sphinx engine to the loader class.

  The recognition of phonemes is dependent on the allocating of the phonetic engine. This means that if the phonetic engine has not been allocated, the system becomes unable to recognize phonemes. If the phonetic engine has been allocated, the Sphinx output class can be asked to decode a sound file. The sequence of phonemes which represents the audio file can then be returned to the loader class which can send it to other parts of the system.
8: Problems and Solutions

In this chapter there will be seven problems listed which were solved during the development phase.

The seven problems which are addressed and their paragraphs are:

8.1) The recognition rates which are very poorly on short sound files.
8.2) The poor way the Dutch corpus appears to perform.
8.3) How the transition databases were created.
8.4) How to create a phonetic engine from a speech engine.
8.5) Why the speed at which the system operates, appears to be slow.
8.6) Why a second type of database was used.
8.7) Why a hash map data structure was chosen.

**8.1 Short file recognition rates**

As mentioned in the previous chapters the program requires a database filled with transition probabilities describing the chance one phoneme follows the other phoneme. This is a very effective method, but it has some problems coping with short sound files. This is due to the fact that short sound files have little speech. This translates into a small amount of phonemes. If there are a few phonemes, only a few transitions can be read. This means that the score of each language will be low, and thus language scores will be very close together. In some cases a different language, than the language spoken, can be chosen as the most likely language. One can say that the longer the sound file, the better the recognition accuracy is.

**SOLUTION:**

There is no satisfactory solution to this problem, except letting users of the system know that the size of their sound file influences the accuracy.

**8.2 Dutch Corpus**

Due to the fact that there was a smaller amount of Dutch speech available, the Dutch corpus is a smaller corpus than the English corpus. Because of the smaller size, the Dutch corpus tends to return greater values then the English corpus. This is because the number of transitions between phoneme “A” to phoneme “B” are divided over all the occurrences of phoneme “A”. So the smaller the amount of phonemes, the greater values there are returned. So in conclusion: the smaller the total amount of transitions the less significant a database is.

The Dutch corpus has 9070 total transitions, if we compare this to the English corpus, 33696 transitions, we can state that the Dutch corpus only is about 27% the size of the English corpus. We can conclude from this data that the Dutch corpus is less significant, and therefore less accurate then the English corpus.
SOLUTION:

The Dutch corpus can be enlarged by analyzing radio broadcasts for instance, or to train the system using a microphone. On the other hand, the solution mentioned in the previous solution also applies to this problem. To increase the accuracy of the database longer sound files can be used. The recommendations are to use sound files of at least 20 seconds.

8.3 Obtaining transition databases

Within the field of Language recognition and even beyond this field, there has been no one that has analyzed a language in this way, as far as the author is aware. Scientific papers do hint in the direction of a transition database, although it is never actually said. Therefore a transition database had to be constructed. The importance of this database is very significant, for most of the theory relies on this database. To create this database a lot of sound data is required, and this data needs to be sorted into different languages, so they can be taught to the system. The problem is not to create a system capable of processing this data, the real problem is the obtaining of enough reliable sound files.

SOLUTION:

To gather the necessary data there are three ways which can yield the necessary data. The first method is the most obvious way, which is to create a sound corpus using a microphone and some test subjects. Creating a sound corpus is a process in which test subjects are asked to perform various verbal tasks. The advantage of this method is that the conditions of the recording can be set to whatever is necessary. Also, test subjects can be asked to perform in an array of tasks to get more varied sound files to fill the corpus with. Research into different kinds of speech can also see if there is a difference in clarity between male and female voices. The obvious disadvantage is the fact that it is relatively hard to obtain this data. One would need to hire test subjects and one would need to spend several hours interacting with them.

The second way is to collect “pod casts” on the Internet. These pod casts can then be cut up into manageable pieces and used for training purposes. This should be a relatively easy way of obtaining the required data. Pod casts are free to download and are a good example of some “spontaneous” text and “tasked” text. The tasked text is mostly prepared text, but not completely written down, so a portion of the text is improvised.

The disadvantage of these pod casts are that they need to be converted to the correct format. This can take some time and will most likely result in the loss of quality.

The third way is to consult on line speech corpora. Corpora are not on line corporations, as the name would suggest, but are collections of sound files typically used to train speech recognition software. These corpora have a lot of different sound files. These sound files are very useful in training the language recognizer and create a statistical language model. The sound files in most corpora are already tuned so they can be used by speech recognition engines, and sorted by extension. This particular method was used to train the language model. The corpus used was the Voxforge corpus. This corpus had Dutch and English sound files in the correct format, so training was relatively simple.
8.4 Phonetic recognition

Automatic phonetic recognition is a relatively slow area of development. Phonetic recognition is a computational intensive operation. This is mostly because of the input processing and the matching of sound to a phoneme. This in turn results in a very low amount of available phonetic recognizers.

Speech recognition uses phonemes and probability calculations to return the most probable words in an audio stream. Basically a speech recognition engine uses a phonetic recognition engine to guess a word. A speech recognition engine needs to guess words, because phonetic recognition engines have very poor performances. So to increase accuracy some phonemes can be replaced by other statistically more significant phonemes.

For this project replacing phonemes is not an option. What is a statistically good replacement for a phoneme in one language can be a very poor substitution in a different language. As long as the phonetic recognizer is consequent in recognizing phonemes, even if they are wrong, the statistical model will adapt automatically. Therefore it is not a great problem that needs to be addressed.

What is a problem on the other hand, is how to find a phonetic engine that does not replace phonemes in a found audio stream so it suits speech recognition better.

SOLUTION:

There are 2 viable ways to acquire a suitable phonetic engine. To start out, a phonetic engine cannot be acquired by looking for one. The fact is that a suitable phonetic engine cannot be found for this project, only embedded ones from speech recognition engines. The first viable way is to “tap” the phonetic recognition engine which is used in the speech recognition engine. The second is to create a phonetic language for the speech recognition engine.

The first method, to tap the phonetic recognition engine, is the hardest way to get a line of phonemes. This is due to the fact that speech- and phonetic engines are merged into one product. To separate these two products is actually a very difficult task. Even if a complete analysis can be made of a speech engine this will remain a really difficult task. It also implies the partial rewrite of the speech engine in total. This is of course beyond the scope of time that is available for this assignment.

The second way is to train the speech engine with a phonetic language. This is also the preferred way to get phonemes is the older version of sphinx 4, the c++ based sphinx 3. A phonetic language is basically a language that has as many words as there are phonemes. So by declaring each word to be one phoneme long, each separate phoneme is seen as a word and therefore treated as one. To accomplish this task one needs to create a dictionary, matching grammar and a language model for this language.

The dictionary should map each phoneme to one word (preferably the same phoneme). A grammar for this language is as easy as: each phoneme can follow each phoneme. The language model for this language would make sure all the transitions are as likely to occur as all other transitions.
will prevent the system from changing phonemes to manipulate the result to one specific language. If all of these tasks are completed a speech recognition engine will effectively recognize phonemes. This method also means that a speech recognition engine would need to be rewritten.

For this project the second way was chosen. This choice was made due to the limited amount of time that was available to complete this project.

8.5 Speed

Speech recognition engines are very extensive pieces of software. They have a wide array of options and internal settings that can be tuned for various tasks. The speech recognition engine used in this project, Sphinx 4, is no different to other speech engines. So to increase speed, an optimal configuration needs to be found and applied to this application.

The optimal configuration for this setup cannot be derived from other speech recognition applications using the sphinx engine. This deriving cannot be done because the sphinx engine is configured in a very specific way that is not standard in any way.

SOLUTION

By the means of practical research one can determine the optimal configuration for the sphinx engine. While testing the system the parts could be identified that were slowing down the system. These parts were then tuned and optimized for this recognition task.

Even though the system was significantly boosted in terms of recognition speed, the system remains slow. The current speed is 20 times real time. This means that the recognizer requires 20 times as much time to decode the audio stream to a phoneme line, as it takes speakers to speak an audio stream. This is an improvement over the 800 times real time that was the standard in the first prototypes.

8.6 The database

The comma separated file format is a format which cannot be read very fast and is not optimal for most applications. In the previous paragraph the relative low speed of the phonetic engine was addressed. Because the phonetic engine is relatively slow the other parts of the system should be as fast as possible to compensate for the loss in speed.

The system already has a comma separated value database, but this is to slow to use when recognizing, and when starting the program, so the problem is how to increase the speed of the comma separated value files so they can be used in the system.

SOLUTION

The database that is used in the program itself is not the CSV database type. To read a single value from this file means that the correct line and column should be found for each read action. This...
requires reading the entire file from the disk into the memory by storing them into some kind of list.

The reading of the file is very slow due to the conversion of a string to a data structure. The reading speed of a file dependent on the amount of phonemes that are used in the program. A file containing more phonemes is bigger and will take more time to load. During the first phase of the project no more than 2 phonemes represented one transition. There is however the possibility that in the future the program can be trained to see transitions to be 3 phonemes long or longer.

In the O-notation we can say that the complexity of loading the comma separated file is:

\[ O(((x^n) * (((1+x)/2) + (O(1))) + y) . \]

The X represents the number of phonemes and the N represents the size of the transitions the Y value is the value needed to open and close the file. The \((x^n)\) represents the amount of values in a table. To store a value in a data structure is considered to be the same no matter the data that is inserted. The \(((1+x)/2)\) represents the average time to look up and read a line of values. To extract a value from a string requires a constant time. The \((O(1))\) is the time it takes to read a value from a string of values. It is not important where the value is in the string, so therefore it is a O(1) operation. The Y value is used to include the time it takes for the operating system to open and close the file.

The complexity of this problem can be reduced using a classical SQL database to one of:

\[ O(((x^n) * O(z)) + y) \]

The X represents the number of phonemes and the N represents the size of the transitions, the Y value is the value needed to open and close the database, Z is the size of the resulting database. The \((x^n)\) represents the amount of values in a table. To store a value in a data structure is considered to be the same no matter the data that is inserted. The \((O(z))\) is the time it takes to read a value from the database. A SQL type of database will be slower if the tables grow bigger. This however is dependent on the implemented database. Some will be quicker than others, but all will be slower if the tables are bigger. Therefore a value Z is used. The Y value is used to include the time it takes for the operating system to open and close the file.

To reduce the complexity of the problem even further an object database can be used. An object database is used to store objects, so an object can be constructed from the data that is available in the database file. This means that instead of filling the data structure one value at a time, it fills the data structure by copying all the data straight into the data structure. The complexity of the system using this database type is:

\[ O(O(z) + y) \]
The Z is the size of the database and y represents the time it takes for the O/S to handle all the data. \( O(z) \) stands for the time it takes to read all the variables and copy them into the database.

In conclusion the DB4O is the fastest solution to load a large table into memory. This only appears to be faster than SQL if the table size remains small, but table sizes will most likely be very large in this project. Therefore The DB4O will grant the fastest speeds.

8.7 Data structure

During the programming phase it was discovered that an array data structure cannot be used when dealing with a lot of information. The system became too slow; to solve this performance issue a different data structure was required.

**SOLUTION**

Before a language can be added to a database tough it needs to be converted to a language object. A language object has a hashmap containing the transitions ask keys and the occurrences as values. A hashmap is a native Java data structure which uses hashing to lookup values in a table. The advantage of hashing over other data structures is that the longer the table to lookup, the faster the algorithm is at average. The speed of a hashmap, in comparison to a sorted list, is slower with a few objects, but from about 400 entries in the table the hashmap becomes faster than a sorted list. This is nearly always the case with this project, only when transitions are not rated, but the occurrence of single phonemes the sorted list performs better.

The contents of the CSV file needs to be converted to a hashmap. This function is available in the CSV package, by using the class CSV reader hash. This is a class that reads the CSV file and creates a hashmap containing the keys and the values which can be found in the original file. The creation of this hashmap is a very intensive process, but when used correctly it only needs to be done once.
9: Recommendations

These are the recommendations for people who wish to continue the project. The recommendations are divided into a research and programming recommendations. Both can be executed separate from each other, but if both are followed the program can be significantly improved.

9.1 Research recommendations

The first recommendation I would like to make on the research is to continue research in the phonetic recognition engines. The current phonetic engine is quite slow, and therefore an engine that can decode an audio stream faster into a string of phonemes can speed up the process. One engine that can be researched is the asterisk telephony engine, the Julius speech recognition engine or any other commercial speech or phonetic engines.

The second recommendation is to change the project to also include the usage of a microphone. This would be a relatively small change in the project, and should be easy to apply to the current phonetic recognition engine. This should speed up the recognition process, as no time would be wasted decoding a waveform audio file (.wav file), into a raw audio file. This can take a noticeable amount of time, especially for large sound files. A microphone in contrast, records raw data and therefore does not require any conversion before it can be used by the Sphinx engine.

The previous recommendation only applies if the Sphinx engine will be used. It might be very well possible that other speech or phonetic engines are better optimized for .wav files, or files in a different format. Sphinx on the other hand was always designed for usage with a microphone. The build in conversion from the waveform audio file format back to raw audio, has been added later to aid more developers. It is therefore not an optimal method for getting input.

A interesting piece of research would be to find the optimal transition length. As was described in previous chapters the transition database is of size $x^n$. In this formula the $x$ represents the amount of phonemes used, and the $n$ value is the transition length. For instance, the current database uses a $40^2$ size, the database has 40 different phonemes, and because it uses transitions which are two phonemes long it can be squared. This gives us a phonetic transition database which contains 1600 entries. This is without counting the totals column added for speed.

There is however the question if the recognizer will improve in accuracy if a transition is bigger than two phonemes as is used now. Also, one size of transition will probably be optimal for each different task. Finding an optimal transition size is a research which can be conducted and will probably improve accuracy.

9.2 Programming recommendations

The first programming recommendation is to take a look at the sphinx engine and work through Automatic language recognition using phonetics page 39 of 46
the tutorials at the project pages which exist for the Sphinx engine. This will create a deeper understanding of the sphinx engine, and will make debugging a lot easier.

It would be a good idea to do some research into the DB4O database. This database should become the standard database of this project, so the CSV package can be demoted so it only serves as a user “window” to the database.

By demoting the CSV package one needs to upgrade the trainer to directly update the DB4O database. This would be a small task, in contrary to the next task, synchronization of two databases. This was formerly done in the CSV package, but the CSV package should be replaced in the future, so a separate program should be made for this task. Synchronizing the databases is a relatively simple task, because one does not need knowledge of the language database to successfully build a synchronizer.

One task that also needs to be done is upgrading the graphical user interface so it can talk to the trainer package. It might even be better to re-factor the entire graphical interface, for both the recognizer and the trainer so they can both be controlled via one GUI. One should do some research into this option, or maybe create a different database merger tool, with a different GUI, which supports merging, replacing and adding better than the GUI used for recognizing and training.

There is also the problem with determining the significance of a language database. This is not necessary for the program to run, but if the program can determine if a language is insignificant it can inform the user. This would grant the user the chance to improve the system training before the system gets a recognition error rate that skyrocket.
10: Conclusion

To summarize, the main research question was:

*How can a language be identified using only sound and how can we use this knowledge to classify audio files?*

The answer to this research question according to this thesis must be:

*Yes, a language can be identified using only the sound which a speaker uses. This requires the usage of phonetic frequency and phonetic composition of the sound a speaker produces. By creating a statistical model of a language, we can use frequency and composition (to a certain extend) and compare it to an audio file. This information can then be used to classify it.*

Other conclusions that were drawn during the research for this project were:

- The Java Hashmap data structure is the fastest Java native data structure for handling large tables.
- Object databases can be faster than conventional databases when loading complete objects.
- Speech recognition is based on phonetic recognition and is therefore a related work field to language recognition. To get a phonetic engine one cannot extract one from a speech recognition engine. A phonetic language can be used to convert a speech recognition engine a phonetic recognition engine.
- By combining statistical analysis and phonetic recognition it is possible to identify a language.
11: References

The following references were consulted when writing this thesis. The references marked with an asterisk (*) are also recommended for readers who wish to learn more about language recognition. All other resources can also be consulted for a broader view.

* B.H. Juang & L.R. Rabiner – *Automatic speech recognition- a brief history of the technology development* - Georgia Institute of Technology (USA), Rutgers University and the University of California (USA) – 2004


* E. Singer, P.A. Torres-Carrasquillo, T.P. Gleason, W.M. Campbell, & D.A. Reynolds - *Acoustic, Phonetic, and Discriminative approaches to automatic language identification* - MIT Lincoln Laboratory (USA ) - 2003

* M. A. Zissman - *Comparison of four approaches to automatic language identification of telephone speech* - IEEE transactions on speech and audio processing, vol 4, no.1, January 1996.

T. Schultz & A. Waibel - *Polyphone decision tree specialization for language adaptation* - Carnegie Mellon University(USA), University of Karlsruhe (Germany)

M. Mohri,  F. Pereira & M. Riley - *Speech recognition with weighted finite state transducers* - Courant Institute(USA ), University of Pennsylvania(USA ), Google Research (USA ) - 2006


Appendices

Appendix A: Original System Design
# Appendix C: International Phonetic Alphabet

## THE INTERNATIONAL PHONETIC ALPHABET (revised to 1993)

<table>
<thead>
<tr>
<th>CONSONANTS (PULMONIC)</th>
<th>Bilabial</th>
<th>Labiodental</th>
<th>Dental</th>
<th>Alveolar</th>
<th>Postalveolar</th>
<th>Retractals</th>
<th>Palatal</th>
<th>Velar</th>
<th>Uvular</th>
<th>Pharyngeal</th>
<th>Cleftal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plosive</td>
<td>p b</td>
<td>t d</td>
<td>t d j k g q g</td>
<td></td>
<td>?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nasal</td>
<td>m m j n</td>
<td>n η j n η</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trill</td>
<td>b</td>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tap or flap</td>
<td>r</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fricative</td>
<td>φ β v θ ð s z ñ s z ç į x ɣ x λ h ʃ h h</td>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lateral fricative</td>
<td>ɬ l s ɭ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Approximant</td>
<td>u a ɻ j w</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lateral approximant</td>
<td>l l ɿ l</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Where symbols appear in pairs, the one to the right represents a voiced consonant, shaded areas denote articulations judged impossible.

### CONSONANTS (NON-PULMONIC)

<table>
<thead>
<tr>
<th>Cucks</th>
<th>Voiced implosives</th>
<th>Ejectives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bilabial</td>
<td>ɓ ɗ Bilabial</td>
<td>as inc</td>
</tr>
<tr>
<td>Dental</td>
<td>p', Dental/alveolar</td>
<td>Bilabial</td>
</tr>
<tr>
<td>Palatoalveolar</td>
<td>s Velar</td>
<td>' Velar</td>
</tr>
<tr>
<td>Alveolar lateral</td>
<td>l Uvular</td>
<td>ɬ Alveolar fricative</td>
</tr>
</tbody>
</table>

### VOWELS

- Front: i y i θ y ɪ u o u
- Central: e o e ø e θ ø ι ø ʊ ø θ ø
- Back: a ø a ø a ø a ø a ø a ø

Where symbols appear in pairs, the one to the right represents a rounded vowel.

### OTHER SYMBOLS

- Voiceless labial-velar fricative: ɬ ɭ Alveo-palatal fricatives
- Voiced labial-velar approximant: ɭ Alveolar lateral flap
- Voiceless labial-palatal approximant: ɭ Alveolar lateral flap
- Voiceless epiglottal fricative: ʃ Syllabic and double articulations can be represented by two symbols joined by a tie bar if necessary
- Epiglottal plosive: kp ʦ

### SUPRASEGMENTALS

- Primary stress: ɬoʊnəˈtʃɪʃən | TONES & WORD ACCENTS
- Secondary stress: | LEVEL
- Long: ɬeː | Extra-high ɬeː | Rising
- Half-long: ɬeː | High ɬeː | Falling
- Extra-short: ɬeː | Mid ɬeː | High rising
- Syllable break: ɬəɪk | Low ɬeː | Low rising
- Minor (foot) group: ɬəɪk | Extra-low ɬeː | Rising-falling etc.
- Major (intonation) group: ɬəɪk | Downstep ɬeː | Global rise
- Linking (absence of a break): ɬəɪk | Upstep ɬeː | Global fall

### DIACRITICS

Diacritics may be placed above a symbol with a descender, e.g. ŋ

- Voiceless: ŋ ɚ Breathy voiced: b a ː Dental: t d
- Voiced: ɭ ɭ Creaky voiced: b a ː Apical: t d
- Aspirated: th ɭ ɭ Lateralized: t d ɭ Lateralized nasal: ɭ
- More rounded: ɭ ɭ Labialized: t ɭ ɭ Nasalized: ɭ
- Less rounded: ɭ ɭ Palatalized: t j ɭ PALATAL RELEASE: d n
- Advanced: ɭ ɭ Volarized: t v d ɭ Lateral release: d ɭ
- Retracted: ɭ ɭ Pharyngealized: t s d ɭ No audible release: d ɭ
- Centralized: ɭ ɭ Volarized or pharyngealized: t t
- Mid-centralized: ɭ ɭ Raised: ɭ ɭ Voiceless labial approximant: ɭ ɭ Voiceless labial-fricative
- Syllabic: ɭ ɭ Lowered: ɭ ɭ Voiced labial-velar approximant
- Non-syllabic: ɭ ɭ Advanced Tongue Root: ɭ ɭ Rhoticity: ɭ ɭ Retracted Tongue Root: ɭ ɭ

Automatic language recognition using phonetics

Page 45 of 46
Appendix D: CD-ROM containing all the information as was presented in this Thesis.

Is your CD-Rom lost?

Please contact the author at the following e-mail address

bram.van.pelt[AT]gmx.net