Master's Thesis

Creating a new Combined Confidence Measure for ASR-Errors on the Word-Level

Philipp Salletmayr, BSc

Signal Processing & Speech Communication Laboratory

Graz University of Technology

in cooperation with

Spoken Language Processing Group

Columbia University in the City of New York

Advisors:

Univ.-Prof. Dipl.-Ing. Dr.techn. Gernot Kubin (TU Graz)

Mag.rer.nat Dr. Barbara Schuppler (TU Graz)

Julia Hirschberg, PhD (Columbia University)

Svetlana Stoyanchev, PhD (Columbia University)

Graz, March 26, 2013

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am 25.03.2013 .... [illegible] (Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

25/03/2013 date .... [illegible] (signature)

ABSTRACT

Most current dialog systems employ very simple strategies when dealing with misrecognitions (i.e., *"please repeat/rephrase"*). This causes problems for the user, as it is not known which exact part of an utterance was misrecognized. This thesis addresses the problem of *localized error detection* in Automatic Speech Recognition (ASR) output and aims at developing methods to identify which particular word(s) in an utterance has been misrecognized. Identifying misrecognized words permits the creation of *targeted* clarification strategies for spoken dialogue systems, allowing the system to ask clarification questions targeting the particular type of misrecognition. This thesis presents results from machine learning experiments using ASR confidence scores, together with prosodic and syntactic features to predict (1) whether an utterance contains an error, and (2) what exact word(s) in a misrecognized utterance is misrecognized. Experiments conducted using different classification techniques on the TRANSTAC database showed that by adding prosodic and syntactic features to the ASR features, prediction of misrecognized utterances improves compared to using ASR features alone. This means that an interactive system with clarification capabilities using the proposed error detection method would attempt to correct over 50% of misrecognized words with a clarification subdialogue. These findings are used to build a classifier for an error detection module in a Spoken Dialog System (SDS).

Keywords: Localized error detection, Automatic Speech Recognition, ASR, Spoken Dialog System (SDS)

Kurzfassung

Die meisten Dialogsysteme verwenden derzeit sehr einfache Strategien um mit Fehlern in der Spracherkennung umzugehen (z.B.: "Bitte wiederholen"). Dies führt oft zu Missverständnissen beim Benutzer, welcher oft nicht weiß, welcher Teil des Gesprochenen nicht verstanden wurde. Die vorliegende Arbeit beschäftigt sich mit diesem Problem der sogenannten lokalen Fehlerdetektion in der automatischen Spracherkennung und hat zum Ziel, Methoden zu entwickeln, mit denen man identifizieren kann, welche Wörter der jeweiligen Benutzereingabe genau zu einem Fehler geführt haben. Diese Identifikation lokaler Fehlerquellen erlaubt es Dialogsystemen, gezielte Fragen abhängig vom Fehlertyp zu stellen. Die vorliegende Arbeit präsentiert Ergebnisse von Experimenten, bei denen verschiedene Methoden des maschinellen Lernens verglichen wurden, um zu klassifizieren (1) ob ein Fehler gemacht wurde bzw. (2) welche Wörter genau den Fehler verursacht haben. Unter Verwendung der Sprachdatenbank TRANSTAC wurde gezeigt, dass durch das Hinzufügen von prosodischer und syntaktischer Information zu den sonst alleinig verwendeten Konfidenzen (die der Spracherkenner selbst ausgibt), die Genauigkeit der Klassifikation signifikant steigt. Die gewonnenen Erkenntnisse wurden bei der Entwicklung eines Dialogsystems ins Fehlerdetektionsmodul integriert. Ein derartiges System, in das die hier präsentierten Methoden integriert sind, versucht bei mehr als der Hälfte der missverstandenen Worte diese mittels einer gezielten Nachfrage zu korrigieren.

Schlagwörter: Lokale Fehlerdetektion, Automatische Spracherkennung, Dialogsysteme

Acknowledgments

I am foremost dearly indebted to both Prof. Julia Hirschberg and Prof. Gernot Kubin, who have made a dream come true. For a long time I had wanted to study abroad and do so in the United States. However, I had never thought that it would be possible to do so at an institution of academic excellence such as Columbia University. This opportunity has not only enriched my understanding of academia, but also my personal development in general. A major reason for this has been Julia Hirschberg's research group, who has welcomed me heartily. Especially my local supervisor, Dr. Svetlana Stoyanchev, who not only provided valuable guidance with regards to my project but also proved to be an inspirational personality. However pleasant the practical part of my thesis has been, it wouldn't have been possible without the support and trust of Prof. Gernot Kubin, who went out of his way to support my aspirations. Also, the written part of my thesis wouldn't have been possible without the both knowledgeable and patient guidance of Dr. Barbara Schuppler

A major part for such an endeavour of course is also the monetary backup which in my case was generously granted by the Austrian Marshall Plan Foundation.

Table of Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Related Work . . . 2
1.2.1 Reprise vs non-reprise questions . . . 2
1.2.2 Detection of erroneous utterances . . . 3
1.3 BOLT . . . 3
1.4 Outline . . . 4

2 Materials and ASR system **5**
2.1 TRANSTAC . . . 5
2.2 Dynaspeak . . . 7

3 Feature Extraction **10**
3.1 Features . . . 10
3.2 Prosodic Features . . . 15
3.3 Syntactic Tagging . . . 16
3.4 Recognition Tagging . . . 17
3.5 Feature Exploration . . . 18
3.5.1 Procedure . . . 18
3.5.2 Results and Discussion . . . 19

4 Modeling **20**
4.1 WEKA Framework . . . 20
4.2 Algorithms . . . 20
4.2.1 Decision Trees . . . 20
4.2.2 Multiboost Decision Trees . . . 22
4.2.3 Support Vector Machines . . . 23
4.3 Experiments . . . 24
4.3.1 Statistical measures . . . 24
4.3.2 Utterances . . . 25

4.3.3 Words 26
4.3.4 Classifier evaluation 27
4.3.5 1-stage and 2-stage error recognition 28

5 Implementation 30

5.1 The system 30
5.1.1 Goals and limitations of the system 30
5.1.2 System overview 31
5.2 Google Protocol Buffers 32
5.3 Code Setup 33
5.3.1 The Confidence Scorer 33
5.3.2 Information Structure 37
5.3.3 Results 42

6 Conclusion 43

6.1 Summary 43
6.2 Outlook 44
6.3 Conclusion 45

I Appendices 46

A Google Protocol Buffers Structure 47

B Information Extraction Scripts 49

B.1 Recognition Tagging 49
B.2 Confidence Tagging 51

II Bibliography 53

Bibliography 54

List of Figures

4.1 Example for a simple DT 21
4.2 Example of a weighted decision committee 23
4.3 Simple linear SVM example 24

5.1 SRI's system pipeline. After the original ASR is rescored and labeled by two components labeled with (UW), the new hypothesis is force aligned to obtain new timestamps before the error prediction can start. 32
5.2 UML diagram of the confidence scorer module 34
5.3 UML diagram of the common data structure 39

List of Tables

2.1 Example dialogue from the IraqComm Corpus. 6

2.2 Data composition for January release. 7

2.3 Data composition for May release. 7

2.4 Example entry for an utterance in Dynaspeak log file. 8

3.1 Overview of all features used in the experiments for section 4.3 11

3.2 Detailed list and description of explored features 12

3.2 Detailed list and description of explored features 13

3.2 Detailed list and description of explored features 14

3.3 Example of input for feature extraction. 15

3.4 Script functionality and description. 15

3.5 Example of output for extract_acoustics.pl script. 16

3.6 Snippet of Dynaspeak logfile modified to output prosodic features. 16

3.7 Example of syntactic tagging. 17

3.8 Example for the marking of differences in reference text (REF) and hypotheses (HYP). 17

3.9 Example for tagging of an utterance. 18

4.1 Precision, Recall, F-measure, overall accuracy, and % accuracy improvement over majority baseline for predicting misrecognition in an utterance. The highest value in each column is highlighted in **bold**. 26

4.2 Precision, Recall, F-measure, for predicting correctly recognized/misrecognized words, change in F-measure for predicting misrecognized words, and overall accuracy. The highest value in each column is highlighted in **bold**. 27

4.3 Classifier performance for training and test split. (c) depicts measurements on the *correct* class while (ic) depicts measurements on the *incorrect* class. MC column presents the Matthews Correlation Coefficient as an auxiliary measure of performance. 28

4.4 Precision, Recall, F-measure, and overall accuracy for correctly recognized/mis-recognized words, overall accuracy, and accuracy improvement compared to the baseline method. The highest values of each column are highlighted in **bold**. 28

5.1 Example clarification dialog. 31

Chapter 1

Introduction

1.1 Motivation

The ability to clarify information is important for successful dialog communication. Humans ask clarification questions in every-day communication whenever they believe they have misunderstood their interlocutor. Automatic Spoken Dialog Systems (SDS) also must use clarification questions to recover from Automatic Speech Recognition (ASR) errors. However, while humans are able to target their clarification questions to address the particular source of their confusion, current SDS typically do not. They make use of simple statements to indicate their lack of understanding followed by requests to the user to repeat or rephrase their input. While this behavior is generally enough to be applied to any type of hypothesized ASR error, it fails to provide the user with information about the source of that error. Such information is useful to humans in formulating responses to human misunderstandings and should be equally helpful to SDS in resolving recognition errors.

One critical requirement for producing reprise clarification questions is the detection of just which part of a user's utterance has been recognized correctly and which part or parts contain an error (*localized error detection*). Previous research on error detection in ASR in general and in SDS applications in particular has focused on identifying how likely an utterance is to have been recognized correctly or incorrectly by using ASR confidence scores, sometimes combined with acoustic and prosodic information. Such information may be used to choose another path through the ASR lattice or to request repetition or rephrasing of the utterance by the user.

Goals A Speech-to-Speech (S2S) system takes speech input, recognizes it automatically, translates the recognized input into text in another language, and produces synthesized speech output from the translation for the conversational partner. The S2S application we target, allows speakers to converse freely about topics that are not specified in advance. In the case of a hypothesized ASR error, the clarification component of the system seeks to clarify errors with the speaker before passing a corrected ASR transcription on to the MT component. In this way, the clarification component attempts to identify and correct speech recognition errors early in the dialog to avoid translating poorly recognized utterances.

The work presented here, aims at (1) identifying which utterances have been misrecognized and (2) which *portions* of utterances have been incorrectly transcribed by the recognizer. This information is used to formulate targeted reprise questions.

1.2 Related Work

1.2.1 Reprise vs non-reprise questions

In his study of human clarification strategies, Purver [2004] distinguishes two types of clarification questions: *reprise* and *non-reprise* questions. He defines a reprise clarification question as one that asks a targeted question about the part of an utterance that was misheard or misunderstood, including portions of the misunderstood utterance which are thought to be correctly recognized. A non-reprise question, on the other hand, is a generic request for repetition, which does not contain contextual information from the misunderstood utterance. Both are illustrated in the example below:

Speaker:	Do you have anything other than these XXX plans?
Reprise:	What kind of plans?
Non-Reprise:	What did you say?/Please repeat.

While clarification questions in informal human conversations contain only about 12% *non-reprise* clarification questions [Purver, 2004], most SDS use *only* non-reprise clarification strategies, asking users to repeat or rephrase when the system hypothesizes a recognition error. Non-reprise clarification questions are easy to construct and are well-suited to simple slot-filling dialog systems where speakers are required to specify values for a fixed number of predefined attributes and concepts. However, other researchers have found that the naturalness of system prompts have an important effect on a user's perception of the system's behaviour and performance [Lopes *et al.*, 2011; Stoyanchev and Stent, 2009]. As we move towards systems that support mixed initiative and eventually user initiative, such as tutoring systems [Litman and Silliman, 2004] and speech to speech translation systems [Akbacak and others, 2009], it becomes more and more relevant to develope SDS which can request more specific information about hypothesized ASR errors.

1.2.2 Detection of erroneous utterances

Handling errors in SDS involves to first determine whether an error has probably occurred and then choosing an appropriate dialog strategy to correct it. There has been considerable work on detecting erroneous utterances in ASR systems and, more specifically, in SDS. Bohis and Rudnicky [2005] analyse tradeoffs between misunderstandings and false rejections in a dialog system. The authors optimize rejection thresholds using data-driven methods. Lopes et al.[2011] also analyse different feature sets for improving confidence score estimation in a dialog system. Komatani and Okuno [2010] use a user's utterance history to determine whether a barge-in user utterance has been correctly recognized. The usage of prosodic features in this thesis is motivated by Hirschberg et al.[2004] who found that prosodic features alone and in combination with other automatically available features improve significantly over simple acoustic confidence scores alone in identifying misrecognized utterances. However, these authors did not address the problem of identifying *which* word(s) in the utterance were misrecognized. Goldwater et al. [2010] found evidence that some words are harder to recognize than others due to their prosodic characteristics, the position they occur in in a turn, their use as *discourse markers*, their location preceding disfluencies, or their confusability with words having similar language model probabilities and similar phonetic make-up. They also found that speaker variability was a considerable source of recognition error. However, these authors do not address the question of how to use these characteristics to predict which words are misrecognized. This thesis addresses this problem: how can we identify misrecognized words accurately in an SDS using automatically extracted, speaker independent features.

There has also been considerable research on determining dialog strategies for error recovery. For example, Dzikovska et al. [2009] describe an approach to dealing with errors in tutoring dialog systems (a dialog system used top provide instructions to the user). Bohus et al. [2006] use supervised learning to determine the optimal error recovery policy in a dialog system, such as providing a help message, repeating a previous prompt, or moving on to the next prompt. This thesis is a study towards introducing a new policy type in a dialog system: *asking a targeted clarification questions.*

1.3 BOLT

This thesis was carried out as part of the 'Broad Operational Language Translation' *(BOLT)* program, funded by the 'Defense Advanced Research Projects Agency' *(DARPA)* of the 'United States Department of Defense'. BOLT's goals are the translation of informal language genres and Bilingual, multi-turn conversation (both on text and speech level).

To achieve flexibility with regard to conversational topics, handling dialectal variations and being able to handle more than single sentences while also guaranteeing reliability in translation accuracy, the program is organized by:

- Three Technical Areas 1. Algorithmic Development and Integrated Systems 2. Data Collection 3. Evaluation
- Six Activities per Technical Area (Except Data) A. Translation and Information Retrieval B. Human-Machine Dialog Systems C. Human-Human Dialog Systems D. Arabic Dialect Translation E. Grounded Language Acquisition F. Basic Technologies

This thesis represents parts of the work carried out by a multi-institutional team lead by SRI International and is concerned with technical area 1, activity B of the BOLT program. All participating sites had as a baseline for data to train on both the *TRANSTAC* (TRANSlation system for TACtical use) and *GALE* (Global Autonomous Language Exploitation) data sets.

1.4 Outline

This thesis is organized as follows: in chapter 2, an overview of used materials will be given. In chapter 3, all explored features for error detection are presented as well as the final set of features used in the subsequently presented experiments. Chapter 4 presents the machine learning algorithms employed as well as the results from using these different algorithms with the feature sets presented in chapter 3. Chapter 5 provides a design overview of both the overall system as well as the implementation of the error-detection component (utilizing classifiers built in chapter 4) into this system. Parts of this thesis have been published in Stoyanchev et al. [2012] and they appear throughout the chapters in reformatted paragraphs.

Chapter 2

Materials and ASR system

This chapter presents an overview of materials used in the thesis with regards to speech data sets abailable for system development (TRANSTAC) and research experiments as well as the ASR system (Dynaspeak) used to process the audio data.

2.1 TRANSTAC

The *Spoken Language Communication and Translation System for Tactical Use (TRANSTAC)* program was the predecessor to today's BOLT program. Data collected by the National Institute of Standards and Technology *(NIST)* during seven months of evaluation exercises performed between 2005 and 2008 form the basis for the development done under BOLT [Weiss and others, 2008]. The data was collected using SRI's *IraqComm* speech-to-speech translation system [Akbacak and others, 2009]. The corpus contains simulated dialogues between English military personnel and Arabic interviewees. Thus, the audio is clean of noise and was recorded using high-performance audio equipment to ensure highest possible usability for context dependent experiments. When an English speaker speaks, the system's ASR component recognizes the utterance, performs machine translation to translate it into (Iraqi) Arabic, and uses a text-to-speech synthesis (TTS) system to produce the Arabic version. When the Arabic speaker replies, the procedure is reversed. Table 2.1 shows a sample dialogue from the dataset, with correct English translations for the Arabic utterances.

English:	good morning
Arabic:	good morning
English:	may i speak to the head of the household
Arabic:	i'm the owner of the family and i can speak with you
English:	may i speak to you about problems with your utilities
Arabic:	yes i have problems with the utilities

Table 2.1: Example dialogue from the IraqComm Corpus.

For experiments and development, two different subsets were provided by NIST. These subsets will be referred to as *January release* and *May release*. Both releases feature only male speakers in an optimal, laboratory style environment. This ensures absolutely noise-free recordings.

The January release includes English and Arabic speech with manual transcriptions. We use only the audio and manually annotated transcript (as the reference) of the English utterances for the experiments presented in this thesis. We removed utterances in which a user directed a command to the computer, such as *Computer, repeat*. We also removed instances with a difference in ASR and transcript due to annotation, (e.g., such as contractions *we're* and *we are*) in order to avoid difficulties in string matching.

The resulting corpus contains a total of 3.7K utterances and 26K words. 28.6% of utterances and 9.1% of words contain an ASR error (Table 2.2). These numbers are based on ASR results obtained by running the Dynaspeak ASR system (see section2.2) release provided with the January release of TRANSTAC data.

The May release is again divided into two subsets, forming a *development set* and a *test set*. Similarly to the January release, both English and Arabic transcriptions were available, where only the English ones were used. Due to ongoing development and tuning of the BOLT System, multiple versions of the Dynaspeak ASR were used. Table 2.3 represents numbers obtained by running the latest (as of June 2012) available Dynaspeak version. As the names suggest, the development set was used for training and tuning purposes across BOLT sites while the test set served as means of obtaining realistic performance numbers.

	Overall	**Correct ASR**	**Error in ASR**
All Utterances	3.729	2.664 (71.4%)	1.065 (28.6%)
All Words	26.098	23.720(90.9%)	2.378(9.1%)
Words in erroneous Utterances.	7.48	5.45 (72.8%)	2.03 (27.2%)

Table 2.2: Data composition for January release.

	Total Words	**Correct Words**	**Erroneous Words**
Development Set	41.801	39.033 (93,4 %)	2.768 (6,6%)
Test Set	37.354	34.927(93,5%)	2.427 (6,5%)

Table 2.3: Data composition for May release.

2.2 Dynaspeak

Dynaspeak is an ASR engine developed and distributed by SRI International [Franco and others, 2002]. It is currently used in industrial, consumer, and military products and systems. It serves as the ASR component deployed in current field units of the IraqComm system which forms the conclusion of the TRANSTAC program. Core features are:

- Hidden Markov Model (HMM)-based speech recognizer. HMMs form a statistical approach to speech recognition. The core functionality is to predict the most probable meaning of speech based on previously observed patterns.

- Supports continuous speech. This enables the user to talk freely without having to provide any indication of beginning and ending of the input.

- Dynamic grammar compilation. The grammar used by the recognizer can be changed and adapted on the fly, during run-time. This allows for quick adaptations to the recognition engine.

- Speaker independent. The ASR does not discriminate between types of users (male/female, young/old, ...) and accuracy is similar for all users.

- Speaker adaptation. The more the ASR is subject to input by a user, the more it will adapt internal modeling to improve accuracy.

- Dynamic noise compensation. The ASR is able to distinguish between user issued speech and environment noise and will compensate for the latter to improve accuracy.

Dynaspeak was used as an as-is application for this thesis as other than providing input for new features to the development team at SRI, no possibility of changing the functionality of components was available. Thus, no modifications to the ASR system were made by the author, but suggestions were made to the responsible development team which then partly were implemented in future releases. The basic usage of Dynaspeak was in providing an input script (part of the application) with a list of audio files and corresponding transcriptions. After the successful recognition process, a log file with following information was available (listed are only fields containing information relevant for this thesis, **bolded** names represent the field names in the log file):

- Utterance ID (field **SENTENCE**)
- File ID (field **FILENAME**)
- Start and end time of word (field **INFO: Alignment**)
- Final ASR hypothesis (field **HYP**)
- Reference text (field **REF**)
- Prescinded information for misrecognitions (insertions, deletions, word swaps) (in both REF and HYP)
- Per-word ASR confidence/posterior (field WORD POSTERIORS)
- Start time for each word (field TIMES)

An example for such a log file can be seen in table 2.4

SENTENCE:	21
FILENAME:	/proj/speech/projects/bolt/.../scen01_oovnne_009.wav
INFO: Alignment	'(-pau- 0 1 pr:-448 gp:-280 cf:0)....((us 266 291)..
REF:	that TRAFFICKER CREPT INTO THE city without us knowing
HYP:	that ********** TRAFFICKERS CRYPTOGRAPHY TO city without us knowing
ERROR:	0 ins 1 del 3 sub 9 wds 44.44% err
TOTAL:	29 ins 2 del 38 sub 173 wds 39.88% err 100.00% sent
WORD POSTERIORS:	that|0.909795 traffickers|0.186832 cryptography|1 to|1 city|0.856397 without|1 us|1 knowing|1

Table 2.4: Example entry for an utterance in Dynaspeak log file.

Chapter 3

Feature Extraction

In the following chapter, all features used in the machine-learning experiments in chapter 4 will be explained as well as how these features were extracted from available data (see chapter 2).

3.1 Features

For experiments, features were available (through extraction presented in sections 3.4, 3.2, 3.3) for both utterance- and word level. A summary of features is presented in Table 3.1. Also denoted in this table are the feature subset affiliations for the acronyms *ASR*, *PROS* and *SYN* which will be used to refer to these subsets. ASR represents posteriors as calculated by the speech recognizer. PROS represents prosodic features which are measurable physical attributes of an audio signal such as frequency and energy their derivatives. SYN refers to syntactic features such as parts-of-speech (see section 3.3). An extensive list of explored features including descriptions can be seen in table 3.2.

Feature type	**Description**	**Utterance-correctness classification experiment**	**Word-correctness classification experiment**
ASR	log of posterior probability	average over all words in hypothesis	in current word; avg over 3 words; avg of all words
Prosodic features (PROS)	F0(MAX/MIN/MEAN/STDEV)	for whole utterance	for word
	RMS(MAX/MIN/MEAN/STDEV)	for whole utterance	for word
	proportion of voiced segments	in whole utterance	in current word
	duration	of an utterance	of current word
	timestamp of beginning of first word	used	not used
	speech rate	over all utterance	not used
Syntactic features (SYN)	POS tags	count of unigram/bigram	this/previous/next word
	word type (content/function)	not used	this/previous/next word

Table 3.1: Overview of all features used in the experiments for section 4.3

All Features		
Group	Name	Description
	ASRconfidence	Word ASR posterior for current word.
ASR	**ASRconfidenceAvg3**	Word ASR posterior averaged over preceding, current and next word.
	ASRconfidenceAvgAll	Word ASR posterior averaged over entire utterance.
	POSTAGTHIS	Stanford part of speech tag for the current word.
	POSTAGPREV	Stanford part of speech tag for the preceding word.
	POSTAGNEXT	Stanford part of speech tag for the following word.
Syntactic	HIGHTAGTHIS	Content tag for the current word.
	HIGHTAGPREV	Content tag for the preceding word.
	HIGHTAGNEXT	Content tag for the following word.
	analysed_dur	Analysed signal duration in ms.
	total_dur	Signal duration in ms. (Redundant due to analysed_dur).
	F0MIN	Minimum of pitch in signal with cutting outliers (highest and lowest 5%).
	F0MAX	Maximum of pitch in signal with cutting outliers (highest and lowest 5%).
	F0MEAN	Mean of pitch in signal with cutting outliers (highest and lowest 5%).
	F0MED	Median of pitch in signal with cutting outliers (highest and lowest 5%).
	F0STDEV	Standard deviation of pitch in signal with cutting outliers (highest and lowest 5%).
	minF0_NOSMOOTH	Minimum pitch without cutting outliers (highest and lowest 5%).
	maxF0_NOSMOOTH	Maximum pitch without cutting outliers (highest and lowest 5%).
	meanF0_NOSMOOTH	Mean pitch without cutting outliers (highest and lowest 5%).
	medianF0_NOSMOOTH	Median pitch without cutting outliers (highest and lowest 5%).
	sdF0	Standard Deviation without cutting outliers (highest and lowest 5%).
	ENGMIN	Minimum of energy in signal with cutting outliers (highest and lowest 5%).
	ENGMAX	Maximum of energy in signal with cutting outliers (highest and lowest 5%).
	ENGMEAN	Mean of energy in signal with cutting outliers (highest and lowest 5%).

Table 3.2: Detailed list and description of explored features

Prosodic

	ENGSTDEV	Standard deviation of energy in signal.
	VCD2TOT	The fraction of pitch frames that are analysed as unvoiced in the analysed audio.
	pctUnvoi	The fraction of pitch frames that are analysed as unvoiced in the analysed audio (Redundant due to VCD2TOT).
	HNR	Harmonics-to-Noise Ratio (HNR). Harmonicity is expressed in dB: if 99% of the energy of the signal is in the periodic part, and 1% is noise, the HNR 20 dB. A HNR of 0 dB means that there is equal energy in the harmonics and in the noise.
	NHR	Inverse to the HNR.
	autocor	Mean autocorrelation coefficient of the signal.
	shimmerapq11	The 11-point Amplitude Perturbation Quotient, the average absolute difference between the amplitude of a period and the average of the amplitudes of it and its ten closest neighbours, divided by the average amplitude.
	shimmerapq5	The five-point Amplitude Perturbation Quotient, the average absolute difference between the amplitude of a period and the average of the amplitudes of it and its four closest neighbours, divided by the average amplitude.
	shimmerapq3	The three-point Amplitude Perturbation Quotient, the average absolute difference between the amplitude of a period and the average of the amplitudes of its neighbours, divided by the average amplitude.
	shimmerlocDB	The average absolute base-10 logarithm of the difference between the amplitudes of consecutive periods, multiplied by 20.
	shimmerloc	The average absolute difference between the amplitudes of consecutive periods, divided by the average amplitude.
	jitterppq5	The five-point Period Perturbation Quotient, the average absolute difference between a period and the average of it and its four closest neighbours, divided by the average period.

Table 3.2: Detailed list and description of explored features

	jitterrap	The Relative Average Perturbation, the average absolute difference between a period and the average of it and its two neighbours, divided by the average period.
	jitterlocabs	The average absolute difference between consecutive periods, in seconds.
	jitterloc	The average absolute difference between consecutive periods, divided by the average period.
	pctVoicebreaks	The total duration of the breaks between the voiced parts of the signal, divided by the total duration of the analysed part of the signal.
	nvoicebreaks	The number of distances between consecutive pulses that are longer than 1.25 divided by the pitch floor. Thus, if the pitch floor is 75 Hz, all inter-pulse intervals longer than 16.67 milliseconds are regarded as voice breaks.
	sdPeriod	Standard deviation of lengths of periods.
	meanPeriod	Mean length of periods.
	nPeriods	Number of different periods in the signal.
	nPulses	Number of pulses in the signal.

Table 3.2: Detailed list and description of explored features

3.2 Prosodic Features

For the January released data (see section 2.1), we extracted prosodic features from the audio file of each utterance using Praat scripts [Boersma, 2001]. These scripts were based on existing scripts with slight modifications. Features from both scripts offered redundancy for some features (e.g., duration) as well as different measurement methods for other features (e.g., smoothed values versus non-smoothed values). This redundancy was designed on purpose in order to test as many features as possible concerning their information gain. Functionality of both scripts as well as an example output for one script are presented in tables 3.4 and 3.5. Both scripts are called simultaneously. They use information regarding start and end time of words extracted by another script, which analyzes Dynspeak logfiles and extracts information. Among others, this script delivers start and end times for each word. Table 3.3 shows an example for such word alignment, which serves as input for both prosodic scripts.

evalTranstac-0508-live-004.wav	0.01	0.66	who
evalTranstac-0508-live-004.wav	0.67	1.52	places
evalTranstac-0508-live-004.wav	1.53	1.78	the
evalTranstac-0508-live-004.wav	1.79	2.4	roadside
evalTranstac-0508-live-004.wav	2.41	2.89	bombs

Table 3.3: Example of input for feature extraction.

Script name	Function and output format
extract_acoustics.pl	-runs a Praat script to extract pitch and energy. -results are saved in 1 FILE PER WORD. -filenames are composed as 'UtterancefileidWordnumber.txt' with Wordnumber beginning at '0' (thus 'evalTranstac-0603-online-1401.txt' is the SECOND word in evalTranstac-0603-online-140.wav).
voice-report.praat	-runs a praat script to extract pitch, energy, shimmer, jitter. -results are saved in 1 FILE FOR ALL WORDS. -filename is 'info-wid.txt'. -each line in 'info-wid.txt' contains the same information as the Praat's standard voice-report PLUS the word ID for each file, starting with 0 for the first word in each utterance.

Table 3.4: Script functionality and description.

F0_MIN: 397.045
F0_MAX: 484.536
F0_MEAN: 451.015
F0_STDV: 34.865
ENG_MAX: 37.689
ENG_MIN: 21.995
ENG_MEAN: 31.060
ENG_STDV: 4.801
VCD2TOT_FRAMES:0.325
WORD:who

Table 3.5: Example of output for extract_acoustics.pl script.

For the May released data (see section 2.1) the task of prosodic feature extraction was carried out by SRI via a modification to the Dynaspeak recognizer which allowed for the needed information to be presented directly in Dynaspeak logfiles (see table 3.6).

FINAL RESULT:	is it couldn't be anywhere near five hundred
is	frames 0-15
F0	mean=118.404515 min=107.349098 max=143.991821 stdev=10.409563
RMS	mean=231.864628 min=0.000000 max=595.293701 stdev=189.380846
Voiced proportion	0.687500

Table 3.6: Snippet of Dynaspeak logfile modified to output prosodic features.

3.3 Syntactic Tagging

Syntactic Tagging for both part-of-speech (*POS*) as well as content-/noncontent-words (*CNT*) was done by another project using the Stanford POS Tagger [Toutanova *et al.*, 2003]. On the utterance level, both syntactic features were represented as unigrams (how often a tag occurs in the utterance) and bigrams (how often a certain pair of tags occurs in the utterance). Table 3.7 shows an example of the tagged output for the word level. The table shows POS as well CNT tags for current, previous and next words. NULL meaning the information is not available (e.g., there was no previous/next word in the utterance). The CNT tag can either be FNC (for function words) or CNT (for content words). POS tags can be any part of speech tag (e.g., VBZ - Verb, 3rd person singular present).

evalTranstac-0508-live-004.wav	0	who	WP	CNT	NULL	NULL	VBZ	CNT
evalTranstac-0508-live-004.wav	1	emplaces	VBZ	CNT	WP	CNT	DT	FNC
evalTranstac-0508-live-004.wav	2	the	DT	FNC	VBZ	CNT	NN	CNT
evalTranstac-0508-live-004.wav	3	roadside	NN	CNT	DT	FNC	NNS	CNT
evalTranstac-0508-live-004.wav	4	bombs	NNS	CNT	NN	CNT	NULL	NULL

Table 3.7: Example of syntactic tagging.

3.4 Recognition Tagging

Recall the basic goal of this thesis: the development of methods for reliable classification of ASR output as either *correct* (recognition is equal to what was said) or *incorrect* (recognition is not equal to what was said i.e. word substitution). To be able to train and test such classifiers, the available data has to be pre-tagged as being part of either *class* in order to provide a measure as to how well a classifier actually performs.

As mentioned in chapter 2, Dynaspeak output files contained two lines presenting both final ASR hypotheses as well as the actual transcription. Already encoded in this representation are misfits in the form of capitalized letters as well as '*' characters. To take advantage of this information, a script was created (see appendix B.1) to tag a word in the final hypothesis as either *correct* or *incorrect*. Tables 3.8 and 3.9 present an example as to how words would be classified based on logfile information. Note that the deletion of words occurring in the transcript is not accounted for since such information would not be available in a live-system. The utterances' tag depend on to the tags of the words they contain. An utterance is classified as *incorrect*, if one or more words in the utterance are also tagged as such.

REF:	that TRAFFICKER CREPT INTO THE city without us knowing
HYP:	that ********** TRAFFICKERS CRYPTOGRAPHY TO city without us knowing

Table 3.8: Example for the marking of differences in reference text (REF) and hypotheses (HYP).

Word	Tag
that	correct
TRAFFICKERS	incorrect
CRYPTOGRAPHY	incorrect
TO	incorrect
city	correct
without	correct
us	correct
knowing	correct

Table 3.9: Example for tagging of an utterance.

3.5 Feature Exploration

3.5.1 Procedure

Analyzing the available set of features and choosing the most significant of those was done using a simple 'Hill-climbing' algorithm. The idea behind the algorithm was to find significant features by eliminating insignificant or even harmful ones. This was done by calculating the overall error of the entire dataset and then eliminating one feature at a time and re-measuring the error. The second method for testing features was starting with one feature and step-wise adding other features to the inspected list.

Both approaches were done by both starting the algorithm once from the top of the list of features as well as once from the bottom. The error can go in one of the following directions:

- The error increases/accuracy decreases, implicating that the feature removed was actually significant towards a better classification. In this case, the feature will be retained in following iterations.
- The error does not change. In case the error does not change, the feature is deemed insignificant and will be excluded for all future runs.
- If the error decreases, the feature seems to have hurt the performance of the classificator and will thus be exluded in all future runs. Also, future performances will be measured against the decreased error.

The error of the entire dataset and any given subset of features was measured using three different scores: F-measure (see section 4.3) for both 'correct' and 'incorrect' labeled words (thus providing two scores) and accuracy representing the percentage of correctly classified instances in the dataset.

3.5.2 Results and Discussion

Interestingly the final set of relevant features was the same for all three evaluated scores, thus presenting a stable and robust set of features independent of the evaluation method. The scores for this dataset show an accuracy of 0.845, F-measure for *correct* of 0.885 and F-measure for *incorrect* of 0.669. As expected, prosodic features helped in improving classification performance as well as POS tags as both present a natural extension of acoustic models and language models used in ASR. Slightly surprising though was the complete lack of impact of fine-grained features like shimmer and jitter. A possible explanation may be found when taking into account the fact that data presented in experiments represented multiple speakers. So, while it may be possible to find shimmer and jitter patterns influence misrecognition rate for a single speaker, a general pattern true for multiple speakers could not be found though. For later experiments, the features nPeriods, meanPeriod, sdPeriod, jitterloc and NHR where dropped from further evaluation due to minimal improvements (ranging in the ‰area), and difficulty of extraction using the built-in mechanisms of Dynaspeak ASR.

Chapter 4

Modeling

This chapter is concerned with the creation of a classifier to predict whether or not ASR output is correct or not. First, three different machine-learning algorithms are explored and applied to the data and to the features gathered in chapters 2 and 3. Then, different performance measure method are used to determine the most useful classifier.

4.1 WEKA Framework

The Waikato Environment for Knowledge Analysis *(WEKA)* is a collection of machine learning algorithms for data mining tasks developed at the Machine Learning Group at the University of Waikato, New Zealand. The algorithms can either be applied directly to a dataset or called from your own Java code. Weka contains tools for data pre-processing (e.g., for value editing), classification, regression (e.g., for finding dependant feature in a dataset), clustering (e.g., for associating data values with different statistical distributions), association rules, and visualization (e.g., for visualizing data arrangement in an N-dimensional space) [Witten and Eibe, 2005].

WEKA was chosen as the framework basis for all experiments as well as the tool with which the final classifiers were built, among other reason, due to the ease of implementation into existing software given the Java based base package. Also, for experimental purposes, the built-in GUI provides the possibility of quickly visualizing, evaluating and comparing different types of classifiers or parameter sets.

4.2 Algorithms

4.2.1 Decision Trees

Decision Trees form the simplest way of building a classifier. The tree starts from a central node, the *root*, which also forms the top layer of the tree. Starting from the root, every following node underneath it leads to two child nodes. The only exception to this rules are terminating nodes - or *leafs*, which represent a final outcome or decision. The connection from one node to another is made via *branches*. Branches represent values upon which the path to the next node is determined. In the application of machine learning, decision trees are often referred to as *classification trees* and are built by mapping observations onto how final states were achieved in a given machine learning problem to paths in the tree. And example of a decision tree can be seen in figure 4.1].

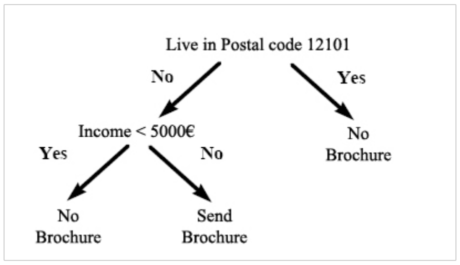


Figure 4.1: Example for a simple DT

Several training algorithms exist as to how such mapping should be done- the simplest being a straight mapping of observed paths given multiple iterations through a problem. This however, is not feasible since such an approach will result in a robust classifier only for the most mundane tasks. WEKA offers a set of established algorithms to generate decision trees (DTs). The most common and well established of those, and also the one which will be used for experiments throughout this thesis, is the *J48* or *C4.5* algorithm. This algorithm is based on step-wise minimization of *entropy* or uncertainty in the tree by adding high-information (low-entropy) attributes of the presented data set as nodes to the tree.

In case of the J48 algorithm, this is done by first analyzing the set of training data available, which is represented as classified examples of feature or attribute vectors. At each node of the tree, J48 chooses one attribute of the data- which most effectively (highest information gain) splits its training set into subsets of classes - as the node attribute. This is also an implementation of the *divide and conquer* principle in machine learning. In addition to this basic principle, the algorithm has three base cases [Quinlan, 1993]:

1. All the (remaining) samples belong to the same class. In this case, J48 simply creates a leaf choosing that class.

2. None of the available/remaining features provide any information gain. In this case, J48 creates a node higher up the tree using the expected value of the class.

3. The same step is taken in the case of encountering a previously unseen class.

In this thesis, decision trees are used for both feature exploration (see chapter 3) as well as for error detection.

4.2.2 Multiboost Decision Trees

Boosting refers to a technique which proposes the use of not just one single classifier to solve a classification problem, but the use of an ensemble or *committee* of such classifiers, where each classifier is to be considered "weaker" (i.e. not as accurate) than an otherwise used single classifier. This is done by using a base learning algorithm (like J48) and providing it with a sequence of training sets which the boosting algorithm synthesizes from the original training set. The resulting classifiers become members of a decision committee, where, in the simplest case, the class with the most votes will be the outcome. A graphical example can be seen in figure 4.2.

For this thesis, a more sophisticated boosting algorithm was chosen, which was also easily available in the WEKA framework - the *MultiBoostAB* method. MultiBoostAB is an extension to the highly successful *adaptive boosting* or *AdaBoost* [Freund and Schapire, 1995] technique for forming decision committees by combining the AdaBoost algorithm with *wagging* [Bauer and Kohavi, 1999].

Wagging is a variant of *bagging* [Breiman, 1996]. Bagging is an ensemble method that creates individual training sets for its ensemble members by random redistribution of the training set. Each classifier's training set is generated by randomly drawing examples of the original training set. One disadvantage of this method is that many of the original examples may be repeated in the resulting training set while others may be left out because each set has to contain the same number of examples as the original. Wagging differs from bagging in that it does not draw random samples but instead assigns a random weight to each example in the training set. Hence, the original name *weighted bagging*, then shortened to *wagging*. Both wagging and bagging do not use weights for their classification decision, but each classifier has equal influence on the output.

AdaBoost, similar to wagging, assigns weights to each of the examples contained in an original training set. With AdaBoost, however, the probability of picking each example is initially set to be 1/N, where N is the total number of samples available in the training set. These probabilities are recalculated after each trained classifier is added to the ensemble, based on the performance of the newly added classifier. AdaBoost combines classifiers using weighted voting, allowing AdaBoost to discount the predictions of classifiers that are not very accurate on the overall problem.

MultiBoost's motivation to combine both methods is based on observations, showing that wagging is effective in reducing the variance of resulting classifiers while AdaBoost succeeds in reducing the bias of classifications [WEBB, 2000]. To benefit from both characteristics, finding a way of combining both algorithms seemed desirable. However, while AdaBoost weights the votes of its committee members, bagging does not, thus making the votes of members of each committee incompatible. An alternative way was found by bagging a set of sub-committees, each formed by application of AdaBoost. Thus, MultiBoosting can be considered as wagging committees formed by AdaBoost.

Throughout this thesis, when referring to classifiers trained with MultiBoost, J48 decision trees were used as a base classifier for the MultiBoost algorithm.

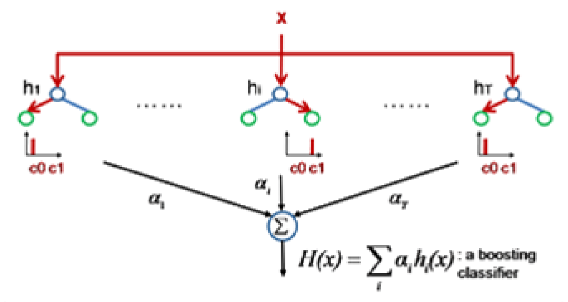


Figure 4.2: Example of a weighted decision committee

4.2.3 Support Vector Machines

The third method for building binary classifers were *Support Vector Machines (SVM)* [Cortes and Vapnik, 1995]. In its most basic form, an SVM solves a 2-dimensional problem of linearly separable classes by positioning a linear separator such that each distance d, measured as the length of a normal drawn from the separator to a data point i, minimizes the overall distance

$$\mathbf{D} = \sum d_i. \tag{4.1}$$

The resulting separator is called a *hyperplane*. The overall distance $\mathbf{D}$ is called the *margin*. Thus, the result of any SVM is the *maximum margin hyperplane* separating any two classes in a feature vector with dimensionality higher than one. An example of such a separator can be seen in figure 4.3.

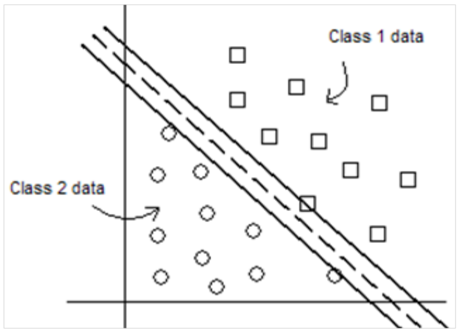


Figure 4.3: Simple linear SVM example

To train SVMs for error detection, the *sequential minimal optimization (SMO)* [Platt, 1998] is used in this thesis. SMO splits the potentially very large optimization problem for finding a suitable maximum margin into a series of smaller problems (divide and conquer algorithm). This avoids a quadratic programming problem and makes the solution analytically computable.

A core feature of SVMs is the *Kernel trick*, which allows the algorithm to obtain results in high-dimensional spaces without having to explicitly compute such results. This is achieved by using a Kernel to map results from a low-dimensional space into higher-dimensional space. For this thesis, a polynomial Kernel as shown in formula 4.2 was used. The exponent $p = 3$ was empirically chosen to allow for reasonable training and re-training times in case of short-notice data changes.

$$K(x, y) = \langle x, y \rangle^p \tag{4.2}$$

4.3 Experiments

4.3.1 Statistical measures

In order to identify the best performing feature set for each of the classifiers we separately evaluate performance of (1) misrecognized utterance prediction and (2) misrecognized word prediction. To be able to measure the quality of a classifiers, several indicators will be used:

- Accuracy; depicts the overall percentage of correctly classified instances. An accuracy of 80% means, that the classifier has correctly classified 80 samples out of 100. As simple as that may be, it is very dangerous to rely on just accuracy for measuring the quality of a classifier if the number of instances of classes in a dataset is highly skewed toward one. For instance, a dataset contains 100 samples of class "X" but only 5 samples of class "Y". If every sample in the dataset is classified as being part of "X", the accuracy would be quite high at 95% but the classifier would have misclassified every single instance of class "Y".

- Precision; a classifiers precision measures the amount of samples, that were correctly assigned to a class versus those that were incorrectly assigned the same class. For instance, if a classifiers predicts 100 samples of a dataset as being part of class "X", but 20 of those samples actually belong to class "Y", the precision for that class would be 0.8.
- Recall; this measure represents the fraction of samples of a certain class in a dataset that were correctly classified as such by the classifier. For instance, if a dataset contains 100 samples of class "X" and a classifier recognizes 80 of those samples as part of class "X", the recall for this class would be 0.8.
- F-measure; also known as F1-measure. F-measure is a simple way of combining both precision and recall of a class into one measure (see formula refform:fmes).

$$F1 = \frac{2 * recall * precision}{(recall + precision)} \tag{4.3}$$

- Matthews Correlation Coefficient; similar to F-measure, the MCC combines different results into one compact measure. However, contrary to F-measure (and even more so contrary to accuracy), the MCC is specifically designed to deal with highly skewed datasets. The MCC is calculated as seen in formula 4.4. TP stands for *True Positives* or samples correctly classified as being part of class "X", TN stands for *True Negatives* or samples correctly classified as NOT being part of class "X" (and thus are indeed part of class "Y"). FN and FP stand for *False Positives/Negatives* and are thus incorrectly classified samples.

$$MC = \frac{(TP * TN) - (FP * FN)}{\sqrt{(TP + FP) * (TP + FN) * (TN + FP) * (TN + FN)}} \tag{4.4}$$

4.3.2 Utterances

To evaluate a 2-stage approach to error detection, where in a first classification run potentially misrecognized utterances are identified to be later on checked by a word level error detector, first we have to evaluate how well different feature sets will perform on the utterance level. These experiments were run on the entire January release data (see chapter 2) using 10-fold cross-validation with a J48 classifier. Table 4.1 shows precision, recall, and F-measure for predicting correctly recognized and misrecognized utterances; improvement in F-measure of our classifier over a classifier using only ASR confidence scores; and overall prediction accuracy.

The majority class baseline (always predicting correct recognition) achieves 71.4% overall accuracy — i.e., failing to detect any incorrectly recognized utterances. Using ASR confidence features alone, we increase overall accuracy to 79.4% with an F-measure for predicting correctly recognized/misrecognized instances of .86/.60, respectively. Contrary to our expectation, a combination of ASR confidence and prosodic features (ASR+PROS) does not improve this performance. However, syntactic features in combination with ASR confidence (ASR+SYN) is the highest performing predictor across all measures. A classifier trained with (ASR+SYN) achieves 83.8% accuracy with F-measures of .89 for correcto and .68 for incorrect words. Since one of the aims was to create targeted clarification questions, of particular interest was to increase the F-measure for detection of misrecognized utterances. We observed that by adding syntactic features to ASR features, the F-measure for detecting misrecognized utterances increases by 13.3%.

Feature	Utt Correctly Rec.			Utt Misrec			F1 *incorrect*	Overall
	P	**R**	**F**	**P**	**R**	**F**	compared to ASR	Accuracy
Maj. Base.	.71	1	.83	-	0	0	-100%	71.4%
ASR	.83	.90	.86	.68	.53	.60	0	79.4%
ASR+RROS	.82	.89	.85	.65	.51	.57	-.05	78.1%
ASR+SYN	**.86**	**.93**	**.89**	**.77**	**.61**	**.68**	**+13.3**	**83.8%**

Table 4.1: Precision, Recall, F-measure, overall accuracy, and % accuracy improvement over majority baseline for predicting misrecognition in an utterance. The highest value in each column is highlighted in **bold**.

4.3.3 Words

This experiment is run on a subset of the data with the words from misrecognized utterances known from the reference transcription of the January release data (see chapter 2) to contain errors. In this dataset, 27.2% of words are misrecognized. We perform a 10-fold cross-validation experiment on this subset of the data. Table 4.2 shows precision, recall, and F-measure for predicting correctly recognized and misrecognized words in utterances known to be misrecognized, improvement in F-measure of misrecognized word prediction over a classifier that uses only ASR features, and overall accuracy of prediction.

The majority class baseline (predict correct recognition) achieves 72.8% overall accuracy, failing to detect any of the incorrectly recognized words. Using the ASR confidence features alone achieves an F-measure for predicting correctly recognized/misrecognized words of .86/.50 respectively. ASR confidence scores together with prosodic features (ASR+PROS) improve the F-measure for predicting misrecognized words to .54. We observe that prosodic features are very useful in predicting misrecognized words, raising F-measure by 8%. A combination of all features (ASR+PROS+SYN) is the highest performing predictor across all measures except for recall on correctly recognized words. The performance of a classifier trained on ASR+PROS+SYN features reaches an F-measure of .90/.70 and overall accuracy of 84.7%. Prosodic and syntactic features account for an increase of 40% for predicting

Feature	*correct*			*incorrect*			F1 *incorrect*	Overall
	P	**R**	**F**	**P**	**R**	**F**	compared to ASR	Accuracy
Maj. Base	.73	1	.84	-	0	0	-100%	72.8%
ASR	.81	**.93**	.86	.69	.40	.50	0%	78.7%
ASR+PROS	.82	.92	.86	.67	.46	.54	+8%	79.0%
ASR+PROS+SYN	**.87**	**.93**	**.90**	**.76**	**.64**	**.70**	**+40%**	**84.7%**

Table 4.2: Precision, Recall, F-measure, for predicting correctly recognized/misrecognized words, change in F-measure for predicting misrecognized words, and overall accuracy. The highest value in each column is highlighted in **bold**.

misrecognized words compared to the classifier that uses only ASR features.

In sum, the experiments presented in this section show, that the best performing feature combination for predicting misrecognized utterances is ASR+SYN and for words ASR+PROS+SYN.

4.3.4 Classifier evaluation

With information gathered during utterance- and word level experiments, we evaluated which of different classifiers (see section 4.2) is the best for the later system implementation when using the full feature set (ASR+PROS+SYN) for word prediction. Each classifier was evaluated with the May release of Transtac data (see chapter 2), which is a very large set of samples for both training and testing. Table 4.3 presents the results for this experiment. Looking at this data we find that in addition to being vastly superior with regards to training time (several hours vs. less than an hour), MultiBoosted decision trees outperform SVM by quite a margin (69% increased performance). This may be caused both by the unbalanced nature of training data (7% of all training samples are of class *incorrect*) as well as by difficulties in the normalization of the discrete valued syntactic features with the continuous valued confidence and prosodic measures. According to this result, all further experiments were performed using MultiBoost J48 decision trees.

Classifier	Accuracy	Precision(c)	Recall(c)	Precision(ic)	Recall(ic)	F1(ic)	MC
DT	96.04 %	0.971	0.987	0.76	0.57	0.651	0.6381
SVM	95.41 %	0.957	0.996	0.86	0.35	0.497	0.5316
MultiBoost	94.78 %	0.966	0.977	0.757	0.679	0.716	0.6882

Table 4.3: Classifier performance for training and test split. (c) depicts measurements on the *correct* class while (ic) depicts measurements on the *incorrect* class. MC column presents the Matthews Correlation Coefficient as an auxiliary measure of performance.

4.3.5 1-stage and 2-stage error recognition

Taking into account results in utterance level, word level and classifier evaluation, we evaluated 1-stage and 2-stage approaches to misrecognized word prediction. In a 1-stage approach, we predicted misrecognition on all words in the test set in a single stage (i.e., is this word correctly recognized or not?). A word is misrecognized if it represents an insertion or a substitution. In the first stage of the 2-stage approach, we predict utterance misrecognition for each utterance in an ASR hypothesis. We considered an utterance to be misrecognized if the word error rate (WER) of the utterance was > 0. In the second stage, we predicted whether each word in the ASR hypothesis is misrecognized or not.

	Method	Misrec. words	*correct*			*incorrect*			Overall	Improvement
		in train./test set	**P**	**R**	**F**	**P**	**R**	**F**	accuracy	over Base.
1	Maj. Base	- / 8.5%	.91	1.0	.95	-	0.0	-	91.5 %	-
2	1-stage original	8.7% / 8.5%	.95	**.99**	**.97**	.77	.49	.60	94.4%	3.2%
3	1-stage upsampled	35% / 8.5%	**.96**	.97	**.97**	.64	**.60**	.62	93.7%	2.4%
4	2-stage original	8.7% / 8.5%	.95	**.99**	**.97**	**.85**	.43	.57	94.5%	3.3%
5	2-stage upsampled	35% / 8.5%	**.96**	.98	**.97**	.76	.52	**.63**	**94.5%**	**3.3%**

Table 4.4: Precision, Recall, F-measure, and overall accuracy for correctly recognized/misrecognized words, overall accuracy, and accuracy improvement compared to the baseline method. The highest values of each column are highlighted in **bold**.

We evaluated word-correctness prediction on the complete dataset using the 1-stage and 2-stage approaches. We split the dataset into 80% training and 20% test sets, maintaining a similar distribution for correct and incorrect utterances of 8.7%/8.5% in each. We trained the utterance classifiers using all utterances in the training set. We trained the misrecognized word classifiers using all words in the training set. In order to improve performance of the classifier, we experimented with upsampling instances of misrecognized words in the training set to 35%(this value was derived empirically) . Upsampling of an unbalanced dataset is a common procedure discussed in [Shriberg and Stolcke, 2002].

We evaluated each of the methods on the same test set, where 8.5% of words are misrecognized. Misrecognized utterance prediction in the 2-stage method uses a combination of ASR confidence and syntactic features (ASR+SYN) which was the highest performing feature combination reported in Table 4.1. Table 4.4 compares the majority baseline, 1-stage, and 2-stage methods for predicting misrecognized words in the test set. Line 1 shows the majority baseline prediction which achieves 91.5% overall accuracy by classifying all instances as 'correct'. Lines 2 and 3 show results for the 1-stage method trained on the original and upsampled datasets. Although the 1-stage method trained on the original dataset achieved higher overall accuracy (94.4%) than the 1-stage method trained on the upsampled dataset

(93.7%), the upsampled training set achieved higher recall and F-measure (.60/.62) for predicting misrecognized words compared to the original training set methods (.49/.60). Lines 4 and 5 show results for the 2-stage method trained on original and upsampled datasets. Both of the 2-stage methods achieved higher overall accuracies (94.5%) compared to the 1-stage methods. The 2-stage method trained on the original dataset achieved the highest precision for detecting misrecognized words of .85, while the 2-stage method trained on the upsampled dataset achieved the highest F-measure of .63.

All of the experimental methods improve overall accuracy performance by between 2.4% up to 3.3% compared to the majority baseline. The highest performance improvement is achieved by the 2-stage predicting methods. On the upsampled dataset, the 2-stage method achieved 52% recall and 76% precision in identifying misrecognized words. This means that an interactive system with clarification capabilities using the proposed error detection method would attempt to correct over half of misrecognized words with a clarification subdialogue. A quarter of clarification attempts in such a system would be made for a word that is actually correct. Unnecessary clarification may lead to a longer dialogue but would not necessarily deteriorate the system's recognition since an answer to a clarification for a correct word is likely to support the original hypothesis.

Chapter 5

Implementation

This chapter presents the software design of both the overall system and the error-detection module featuring classifiers built in chapter 4 as well as the implementation of the design for the error-detection module.

5.1 The system

5.1.1 Goals and limitations of the system

As addressed in 1.3, the BOLT system is designed to work as both a *Human-Machine Communication System* as well as a *Human-Human Dialog System.* The confidence scoring module is a part of a subsystem of which of the Human-Machine communication system. The goal of this system is to make sure, that the human input as understood by the machine is as close to the actual input as possible. For this purpose, the system starts with a confirmation dialog: The user is asked questions with regards to the input as understood by the machine and confirms whether this was the intended meaning or not. Given the greater context of an actual human to human dialog, this clarification dialog has to conform to standards ensuring maximum fluidity of the dialog as perceived by the interacting humans. This standard is reached by allowing for only a maximum of four turns before the input has to be accepted and post-processed. An example for such a maximum-length dialog can be seen in table 5.1.

User (Turn 1)	Hi, my name is Captain Pierce.
System	Could you please spell <audio-for-Pierce >?
User (Turn 2)	Papa,India, Echo, Romeo, Charlie, Echo.
System	You said P._I._E._R._C._E. Is that right?
User (Turn 3)	Yes, that is right.

Table 5.1: Example clarification dialog.

5.1.2 System overview

The proposed system was designed as a pipeline based on a central, multi-layered data structure (see 5.2). The different components of the system add and modify the content of this structure. This has to be done such that current data is available in time for any components down the pipeline. A diagram of the most recent version (as of August 2012) of the pipeline can be seen in figure5.1. The pipeline starts with a new speech input being recognized. During the recognition process, the ASR component saves both the final confusion network as well as the lattice generated to the data structure and also generates the 1-best transcription of the input. This information is used by the second component to re-score the lattice, i.e. to find a better 1-best solution. This step, however, was skipped in the final version of the system (August 2012) and just the original 1-best was used. The third component detects and marks (*out of vocabulary-OOV*, words which are not covered by the ASR vocabulary). During these computations the component also creates part-of-speech *(POS)* tags and writes the ASR confidence for each word given the information in the lattice to the data structure.

In the next step, the ASR runs in forced-alignment mode, using the 1-best transcription of the audio. This step is also used to compute prosodic information for each word. The fifth component is the word-confidence scorer based on the classifier built in chapter 4, adding word level confidences to the stored information. The following component *"Answer Extraction & Merging"* is only called if the ASR input is the answer to a previously issued reprise question. *"ASR Error Annotation"*, however, is called for every input and denotes regions in the recognition string that may contain an error. These regions are finally used by the *"Dialog Manager"* (DM) to determine whether or not (further) clarification questions are needed before the (combined) result is forwarded to the machine translation *(MT)*. Note, that everything from the Dialog Manager component onward is subject to DM internal logic and will thus not be explained in greater detail.

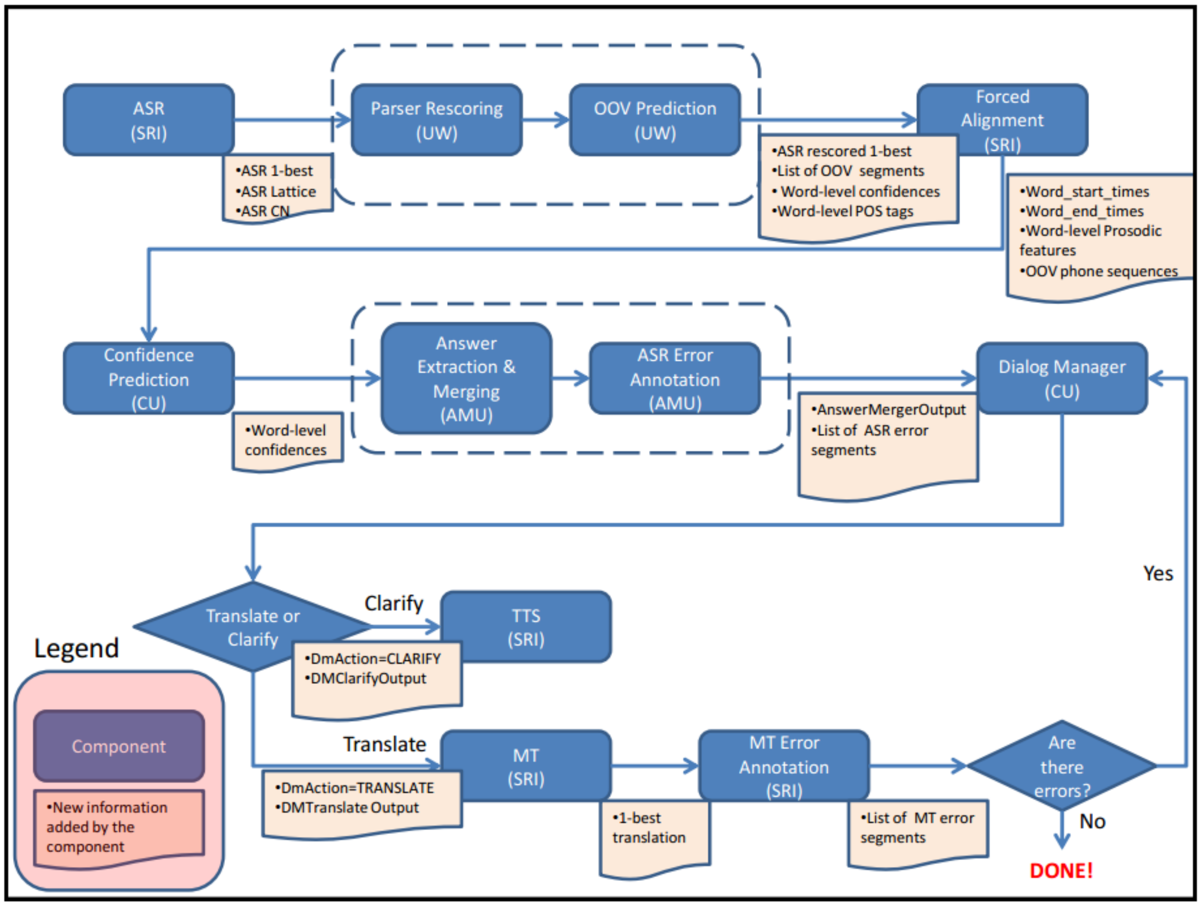


Figure 5.1: SRI's system pipeline. After the original ASR is rescored and labeled by two components labeled with (UW), the new hypothesis is force aligned to obtain new timestamps before the error prediction can start.

5.2 Google Protocol Buffers

Google Protocol Buffers presented a viable platform when choosing a central, multi-layered data structure serving as the foundation for the pipeline used in the system. Google Protocol Buffers are a language-neutral, platform-neutral, extensible mechanism for serializing structured data. Their structure is similar to that of XML based databases, however more specialized around ease of use in software projects of any size. Several circumstances led to the choice of implementing Google Protocol Buffers:

- The use of multiple programming and scripting languages throughout the different sites involved in the project made it necessary to find a container supported with support for all languages used and would also be easy to adapt. Google Protocol Buffers primarily support C++, Java, and Python but community built support packages for different languages are available.

- When compared to something like XML, Google Protocol Buffers are more compact and the objects themselves are directly populated as opposed to pulling from the XML fields to populate an object, saving both CPU time and memory as well as minimizing sources for errors.
- New fields can be easily introduced from one revision to the next without causing errors in modules not using those fields.

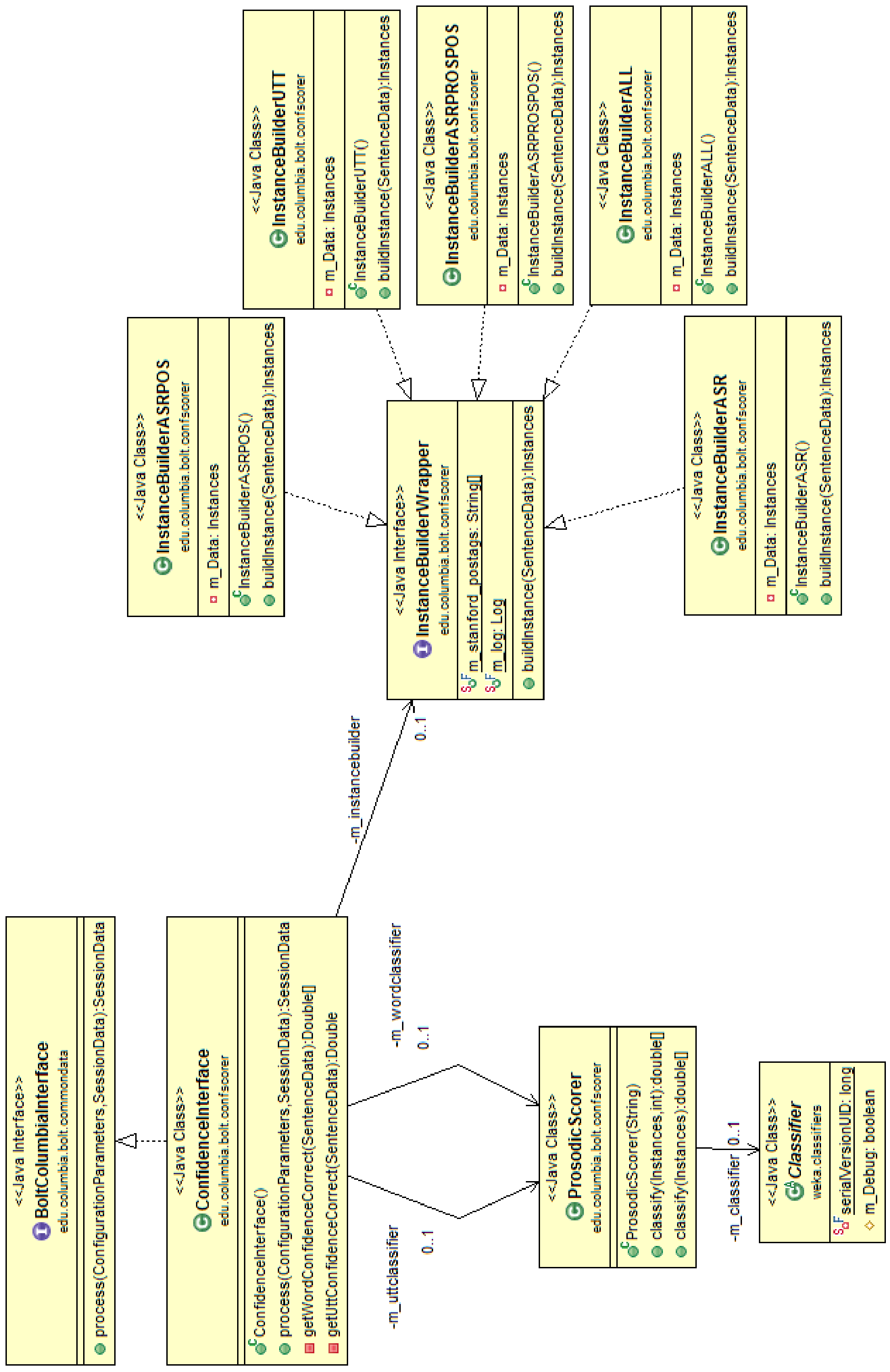
Creating multiple layers in the context of the BOLT system means that we categorize and save data at the following levels (A detailed design of the used buffer structure can be seen in appendix A.):

- Session level: one session represents one starting utterance plus up to three clarification turns. The entire history of these up to four turns is saved.
- Utterance level: represents the information gathered for a single utterance. Lattices, confusion networks as well as error segments are saved as well as the dialog manager action.
- Word level: represents data for every word in an utterance. Classification features like prosodic information is stored together with spelling information in case of an OOV word etc.

5.3 Code Setup

5.3.1 The Confidence Scorer

This code structure is built to first translate data retrieved via the internally used information structure (see section 5.3.2) to a format usable by the classifier built in chapter 4. Figure 5.2 shows a UML diagram presenting the major components.



<<Java Class>>
InstanceBuilderUTT
edu.columbia.bolt.confscorer
m_Data: Instances
InstanceBuilderUTT()
buildInstance(SentenceData):Instances
<<Java Class>>
InstanceBuilderASRPROSPOS
edu.columbia.bolt.confscorer
m_Data: Instances
InstanceBuilderASRPROSPOS()
buildInstance(SentenceData):Instances
<<Java Class>>
InstanceBuilderALL
edu.columbia.bolt.confscorer
m_Data: Instances
InstanceBuilderALL()
buildInstance(SentenceData):Instances
<<Java Class>>
InstanceBuilderASRPOS
edu.columbia.bolt.confscorer
m_Data: Instances
InstanceBuilderASRPOS()
buildInstance(SentenceData):Instances
<<Java Interface>>
InstanceBuilderWrapper
edu.columbia.bolt.confscorer
m_stanford_postags: String[]
m_log: Log
buildInstance(SentenceData):Instances
<<Java Class>>
InstanceBuilderASR
edu.columbia.bolt.confscorer
m_Data: Instances
InstanceBuilderASR()
buildInstance(SentenceData):Instances
-m_instancebuilder
0..1
<<Java Interface>>
BoltColumbiaInterface
edu.columbia.bolt.commondata
process(ConfigurationParameters,SessionData):SessionData
<<Java Class>>
ConfidenceInterface
edu.columbia.bolt.confscorer
ConfidenceInterface()
process(ConfigurationParameters,SessionData):SessionData
getWordConfidenceCorrect(SentenceData):Double[]
getUttConfidenceCorrect(SentenceData):Double
-m_wordclassifier
0..1
-m_uttclassifier
0..1
<<Java Class>>
ProsodicScorer
edu.columbia.bolt.confscorer
ProsodicScorer(String)
classify(Instances,int):double[]
classify(Instances):double[]
-m_classifier
0..1
<<Java Class>>
Classifier
weka.classifiers
serialVersionUID: long
m_Debug: boolean

Figure 5.2: UML diagram of the confidence scorer module

5.3.1.1 ConfidenceInterface

The class *ConfidenceInterface* is a singleton responsible for handling all in- and output with regards to confidence scoring as well as setting up and calling the actual scoring. The only publicly callable method is *process* which is calling the method*getWordConfidenceCorrect*, responsible for calling the actual scoring mechanism.

process process is being called by an external controller (the pipeline controller) and is responsible for both initializing the *ProsodicScorer* and the structure responsible for aligning incoming data in such a way that it is readable by the scorer, the *InstanceBuilder*. Both components are initialized according to information saved in a central configuration file (*ConfigurationParameters*). Data is transferred to the method as a copy of the content of the Google Protocol Buffer *(protobuffer)* SessionData structure.

The data is then converted to the Columbia-internally used information structure (see 5.3.2) and if this conversion was successful, the method for processing the data is called. After the classification is finished, the data structure is again converted back into protobuffer type information and saved to the protobuffer structure (see code snippet 5.1).

Code 5.1: Setting values in the data structure and writing back to the Protocol Buffer

```
.
.
data.setcfConfidence(Arrays.asList((getWordConfidenceCorrect(data))));
.
.
return wrapper.encode(sessiondata, data);
```

getWordConfidenceCorrect In this method, data in the Columbia structure is again converted to a structure readable by the WEKA classifier. After creating an array of sufficient size to hold confidence scores for every word in the currently processed utterance, the classifier is then called once for each word, returning confidence values for this word being correctly recognized. The returned information is then the array containing the confidence values (see code snippet 5.2).

Code 5.2: The getWordConfidenceCorrect method

```
private Double[] getWordConfidenceCorrect(SentenceData input) throws Exception{

                //build instance from InputData
                Instances data = m_instancebuilder.buildInstance(input);

                //Double uttconf = uttclassifier.classify(data)[0];
                Double[] wordconf = new Double[input.getWordsCurrentUtt().size()];

                //run classifiers and get confidence score for the word being 'correct'
                for (int i = 0; i < input.getWordsCurrentUtt().size(); i++) {
            wordconf[i] = m_wordclassifier.classify(data,i)[0];
```

```
        //index '0' refers to the confidence of the word/utterance being 'correct'.
        //index '1' would refer to the confidence being incorrect. both scores sum up to 1
                }

                //return confidences (words only in this case)
                return wordconf;

}
```

5.3.1.2 InstanceBuilder

The *InstanceBuilder* structure consists of both a central wrapper with which multiple different set-ups of data converters can be called. Which one of the converters is called depends on a central configuration file specifying which version of the classifier has to be called depending on the feature set available (see chapter 4).

The structure built is an *instances* file, which is usable by WEKA (see section 4.1) and contains a set of features defined in chapter 4. This is done by first defining the data entries (the header of the instances file) and then creating one instance per word, filling in values as available to the applicable fields as shown in code snippet 5.3.

Code 5.3: Snippets from the buildInstance method

```
public Instances buildInstance(SentenceData input){
        .
        .
        .
        //create all the attributes and add them to the vector
                Attribute total_dur = new Attribute ("total_dur"); //numeric\n";
                attributes.addElement(total_dur);

                Attribute f0mean = new Attribute ("F0MEAN");// numeric\n";
                attributes.addElement(f0mean);

                Attribute f0min = new Attribute ("F0MIN");//numeric\n";
                attributes.addElement(f0min);
        .
        m_Data = new Instances(nameOfDataset, attributes, 0);
        .
        for (int i = 0; i < input.getWordsCurrentUtt().size(); i++) {
        Instance inst = new Instance(22);
        .
        .
        inst.setValue(f0mean, input.getF0meanWords().get(i));
            inst.setValue(f0min, input.getF0minWords().get(i));
            inst.setValue(f0max, input.getF0maxWords().get(i));
        .
        .
        }

            return m_Data;
        }
```

5.3.1.3 ProsodicScorer

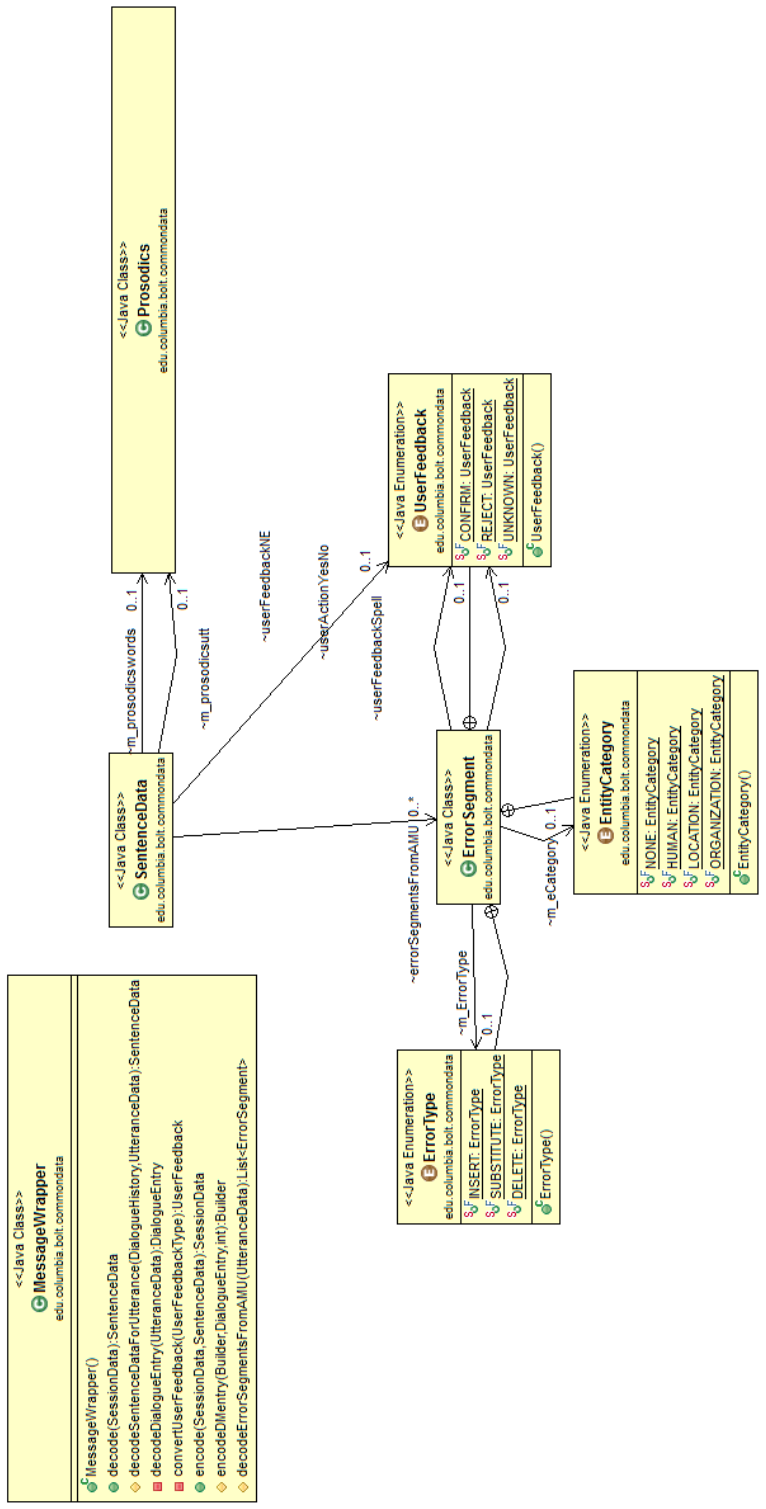
The ProsodicScorer, upon creation, loads a classifier according to a central configuration file specifying which features are available for classification. The only method available in this class is responsible for calling the classifier with the current *instances* dataset and the ID of which instance has to be classified. After the classification was successful, the confidence measure of the word being correctly classified by the ASR is returned to the caller (see code snippet 5.4).

Code 5.4: The classify method

```
public double[] classify(Instances data, int instanceid) throws Exception{

            data.setClassIndex(data.numAttributes()−1);

            //get confidence score
            return m_classifier.distributionForInstance(data.instance(instanceid));

      }
```

5.3.2 Information Structure

Due to the relatively late incorporation of Google Protocol Buffers in to the project, manual solutions were developed for the data transfer problem, especially for early software tests. An attempt to solve this problem was done at Columbia University in the form of a Java structure responsible for both holding as well as distributing information. A UML diagram showing the major components is shown in figure 5.3.



<<Java Class>>
Prosodics
edu.columbia.bolt.commondata
<<Java Enumeration>>
UserFeedback
edu.columbia.bolt.commondata
CONFIRM: UserFeedback
REJECT: UserFeedback
UNKNOWN: UserFeedback
UserFeedback()
<<Java Class>>
SentenceData
edu.columbia.bolt.commondata
~m_prosodicswords 0..1
~m_prosodicsutt 0..1
~userFeedbackNE
~userActionYesNo
~userFeedbackSpell
0..1
0..1
0..1
<<Java Class>>
ErrorSegment
edu.columbia.bolt.commondata
~errorSegmentsFromAMU 0..*
<<Java Enumeration>>
EntityCategory
edu.columbia.bolt.commondata
NONE: EntityCategory
HUMAN: EntityCategory
LOCATION: EntityCategory
ORGANIZATION: EntityCategory
EntityCategory()
~m_eCategory 0..1
~m_ErrorType 0..1
<<Java Enumeration>>
ErrorType
edu.columbia.bolt.commondata
INSERT: ErrorType
SUBSTITUTE: ErrorType
DELETE: ErrorType
ErrorType()
<<Java Class>>
MessageWrapper
edu.columbia.bolt.commondata
MessageWrapper()
decode(SessionData):SentenceData
decodeSentenceDataForUtterance(DialogueHistory,UtteranceData):SentenceData
decodeDialogueEntry(UtteranceData):DialogueEntry
convertUserFeedback(UserFeedbackType):UserFeedback
encode(SessionData,SentenceData):SessionData
encodeDMentry(Builder,DialogueEntry,int):Builder
decodeErrorSegmentsFromAMU(UtteranceData):List<ErrorSegment>

Figure 5.3: UML diagram of the common data structure

5.3.2.1 MessageWrapper

This class was originally responsible for the transfer of data between the two modules developed at Columbia (the Dialog Manager and the Confidence Scorer) and was later modified to work as the link between those modules and the Google Protocol Buffer structure.

The two most important methods of *MessageWrapper* are *decode* and *encode*. These methods are responsible for accurately moving information both from (decode) and also back to (encode) the protobuffer structure.

decode The heart of the decoding method is a for-loop which iterates through every word *(protoWord)* in the currently processed utterance *(protoUtt)* and extracts word level features saved for these words *WordLevelAnnotations*. The information is extracted by getter-methods which are automatically created in the protobuffer package for each field held within it. Finally, the extracted values are assigned to a corresponding List (see code snippet 5.5).

Code 5.5: The decode method

```
protected SentenceData decodeSentenceDataForUtterance(DialogueHistory history, UtteranceData protoUtt)
     throws SentenceDataException
{
.
.
List<Double> oovConf = new ArrayList<Double>();
               List<Double> asrConf = new ArrayList<Double>();
               List<Double> parseConf = new ArrayList<Double>();
               List<Double> neConf = new ArrayList<Double>();
        .
        .
 for (WordAnnotation protoWord: protoUtt.getWordLevelAnnotationsList()) {
                              wordcount = protoWord.getWordIndex();
                              words.add(wordasr[wordcount]);
                              starttime = protoWord.getStartOffsetSeconds();
                              endtime = protoWord.getEndOffsetSeconds();
                              duration.add((endtime − starttime)+1);
                       asrConf.add(protoWord.getAsrPosterior().getValue());
                       parseConf.add(protoWord.getParserConfidence().getValue());
        .
        .
        .
}
```

encode Inverse to the decode method, encode changes an entry in the protobuffer. This is done by first calling the information stored in the structure so there can be data saved to it. This is called invoking the *builder* of the structure subject to change. Due to the layered structure of the protobuffer used, this invoke call has to be done in hierarchical order down to the applicable layer, which in this case is the *WordAnnotation* layer, the computed *word confidence* is written to the field reserved for this information (see code snippet 5.6).

Code 5.6: The encode method

```
public SessionData encode(SessionData sessionData, SentenceData sentData){
            SessionData.Builder sessionBuilder = sessionData.toBuilder();
            //update the last utterance only
            UtteranceData.Builder utteranceBuilder = sessionBuilder.getUtterancesBuilder(
                        sessionBuilder.getUtterancesCount()−1);

            if (sentData.getcfConfidence() != null) {
            //set CU confidence values
            int wid = 0;
            for (WordAnnotation.Builder word: utteranceBuilder.getWordLevelAnnotationsBuilderList()){
                  word.setCuConfidence(word.getCuConfidence().toBuilder().setValue((sentData.
                      getcfConfidence().get(wid))));
                  wid++;
            }
            }

            //set DM output
            DialogueEntry dmEntry = sentData.getDmEntry();

        if(dmEntry!=null)
              utteranceBuilder = encodeDMentry(utteranceBuilder, dmEntry, sentData.
                  getM_addressErrorSegmentIndex());

        sessionBuilder.setUtterances(sessionBuilder.getUtterancesCount()−1, utteranceBuilder );
        return sessionBuilder.build();


}
```

5.3.2.2 SentenceData

SentenceData is an internally used information structure consisting of *Lists* holding information about the words contained in an utterance. Every word is represented by a certain index, which stays the same for all those list objects. The only information which is not saved internally in lists is prosodic information (held by a child-class called *Prosodics*). Also implemented in these structures are the getter and setter methods used to access the saved data (see code snippet 5.7).

Code 5.7: The SentenceData class

```
public class SentenceData {
      .
      .
//asr confidence for each word
List<Double> m_asrConfidence;
//parse confidence for each word
List<Double> m_parseConfidence;
      .
      .
//prosodic features for each words
Prosodics m_prosodicswords = new Prosodics();
      .
```

```
          .
public void setAsrConfidence(List<Double> asrConfidence) throws SentenceDataException {
            checkSize(asrConfidence);
            this.m_asrConfidence = asrConfidence;
}

public List<Double> getAsrConfidence() {
            return m_asrConfidence;
}
          .
          .
```

5.3.3 Results

By replacing the originally planned 2-stage approach with a 1-stage approach as described in section 4.3.5, the error detection module proved to be working within the constraints of the overall dialog system. Due to difficulties to implement the error score in a timely manner into the error annotation module (see figure 5.1) those scores were not used for evaluation tests done by DARPA. However, the methods proposed in chapter 4 were instead successfully integrated directly into the error annotation module, rendering a separate error prediction module obsolete.

Chapter 6

Conclusion

6.1 Summary

The aims of this thesis were to examine if the usage of prosodic and syntactic features could lead to building a classifier able to (1) identify utterances that have been misrecognized by an Automatic Speech Recognizer and (2) to identify which *portions* of utterances have been incorrectly transcribed by the recognizer. Furthermore, the developed classification methods were to be implemented into a Spoken Dialog System (developed as part of the 'Broad Operational Language Translation' *(BOLT)* program, funded by the 'Defense Advanced Research Projects Agency' *(DARPA)* of the 'United States Department of Defense') to allow for *targeted clarification questions* in case of misrecognitions. It was found that a classification system which in additon to standard ASR posteriors also makes use of prosodic and syntactic information, performed superior compared to systems using ASR posteriors only.

Feature Exploration A broad set of features, which was extracted using audio processing scripts, was tested for classification significance. Testing and evaluation of features yielded a stable set of features independent of the evaluation method. As expected, prosodic features helped to improve classification performance as well as POS tags, since both present natural extensions of acoustic models and language models used in today's ASR systems. Slightly surprising though was the complete lack of impact of fine-grained features like shimmer and jitter. A possible explanation may be found when taking into account the fact that data presented in experiments represented multiple speakers. While it may be possible to find shimmer and jitter values that increase misrecognition rate for a single speaker, a general pattern could not be found.

Modeling After testing three different machine learning algorithms (Decision Trees, Multi-Boost, Support Vector Machines) for performance, it was determined that *MultiBoost* Decision Trees yielded the best performance values. Taking this into account, 1-stage and 2-stage approaches to misrecognized word prediction were evaluated. In a 1-stage approach, we predicted misrecognition on all words in the test set in a single stage (i.e., is this word correctly recognized or not?). In the first stage of the 2-stage approach, misrecognitions were predicted for each utterance in the test set. In the second stage, it was predicted whether each word in the ASR hypothesis for these utterances was misrecognized or not. Each of the methods was evaluated on the same test set. The highest performance improvement over a simple majority-voter (e.g., classifying every word as being correctly recognized) was achieved by the 2-stage predicting method. An interactive system utilizing the 2-stage prediction approach would attempt to correct over half of misrecognized words with a clarification subdialogue thus decreasing the WER by 50% assuming perfect error correction.

Implementation To implement the proposed error detection into an SDS, the classifier had to be packed into a module fit to be part of the SDS design. To achieve this, the 2-stage approach was replaced by a 1-stage approach as there was no utterance information available. Also, a data structure was built to make incoming data readable by the classifier and also allow prediction results to be accessible to the SDS data structure. The final classification module proved to be working within the design constraints of the SDS.

6.2 Outlook

Although this thesis proved that error prediction on the word level of ASR output is possible, the scope was also rather limited. All experiments were performed using only data from a single, high-quality dataset and only one ASR system was used in the process. Possible directions for future work may include:

- **Adding datapoints.** By using different datasets as well as ASR systems, future research may test the robustness of the feature set developed in this thesis. Such research may also lead to additional features or eventual substraction of an entire feature group.

- **Speaker dependence.** Several features (e.g., shimmer, jitter) were found to have no or even harmful impact on error detection in this thesis. However, this may not prove true in a single speaker environment and thus lead to even higher performance when a system is able to adapt to a single user. Speaker dependent weighting of features may also provide additional performance.

- **Error analysis.** Due to time constraints in this thesis, little to no time was spent on analyzing which feature values (e.g., POS tag, F0 value) were especially good indicators for misrecognitions. Such analysis may provide information with regards to adaptation of acoustic models and language models used in ASR to achieve decreases in word error rate (WER).

- **OOV prediction.** Being able to predict and identify words which are not part of the ASR vocabulary is of great importance in speech processing. Error prediction as presented in this thesis does not distinguish between different sources of errors and thus may form a first level of OOV prediction.

6.3 Conclusion

All in all, it has been shown that error prediction on the word level of ASR output is possible and also capable of high performance with negligible impact on overall system performance/computation time. However, to be fully able to take advantage the predictions, the recognizer and/or the dialog system have to be able to perform sophisticated error correction. This may be done either by finding higher confidence hypotheses automatically (recognition) and/or by offering simple and understandable correction options to the user (dialog system). Dialog systems specifically have to be able to handle different kinds of user clarification attempts which may require natural language understanding algorithms not (yet) available.

Part I

Appendices

Appendix A

Google Protocol Buffers Structure

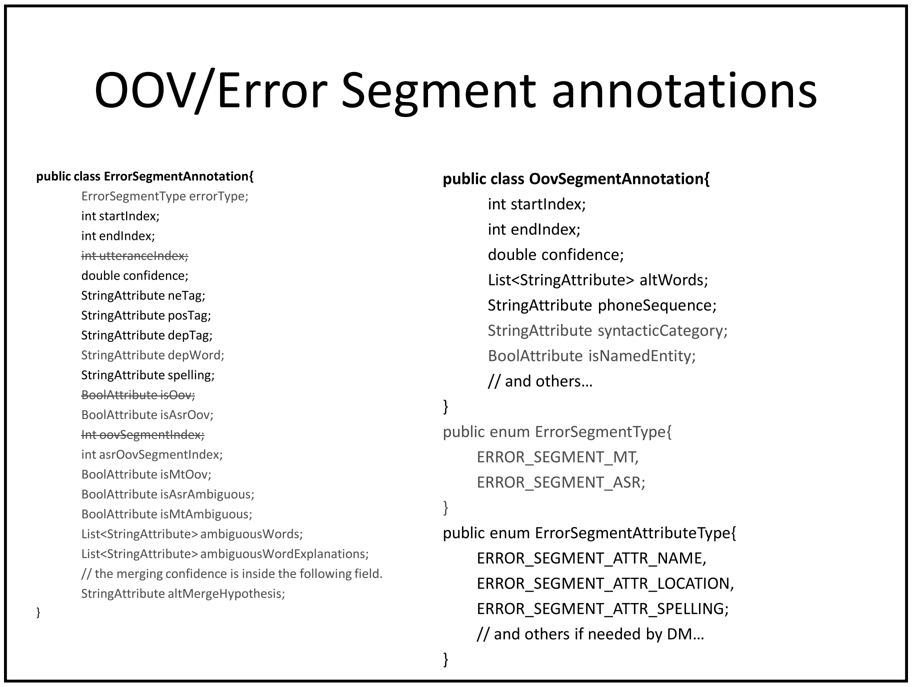
SessionData (Central Data Structure)

- Session represents one starting utterance plus up to 3 clarification turns.
- SessionData stores the entire history for these 4 turns.
- Each component has access to all the current utterance information generated up to that point PLUS all the information from previous utterances (e.g., AnswerMerging component has access to DM actions in the previous turns)

```
public class SessionData {
    int sessionId;
    // for counting number of turns (used by DM and answer merger)
    // 0 corresponds to first turn, 1 corresponds to second turn, etc. (max value = 3)
    int currentTurn;
    // keep all history information
    // utterences.get(0) stores the original utterance information
    List<UtteranceData> utterances;
}
```

Utterance information

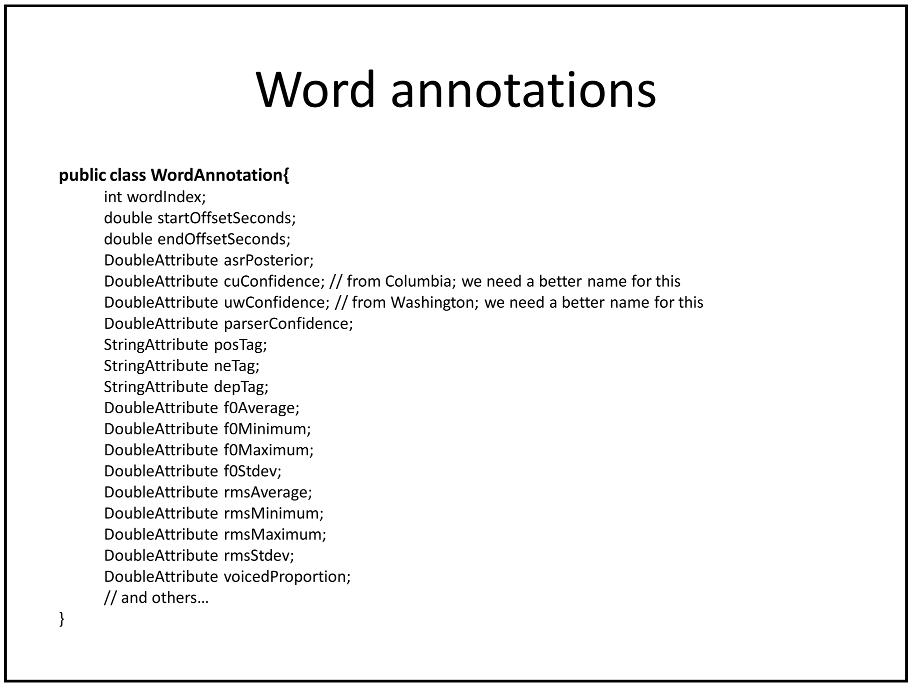
```
public class UtteranceData{
// 1-best ASR output
String recognizer1best;

// ASR lattice
String recognizerLattice;

// ASR word confusion network
String recognizerWcn;

// list of oov segments by UW
List<OovSegmentAnnotation> asrOovAnnotations;

// list of asr error segments by AMU
List<ErrorSegmentAnnotation> asrErrorSegments;

// each word is associated with multiple attributes
// word attributes are pos tags, confidence scores,  etc.
List<WordAnnotation> wordLevelAnnotations;

// rescored one best ASR output by UW
String rescored1best;

// the output produced by AnswerMerger
// moved the three fields in the old version to keep this class simple
AnswerMergerOutput mergedUtterance;

// type of action generated by DM (clarify or translate)
DmActionType dmAction;

// the output information generated by DM; only one must be filled
DmClarifyOutput dmClarifyOutput;
DmTranslateOutput dmTranslateOutput;

// list of mt error segments
List<ErrorSegmentAnnotation> mtErrorSegments;

// MT output
String mt1best;
}
```



OOV/Error Segment annotations

```
public class ErrorSegmentAnnotation{
        ErrorSegmentType errorType;
        int startIndex;
        int endIndex;
        ~~int utteranceIndex;~~
        double confidence;
        StringAttribute neTag;
        StringAttribute posTag;
        StringAttribute depTag;
        StringAttribute depWord;
        StringAttribute spelling;
        ~~BoolAttribute isOov;~~
        BoolAttribute isAsrOov;
        ~~Int oovSegmentIndex;~~
        int asrOovSegmentIndex;
        BoolAttribute isMtOov;
        BoolAttribute isAsrAmbiguous;
        BoolAttribute isMtAmbiguous;
        List<StringAttribute> ambiguousWords;
        List<StringAttribute> ambiguousWordExplanations;
        // the merging confidence is inside the following field.
        StringAttribute altMergeHypothesis;
}
```

```
public class OovSegmentAnnotation{
        int startIndex;
        int endIndex;
        double confidence;
        List<StringAttribute> altWords;
        StringAttribute phoneSequence;
        StringAttribute syntacticCategory;
        BoolAttribute isNamedEntity;
        // and others…
}
public enum ErrorSegmentType{
      ERROR_SEGMENT_MT,
      ERROR_SEGMENT_ASR;
}
public enum ErrorSegmentAttributeType{
      ERROR_SEGMENT_ATTR_NAME,
      ERROR_SEGMENT_ATTR_LOCATION,
      ERROR_SEGMENT_ATTR_SPELLING;
      // and others if needed by DM…
}
```



Word annotations

```
public class WordAnnotation{
        int wordIndex;
        double startOffsetSeconds;
        double endOffsetSeconds;
        DoubleAttribute asrPosterior;
        DoubleAttribute cuConfidence; // from Columbia; we need a better name for this
        DoubleAttribute uwConfidence; // from Washington; we need a better name for this
        DoubleAttribute parserConfidence;
        StringAttribute posTag;
        StringAttribute neTag;
        StringAttribute depTag;
        DoubleAttribute f0Average;
        DoubleAttribute f0Minimum;
        DoubleAttribute f0Maximum;
        DoubleAttribute f0Stdev;
        DoubleAttribute rmsAverage;
        DoubleAttribute rmsMinimum;
        DoubleAttribute rmsMaximum;
        DoubleAttribute rmsStdev;
        DoubleAttribute voicedProportion;
        // and others…
  }
```

Appendix B

Information Extraction Scripts

B.1 Recognition Tagging

```
#! /usr/bin/python

import os
import sys
import re


tf = open('C:/Users/phisa/Desktop/BOLT/Transdac/IA_live_eval_0508/logs.rec.log')
w = open('C:/Users/phisa/Desktop/BOLT/Transdac/confidences_utterance.txt', 'w')

data = tf.read()
start = 0
end = len(data)

i = data.count("SENTENCE:")

while(i > 0):
    sendex = data.index("SENTENCE:",start,end)
    fildex = data.index("FILENAME:",sendex,end)
    fnamdex = data.index(".wav",fildex,end)
    filename = data[fnamdex-26:fnamdex+4]
    refdex = data.index("REF:",fnamdex,end)
    hypdex = data.index("HYP:",refdex,end)
    ref = data[refdex+4:hypdex-2]
    correct = "correct"
    for letter in ref:
            if letter.isupper():
                correct = "incorrect"
                break
            if (letter == '*'):
                correct = "incorrect"
                break
    postdex = data.index("POSTERIORS:",hypdex,end)
    postend = data.index('\n',postdex,end)
    confidence = data[postdex+11:postend]
    regex = re.compile(r'([\d.]*\d+)')
    sum = 0
    j = 0
    for match in regex.finditer(confidence):
            sum = sum + float(match.group(1))
            j = j + 1
    confdence = sum/j
    finalconfidence = str(confdence)
    w.write(filename + ", " + correct + ", " + finalconfidence + "\n")
    i = i-1
    start = postend
```

B.2 Confidence Tagging

```
#! /usr/bin/python

import os
import sys
import re
import math

#Scans a textfile containing information taken out of .wcn files and
# returns average between this, previous, and consecutive values
def  getavg3valueLog(arr, i):
   total = arr[i];
   cnt = 1;
   if i>0: total= total+arr[i-1]; cnt = cnt+1
   if i<len(arr)-1: total= total+arr[i+1]; cnt = cnt+1
   return float(total)/float(cnt)

path = 'C:/Users/phisa/Desktop/BOLT/Transtac/20120703/'

ww = open(path + 'wcns_avg_log.txt', 'w')
tf = open(path + 'wcns.txt')

words = []
postag = []
oovconf = []
asrconf = []
parseconf = []
correct = []
oldfileid = ""
for line in tf:
    data = line.split(',')
    fileid = data[0]
    if(fileid == oldfileid):
        words.append(data[2])
        oovconf.append(math.log(float(data[6])))
        postag.append(data[5])
        parseconf.append(math.log(float(data[4])))
        asrconf.append(math.log(float(data[3])))
        correct.append(data[7].strip('\n'))
    else:
        for i in range(len(asrconf)):
            asrconfavg3 = str(getavg3valueLog(asrconf, i))
            parseconfavg3 = str(getavg3valueLog(parseconf, i))
            oovconfavg3 = str(getavg3valueLog(oovconf, i))
            if(i>0):
                postagprev = postag[i-1]
            else:
                postagprev = "Null"
            if(i<len(asrconf)-1):
                postagnext = postag[i+1]
            else:
                postagnext = "Null"
            ww.write(oldfileid + "," + str(i)  + ",\'" + words[i].strip('\'') +
"\'," +
                     str(asrconf[i]) + "," + asrconfavg3 + "," +
                     str(parseconf[i]) + "," + parseconfavg3 + "," +
                     postagprev + "," + postag[i] + "," + postagnext + "," +
                     str(oovconf[i]) + "," + oovconfavg3 + "," +
                     correct[i] + "\n")
        words = []
        postag = []
        oovconf = []
        asrconf = []
        parseconf = []
        correct = []
        words.append(data[2])
        oovconf.append(math.log(float(data[6])))
        postag.append(data[5])
        parseconf.append(math.log(float(data[4])))
        asrconf.append(math.log(float(data[3])))
        correct.append(data[7].strip('\n'))
        oldfileid = fileid

for i in range(len(asrconf)):
            asrconfavg3 = str(getavg3valueLog(asrconf, i))
            parseconfavg3 = str(getavg3valueLog(parseconf, i))
            if(i>0):
                postagprev = postag[i-1]
            else:
                postagprev = "Null"
            if(i<len(asrconf)-1):
                postagnext = postag[i+1]
            else:
                postagnext = "Null"
            ww.write(oldfileid + "," + str(i)  + ",\'" + words[i].strip('\'') + 
"\'," +
                     str(asrconf[i]) + "," + asrconfavg3 + "," +
                     str(parseconf[i]) + "," + parseconfavg3 + "," +
                     postagprev + "," + postag[i] + "," + postagnext + "," +
                     str(oovconf[i]) + "," + oovconfavg3 + "," +
                     correct[i] + "\n")
ww.close()
tf.close()
```

Part II

Bibliography

Bibliography

[Akbacak and others, 2009] M. Akbacak et al. Recent advances in SRI's IraqComm$^{\text{tm}}$ Iraqi Arabic-English speech-to-speech translation system. In *ICASSP*, pages 4809–4812, 2009.

[Bauer and Kohavi, 1999] E. Bauer and R. Kohavi. An empirical comparison of voting classification algorithms: Bagging, boosting,and variants. *Machine Learning*, 36:105–139, 1999.

[Boersma, 2001] Paul Boersma. Praat, a system for doing phonetics by computer. *Glot International*, 5(9/10):341–345, 2001.

[Bohus and Rudnicky, 2005] D. Bohus and A. I. Rudnicky. A principled approach for rejection threshold optimization in spoken dialog systems. In *INTERSPEECH*, pages 2781–2784, 2005.

[Bohus *et al.*, 2006] D. Bohus, B. Langner, A. Raux, A. Black, M. Eskenazi, and A. Rudnicky. Online supervised learning of non-understanding recovery policies. In *Proceedings of SLT*, 2006.

[Breiman, 1996] L. Breiman. Bagging predictors. *Machine Learning*, 24:123140, 1996.

[Cortes and Vapnik, 1995] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[Dzikovska and others, 2009] M. Dzikovska et al. Dealing with interpretation errors in tutorial dialogue. In *SIGDIAL Conference*, pages 38–45, 2009.

[Franco and others, 2002] H. Franco et al. Dynaspeak: Sri's scalable speech recognizer for embedded and mobile systems. In *Proceedings of the second international conference on Human Language Technology Research*, HLT '02, pages 25–30, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[Freund and Schapire, 1995] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55:119139, 1995.

[Goldwater and others, 2010] S. Goldwater et al. Which words are hard to recognize? prosodic, lexical, and disfluency factors that increase speech recognition error rates. *Speech Communication*, 52(3):181–200, 2010.

[Hirschberg *et al.*, 2004] J. Hirschberg, D. J. Litman, and Marc Swerts. Prosodic and other cues to speech recognition failures. *Speech Communication*, 43(1-2):155–175, 2004.

[Komatani and Okuno, 2010] Kazunori Komatani and Hiroshi G. Okuno. Online error detection of barge-in utterances by using individual users' utterance histories in spoken dialogue system. In *SIGDIAL Conference*, pages 289–296, 2010.

[Litman and Silliman, 2004] D. J. Litman and S. Silliman. Itspoke: an intelligent tutoring spoken dialogue system. In *Demonstration Papers at HLT-NAACL 2004*, HLT-NAACL–Demonstrations '04, pages 5–8, Stroudsburg, PA, USA, 2004. Association for Computational Linguistics.

[Lopes *et al.*, 2011] J. Lopes, M. Eskenazi, and I. Trancoso. Towards choosing better primes for spoken dialog systems. In *Proceedings of the IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*, 2011.

[Platt, 1998] John C. Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. Technical report, ADVANCES IN KERNEL METHODS - SUPPORT VECTOR LEARNING, 1998.

[Purver, 2004] M. Purver. *The Theory and Use of Clarification Requests in Dialogue.* PhD thesis, King's College, University of London, 2004.

[Quinlan, 1993] Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers, 1993.

[Shriberg and Stolcke, 2002] E. Shriberg and A. Stolcke. Prosody modeling for automatic speech recognition and understanding. In *Proceedings of the Workshop on Mathematical Foundations of Natural Language Modeling*, pages 105–114. Springer, 2002.

[Stoyanchev and Stent, 2009] S. Stoyanchev and A. Stent. Lexical and syntactic priming and their impact in deployed spoken dialog systems. In *Proceedings of the Meeting of the North American Chapter of the Association for Computational Linguistics (NAACL)*, 2009.

[Stoyanchev *et al.*, 2012] S. Stoyanchev, P. Salletmayr, J. Yang, and J. Hirschberg. Localized detection of speech recognition errors. 2012.

[Toutanova *et al.*, 2003] Kristina Toutanova, Dan Klein, Christopher D. Manning, and Yoram Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *IN PROCEEDINGS OF HLT-NAACL*, pages 252–259, 2003.

[WEBB, 2000] G. I. WEBB. Multiboosting: A technique for combining boosting and wagging. *Machine Learning*, 40:159–196, 2000.

[Weiss and others, 2008] B. A. Weiss et al. Performance evaluation of speech translation systems. In *LREC*, 2008.

[Witten and Eibe, 2005] I. Witten and F. Eibe. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, San Francisco, 2nd edition, 2005.