Spoken Language Processing Group
Columbia University in the City of New York

Signal Processing and Speech Communication
Laboratory
Graz University of Technology

Marshal Plan Scholarship Report

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

Philipp Salletmayr

Advisor: Univ.-Prof. Dipl.-Ing. Dr.techn. Kubin, Gernot

Advisor: Professor Hirschberg, Julia

Graz, December 10, 2012

LOCALIZED DETECTION OF SPEECH RECOGNITION ERRORS

Svetlana Stoyanchev[1], *Philipp Salletmayr*[2], *Jingbo Yang*[1], *and Julia Hirschberg*[1]

[1]Department of Computer Science, Columbia University, USA
[2]Signal Processing and Speech Communication Laboratory, Graz University of Technology, Austria
sstoyanchev@cs.columbia.edu, phisa@sbox.tugraz.at, jy2477@columbia.edu, julia@cs.columbia.edu

ABSTRACT

We address the problem of *localized error detection* in Automatic Speech Recognition (ASR) output. Localized error detection seeks to identify which particular words in a user's utterance have been misrecognized. Identifying misrecognized words permits one to create *targeted* clarification strategies for spoken dialogue systems, allowing the system to ask clarification questions targeting the particular type of misrecognition, in contrast to the *"please repeat/rephrase"* strategies used in most current dialogue systems. We present results of machine learning experiments using ASR confidence scores together with prosodic and syntactic features to predict whether 1) an utterance contains an error, and 2) whether a word in a misrecognized utterance is misrecognized. We show that by adding syntactic features to the ASR features when predicting misrecognized utterances the F-measure improves by 8% compared to using ASR features alone. By adding syntactic and prosodic features when predicting misrecognized words F-measure improves by 40%.

1. INTRODUCTION

The ability to clarify information is important for success in both human/human and human/machine communication. When human speakers believe they have misunderstood their interlocutors, they ask clarification questions that typically take advantage of context to formulate their queries.

Examining human clarification strategies, Purver[1] distinguishes two types of clarification question: *reprise* and *non-reprise*. While reprise clarification questions ask 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, non-reprise questions are a generic request for repetion, which does not contain contextual information from the misunderstood utterance. The two are illustrated below:

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

About 88% of human clarification questions are reprise questions, compared to 12% non-reprise.

In human/machine communication, automatic Spoken Dialogue Systems (SDS) also employ clarification questions to recover from 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, adopting simple statements of misunderstanding followed by requests to repeat or rephrase user input that can be applied to any type of hypothesized ASR error. Non-reprise clarification questions are easy to construct a priori and are well-suited to simple slot-filling dialogue systems where speakers are required to specify values for a fixed number of predefined attributes and concepts. However, previous research has found that system prompts have an important effect on a user's perception of the system's behaviour and performance [2, 3]. As we move towards systems that support mixed or even user initiative, such as tutoring systems [4] and speech to speech translation systems [5], SDS which can request more specific information about hypothesized ASR errors become more critical to create.

One requirement for producing reprise clarification questions is the detection of just which part of a user utterance has been recognized correctly and which part or parts contain an error. Previous research on error detection in ASR in general and in SDS applications in particular has focused primarily on determining how likely an utterance is to have been recognized correctly or incorrectly using posterior probabilities from the ASR acoustic and language models. Such information may be used to choose another path through the ASR lattice or to request repetition or rephrasing of the utterance from the user.

In the work presented here, we seek to identify not only which utterances have been misrecognized but also which *portions* of utterances have been incorrectly transcribed by the recognizer, in order to use this information to formulate targeted reprise questions in a Speech-to-Speech (S2S) translation system. In such systems, two speakers communicate orally in two different languages through two ASR systems and two Machine Translation (MT) systems. An S2S system takes speech input, recognizes it automatically, translates the recognized input into text in another languages, and produces synthesized speech output from the translation for the conversational partner. In the S2S application we target, speakers may 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 intercept speech recognition errors early in the dialogue to avoid translating poorly recognized utterances.

In this paper we describe a two-stage approach to ASR error localization by first predicting whether an ASR hypothesis is misrecognized and then identifying which words of the errorful utterance have been misrecognized. We explore a combination of ASR posterior probabilities, prosodic, and syntactic features in a machine learning classification task. While prosodic information has previously been used to identify ASR error at the utterance level, its use in localizing word error has been much less studied and rarely used for tasks other than simple reduction of ASR error. Our research represents both new results in detecting local errors and a new application for this task.

In Section 2 we discuss previous research on error prediction and question generation for dialogue systems. In Section 3 we describe the corpus we use in our experiments. In Section 4, we describe our approach and in Section 5 we present results. We conclude in Section 6 and outline future research directions.

2. RELATED WORK

Handling of errors in automatic spoken dialogue systems involves detecting the occurrence of an error and determining an appropriate dialogue strategy to correct it. To improve error detection, Bohus and Rudiniki [6] analyse tradeoffs between misunderstandings and false rejections in a dialogue system and optimize rejection thresholds using data-driven methods. Hirschberg et al. [7] find that prosodic features are very useful in identifying misrecognized utterances. Lopes et al. [8] analyse different feature sets for improving confidence score estimation for a user utterance in a dialogue system. Komatani and Okuno [9] use utterance history to determine whether a barge-in user utterance has been correctly recognized.

For determining dialogue strategies for error recovery, Dzikovska et al. [10] describe an approach to dealing with errors in tutoring dialogue systems. Bohus et al. [11] use supervised learning to determine the optimal error recovery policy in a dialogue system, such as providing a help message, repeating a previous prompt, or moving on to the next prompt.

Our work on localized error detection is a step towards introducing a new policy type in a dialogue system: *asking a targeted clarification question*. Our goal is to detect misrecognized words in a user utterance. Ogawa and Nakamura [12] address a similar question of word-level speech recognition confidence optimization by joint modeling of error confidence and potential error causes, such as noise, speakers' gender, and use of an out-of-vocabulary word. In our work, we model ASR confidence using a combination of the recognizer's confidence score, prosodic, and syntactic features.

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 1: Example dialogue from the IraqComm Corpus.

Our use of prosodic features is motivated by [13, 7, 14]. Shriberg et al. [13] summarize successful use of prosodic features for a variety of tasks including disfluency detection, overlap modeling, and sentence segmentation. Results show that prosodic information can significantly improve accuracy on classification and tagging tasks. Hirschberg et al. [7] find that prosodic features are useful in identifying misrecognized utterances. Goldwater et al. [14] discover that there are more recognition errors for words with extreme prosodic values than words with typical values.

3. DATA

	Overall	**Correct ASR**	**Error in ASR**
All Utt	3,729	2,664 (71.4%)	1,065 (28.6%)
All Words	24,857	22,697 (91.3%)	2,160 (8.7%)
Words in err Utt	7.48	5.45 (72.8%)	2.03 (27.2%)

Table 2: Data set from *IraqComm* system.

We perform our experiments on data from SRI's *IraqComm* speech-to-speech translation system [5]. The data was collected by NIST during seven months of evaluation exercises performed between 2005 and 2008 [15]. The corpus contains simulated dialogues between English military personnel and Arabic interviewees. Table 1 shows a sample dialogue from the dataset, with correct English translations for the Arabic utterances.

The corpus includes English and Arabic speech with manual transcriptions. We use the audio and manually annotated transcripts of English utterances for our experiments. We ran the SRI DynaSpeak [16] speech recognizer on the English utterances, generating posterior probabilities and word-audio alignments using acoustic and language models generated from the NIST training dataset by SRI. We then removed utterances in which a user directed a command to the computer, such as *Computer, repeat*. We also removed instances where a difference in ASR and transcript are due to annotation, such as contractions *we're* and *we are* and utterances containing disfluencies.

The resulting corpus contains a total of 3.7K utterances and 26K words. 28.6% of utterances and 8.7% of words contain an ASR error. A misrecognized utterance (an utterance with at least one recognition error) contains an average of 7.48 words in length and includes average 2.03 misrecognized words (Table 2). This corpus is well-suited for the task of localized error detection, since, in utterances containing an ASR error, the majority of words are recognized correctly.

4. METHOD

Localized detection of speech recognition errors is achieved by predicting which words in an ASR hypothesis are misrecognized. We evaluate 1-stage and 2-stage approaches to misrecognized word prediction. In a 1-stage approach, we predict misrecognition on all words in a 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 a 2-stage approach, we predict utterance misrecognition for each utterance in an ASR hypothesis. We consider an utterance to be misrecognized if the word error rate (WER) of the utterance is > 0. In the second stage, we predict whether each word in the ASR hypothesis is misrecognized or not.

To identify misrecognitions at the utterance and the word level, we use ASR confidence scores, as well as prosodic and syntactic features, as summarized in Table 3.

Our ASR confidence scores are the posterior probabilities generated by the DynaSpeak recognizer. For the misrecognized utterance prediction experiments, we look at the average posterior probabilities of all words in the ASR hypothesis. For the misrecognized word experiments, we use the posterior probability of the current (target), previous, and next words in the ASR hypothesis, and the average of these posterior probabilities. We obtain syntactic features by automatically assigning part-of-speech tags using the Stanford POS tagger [17] on the ASR hypotheses. For the misrecognized utterance prediction experiment, we use unigram and bi-gram counts of POS tags. To avoid data sparsity, we only count unigrams and bigams that appear more than 10 times in the corpus. For misrecognized word prediction experiments, we use the POS tag and broad class tag type (content vs. function word) of the current, previous, and next words. We assign function tags to prepositions, pronouns, determiners, conjunctions, modals, and adverbs and content to all of the other words. We extract prosodic features from the entire utterance for the misrecognized utterance experiments, using word boundary information generated by DynaSpeak for the misrecognized word experiment. We report results with J48 decision tree machine learning classifier boosted with MultiBoostAB method [18] using the Weka [19] machine learning library. Boosted J48 performed best on our data set compared with other machine learning algorithms we experimented with, including Support Vector Machines, Ripper, and regular J48 decision trees.

5. RESULTS

5.1. Comparison of Feature Sets

First, 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. We present results of misrecognized utterance and word prediction experiments on ASR confidence, prosodic, and syntactic features, compared to the majority class baseline. We evaluate the effect of combining prosodic and syntactic features with the ASR confidence features. In these experiments, we perform 10-fold cross-validation on the full dataset. Table 4 shows precision, recall, and F-measure[1], 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 .93/.68. We observe that across all measures a combination of syntactic and ASR features achieves the highest performance. In order to create targeted clarifications, we are particularly interested in increasing the F-measure for detection of misrecognized utterances. We observe that by adding syntactic features to ASR features, the F-measure of detecting misrecognized utterances increases by 13.3%.

Our ultimate goal is to use the output of utterance misrecognition prediction as an input to word misrecognition prediction. We run this experiment on a subset of the data with the words from misrecognized utterances utterances known from the reference transcription 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 5 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 predic-

[1]F-measure = $2 * recall * precision/(recall + precision)$

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
	energy(MAX/MIN/MEAN/STDEV)	for whole utterance	for word
	proportion of voiced segments	in whole utterance	in current word
	duration	of 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: Features used in the experiments

Feature	Misrec	Utt Correctly Rec.			Utt Misrec			F Misrec	Overall
	utt in corpus	**P**	**R**	**F**	**P**	**R**	**F**	compared to ASR	Accuracy
Maj. Base.	28.6%	.71	1	.83	-	0	0	-100%	71.4%
ASR	28.6%	.83	.90	.86	.68	.53	.60	0	79.4%
ASR+RROS	28.6%	.82	.89	.85	.65	.51	.57	-.05	78.1%
ASR+SYN	28.6%	**.86**	**.93**	**.89**	**.77**	**.61**	**.68**	**+13.3**	**83.8%**

Table 4: 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**.

tion. The majority class baseline (predict correct recognition) achieves 72.8% overall accuracy, again failing to detect any of the incorrectly recognized words. Using the ASR confidence features alone, we achieve 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 misrecognized words compared to the classifier that uses only ASR features.

These experiments show that the best performing feature combination for predicting misrecognized utterances is ASR+SYN and for words ASR+PROS+SYN. In the next set of experiments we use these feature sets to construct classifiers in 1-stage and 2-stage misrecognition prediction methods.

5.2. Estimating System Improvement

We evaluate word-correctness prediction on the complete dataset using 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 train the utterance classifiers using all utterances in the training set. We train the misrecognized word classifiers using all words in the training set. We experiment with upsampling instances of misrecognized words in the training set to 35%[2] in order to improve performance of the classifier. Upsampling of an unbalanced dataset is a common procedure discussed in [13].

We evaluate each of the methods on the same test set where 8.5% of words re 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. Table 6 compares the majority baseline, 1-stage , and 2-stage method for predicting misrecognized words in a 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 a 1-stage method trained on the original and upsampled datasets. We observe that, although the 1-stage method trained on the original dataset achieves higher overall accuracy (94.4%) than the 1-stage method trained on the upsampled dataset (93.7%), the upsampled training set achieves higher recall and F-measure (.60/.62) for predicting misrecognized words compared to original training set methods (.49/.60). Lines 4 and 5 show results for a 2-stage method trained on original and upsampled

[2]We derived this value empirically.

Feature	Misrec words in corpus	Word Correctly Rec. **P**	**R**	**F**	Word Misrec **P**	**R**	**F**	F Misrec compared to ASR	Overall Accuracy
Maj. Base	27.2%	.73	1	.84	-	0	0	-100%	72.8%
ASR	27.2%	.81	**.93**	.86	.69	.40	.50	0%	78.7%
ASR+PROS	27.2%	.82	.92	.86	.67	.46	.54	8%	79.0%
ASR+PROS+SYN	27.2%	**.87**	**.93**	**.90**	**.76**	**.64**	**.70**	**40%**	**84.7%**

Table 5: 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**.

	Method	Misrec. words in train. set	Misrec. words in test set	Word Correctly Rec. **P**	**R**	**F**	Word Misrec. **P**	**R**	**F**	Overall accuracy	Acc Improve over maj. 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 6: 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 in each column are highlighted in **bold**.

datasets. Both of the 2-stage methods achieve higher overall accuracy (94.5%) compared to the 1-stage methods. The 2-stage method trained on the original dataset achieves the highest precision for detecting misrecognized words of .85, while the 2-stage method trained on the upsampled dataset achieves the highest F-measure of .63.

All of the experimental methods improve overall accuracy performance by 2.4%-3.3% compared to the majority baseline. The highest performance improvement is achieved by the 2-stage predicting methods.

Note that, in comparison, Ogawa and Nakamura [12] achieved 87%[3] accuracy using joint modeling of confidence and potential error causes on a corpus of Japaneese spoken words with simulated noise conditions. These results are not directly comparable to our results as they were achieved on a different language, different domain, using a different speech recognizer with a higher word error rate of 30.69%.

6. CONCLUSIONS AND FUTURE WORK

We have presented results of experiments designed to localize ASR errors in spoken dialogue systems in order to construct error-targeted reprise clarification questions automatically. By responding to hypothesized errors more specifically we should be able to obtain clarifications from users in a more natural and effective way. We employ ASR confidence scores, prosodic, and syntactic features in 1-stage and 2-stage approaches to error localization. In our 1-stage approach we identify misrecognized words in all utterances while in our 2-stage approach we first identify misrecognized utterances and then identifying the words within those utterances that have been misrecognized. On a corpus of English utterances collected from the SRI IraqComm Transtac speech-to-speech translation system, we have found that a combination of ASR confidence scores and syntactic features can detect utterance recognition errors with an overall accuracy of 94.5% which is 3.3% improvement over a majority class baseline of 91.5%.

Our experimental results show that prosodic and syntactic features improve performance of misrecognized utterance and word classifiers. A combination of ASR and syntactic features achieves the highest F-measure on misrecognized utterance prediction with 13.3% improvement over the ASR features alone. A combination of ASR, prosodic, and syntactic features achieves the highest F-measure on misrecognized word prediction with 27.1% improvement over the ASR features alone.

Comparing the performance of 1-stage and 2-stage word prediction methods, we observe that each method achieves highest performance according to a different measure of misrecognized word prediction: the 1-stage method trained on the upsampled set optimizes recall and F-measure, the 2-stage method trained on the original set optimizes precision, and the 2-stage method trained on the upsampled set optimizes F-measure. When misrecognized word prediction is used in a system, the difference between the classification methods and training set class distributions may be taken into consideration based on the system goals.

In future work, we will examine the usefulness of additional syntactic, semantic, and prosodic features, such as higher level prosodic features using Rosenberg's AuToBI [20]. Finally, we will incorporate our results into our clarification question generator. We are currently collecting a corpus of human clarification questions in response to utterances with ASR errors using Amazon Mechanical Turk, to

[3] 1 - EER reported in Table 5 of [12] for Confidence y_0 using all features.

build our model of error-targeted clarification questions and evaluate it in our speech-to-speech translation application.

7. ACKNOWLEDGEMENTS

This work was partially funded by DARPA HR0011-12-C-0016 as a Columbia University subcontract to SRI International.

8. REFERENCES

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

[2] 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.

[3] 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.

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

[5] M. Akbacak et al., "Recent advances in SRI's IraqComm^{tm} Iraqi Arabic-English speech-to-speech translation system," in *ICASSP*, 2009, pp. 4809–4812.

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

[7] J. Hirschberg, D. J. Litman, and Marc Swerts, "Prosodic and other cues to speech recognition failures," *Speech Communication*, vol. 43, no. 1-2, pp. 155–175, 2004.

[8] J. Lopes, M. Eskenazi, and I. Trancoso, "Incorporating asr information in spoken dialog system confidence score," in *Computational Processing of the Portuguese Language*, Lecture Notes in Computer Science.

[9] 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*, 2010, pp. 289–296.

[10] M. Dzikovska et al., "Dealing with interpretation errors in tutorial dialogue," in *SIGDIAL Conference*, 2009, pp. 38–45.

[11] 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.

[12] A. Ogawa and A. Nakamura, "Joint estimation of confidence and error causes in speech recognition," *Speech Communication*, vol. 54, no. 9, pp. 1014 – 1028, 2012.

[13] 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*. 2002, pp. 105–114, Springer.

[14] S. Goldwater et al., "Which words are hard to recognize? prosodic, lexical, and disfluency factors that increase speech recognition error rates," *Speech Communication*, vol. 52, no. 3, pp. 181–200, 2010.

[15] B. A. Weiss et al., "Performance evaluation of speech translation systems," in *LREC*, 2008.

[16] 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*, San Francisco, CA, USA, 2002, HLT '02, pp. 25–30, Morgan Kaufmann Publishers Inc.

[17] K. Toutanova et al., "Feature-rich part-of-speech tagging with a cyclic dependency network," in *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*, 2003.

[18] G. I. Webb, "Multiboosting: A technique for combining boosting and wagging," in *MACHINE LEARNING*, 2000, pp. 159–196.

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

[20] A. Rosenberg, "Autobi - a tool for automatic tobi annotation," in *INTERSPEECH*, 2010, pp. 146–149.

Acoustic Model Building and Forced Alignment for the Let's Go! Data Set on Sphinx4

Philipp Salletmayr

November 8, 2012

Abstract

This Seminary Project report is concerned with the preparation of the Let's Go! [6] data set as a basis for to researching the impact of prosodic features to ASR error prediction. This preparation requires the execution of *forced alignment* on the available data in order to provide word-level start and and end time makers. In the course of this report, a brief analysis of the data available is made. After simple overviews about language and acoustic models are given, a more in-depth explanation how those models were created is provided.

Contents

List of Tables

Chapter 1

Introduction & Goals

To conduct sophisticated experiments in Automatic Speech Recognition (ASR), the data to be used has to fulfill certain criteria. The most basic of these is the availability of transcriptions describing the content of the audiodata as accurately as possible. One characteristic of these transcription may also be the marking of start- and end-times of words. As most transcriptions are done manually those timestamps are rarely part of this process. Even if the marking of timestamps is part of the transcription process, the accuracy is usually lacking due to rough scaling (seconds vs milli-seconds). Also the extent of the transcribed data is usually rather small due to dramatically increased costs in both time and money. One method to automatically derive timestamps is *forced alignment*. Forced Alignment is a process where a predefined text is being fitted onto an audio file such that each consecutive word is put to the most likely timespan of the audio according to ASR. Thus, in addition to the audio and transcriptions, a working ASR system capable of processing the audio data is needed.

In the context of preparing experiments concerned with word-level feature extraction, the data chosen to serve as the basis for these experiments was the Let's Go! data set. While both audio and transcriptions were available, no word-level marking of start- and end-times was done during the transcription process. As these markings were necessary for experiments, the decision was made to obtain these markers by running forced alignment on the data set using transcriptions. The ASR system chosen to serve as a basis for the experiments - CMU Sphinx4 - however does not support either forced alignment or the audio format used for the recording of audio in the Let's Go! data set. So in order to conduct experiments, 2 prerequisites had to be achieved and were defines as the goals of this Seminary Project:

1. Train models so that the ASR would be able to process the Let's Go! dataset
2. Prepare forced alignment using Sphinx4 ASR and compute word-level timestamps

Chapter 2

Data

The Let's Go! data set is the result of a project done by the Carnegie Mellon University (CMU) in Pittsburgh, PA. Its goal was to build a spoken dialog system which enhances the usability for non-native speakers and the elderly. In order to gather and evaluate data, the system was set up as a bus information system, providing users with the ability to inquire about current schedules of the public bus system in the greater Pittsburgh area. [6] Table 2.1 shows an example of a typical dialog. The data used in the training and testing of our ASR was collected from September 2008 to August 2009 containing 257654 utterances corresponding to roughly 150 hours of speech.

SYSTEM:	WELCOME TO CMU LETS GO BUS INFORMATION
SYSTEM:	WHAT CAN I DO FORYOU?
User:	Id like to go to Squirrel Hill.
SYSTEM:	GOING TO SQUIRREL HILL. IS THIS CORRECT?
User:	Yeah.
SYSTEM:	WHERE ARE YOU LEAVING FROM?
User:	Im leaving from CMU.
SYSTEM:	LEAVING CMU. AT WHAT TIME DO YOU WANT TO LEAVE?
User:	Now.
SYSTEM:	LET ME LOOK THAT UP FOR YOU. THE NEXT 61C LEAVES FORBES AND CRAIG AT []

Table 2.1: Example dialogue from the Let's Go! Corpus. [2]

The audiofiles were provided as 8kHz raw audio format (RAW) with little endian coding. Transcriptions were provided in a single .csv file. Every transcription included:

- id: the ids for that particular utterance. It also corresponds to the unique path to the .raw file. E.g., ./20081001/001/004.raw (the turn 004 of dialog 000 on 2008/10/01)
- label: this is either "nonunderstandable" (part or all of the utterance is not understandable), "understandable_correct" (the utterance is understandable and was correctly recognized by the ASR) or "understandable_incorrect" (the utterance is understandable and was not correctly recognized by our ASR)
- asr_output : output given by our ASR.
- first_confidence : confidence on the first pass, as explained below
- crowd_transcript : human transcript of that utterance. explained below.
- second_confidence : confidence on the crowd_transcript

The data set was transcribed through crowdsourcing. The transcription consisted of two passes. The first pass was to try to weed out the real bad recordings, where part of it was non-understandable. For that first passed, 5 persons were asked to pick one of the 3 labels. So the first_confidence measure is the number of workers who voted for that label. For example, a first_confidence of 0.8 on a ”nonunderstandable” means that 4 persons out of 5 agreed on this label. At that point, the goal was to put the ”understandable_incorrect” in the second_pass where workers would actually transcribe the utterances. However, since this is the most expensive step in the process, it was decided to put up only the ”understandable_incorrect” with a first_confidence of 0.8 and higher, so that there wouldn't be any ”hard to transcribe” utterances in the second pass. [7]

Every crowd_transcript would thus fall in one of these 4 categories :

1. ASR worked well : crowd_transcript is the output of ASR, second_confidence is the percent of workers who think the ASR output is good
2. The utterances is all/partly not understandable: crowd_transcript is non_understandable and the second_confidence is the percent of workers who think the utterance is not understandable
3. The crowd transcribed the utterance : crowd_transcript is what workers agreed on what the transcript should be. Second confidence is either 0.8 if 4 workers gave that transcript, or 1.0 if all 5 workers gave that transcript.
4. The transcript proposed is not reliable : crowd_transcript is going to be the best guess on how the utterance should be transcribed. The second_confidence is going to be 0.

Table 2.2 shows an example of how an utterance was transcribed and labeled.

id :	./20081001/000/003.raw
label :	understandable_incorrect
asr output:	forbes if than the re going
first_confidence :	0.800000
crowd_transcript :	forbes and bigelow
second_confidence :	0.644444444444
Workers involved:	5

Table 2.2: Example data entry from the Let's Go! Corpus.

54% of utterances in the corpus were labeled as ”understandable_correct”, 29% as ”understandable_incorrect” 29% and 17% as ”nonunderstandable”. In order to achieve best possible performance, only transcriptions labeled either ”understandable_correct” or ”understandable_incorrect” with a ”first_confidence” equal or higher than 0.8 were used for training and testing of the ASR engine. Which left a total of 212828 utterances. The audio files were also converted to 8kHz .wav files to simplify processing. For training and testing we split the data up into smaller junks:

- Data from September through December 2008 was used to train both the language and acoustic models.
- Data from January 2009 was used for testing purposes.
- Other data was left out of the train/test circle and only used to further evaluate performance.

Chapter 3

Language Model

A language model (LM) is used to restrict word search. It defines which word could follow previously recognized words (as the matching is a sequential process) and helps to significantly restrict the matching process by stripping words that are not probable. Search is constrained either absolutely by enumerating some small subset of possible expansions or probabilistically by computing a likelihood for each possible successor word. The former usually relies on an associated grammar which is compiled down into a graph, the latter is trained from a corpus. To reach a good accuracy rate, a language model must be very successful in search space restriction. This means it should be good at predicting the next word. A language model usually restricts the vocabulary considered to the words it contains. A language model can also contain smaller chunks like subwords or even phones. Search space restriction in this case is usually worse and corresponding recognition accuracies are lower than with a word-based language model though. Such models are usually called upon for name recognition. [4]

Sphinx4 requires a statistical language model which can be trained using the *CMU-Cambridge Language Modeling Toolkit (CMUCLMT)*. For training of the LM, only the transcriptions of the corpus are needed, but have to be prepared to follow a standard input model defined by the toolkit:

- Utterances may not contain any numericals (e.g. 64). Thus, all numericals have to be converted into a string. This was done using a python script. A.1
- All words in an utterance have to be capitalized
- Utterances have to be delimited by >s <and >/s <tags. The result should be the set of sentences that are bounded by the start and end sentence markers:

>s <and >/s <. See 3.1 for examples of the transcription

>s <THE SIXTY ONE C >/s <
>s <WHEN IS THE NEXT SIXTY FOUR A FROM SHADYSIDE TO SQUIRREL HILL >/s <
>s <SOUTH EIGHTEENTH STREET >/s <
>s <WHEN IS THE NEXT FIFTY NINE U FROM FIFTH AND ATWOOD >/s <

Table 3.1: Transcription Examples for the LM.

To make sure the used transcriptions are as close to the actual information contained in the audiofiles, we used the ”crowd_transcript” provided with the corpus as well as information in the field ”id” (see 2.2). Both fields were extracted using Java script (A.2 by copying content of both fields into a double column .csv file.

After these preperations, the pipeline for building the LM were as follows (all tools mentioned are provided with a download of the CMUCLMT):

1. Generate the vocabulary file containing all the words in the transcription. This was done using the *text2wfreq* tool - which creates a list of every word which occurred in the text, along with its number of occurrences - and the *wfreq2vocab* tool which will sort the most frequent 20,000 words alphabetically into a *.vocab* file.

2. Check the vocabulary file for misspellings like numbers that weren't converted correctly, general misspellings or unnatural words.These mistakes then were corrected in the transcription file itself and the vocabulary file was re-computed.

3. The vocabulary file and the transcriptions are then combined to compute an *id n-gram* file containing a numerically sorted list of n-tuples of numbers, corresponding to the mapping of the word n-grams relative to the vocabulary.

4. The *id n-gram* is then used to generate a *closed vocabulary* ARPA format n-gram weighted statistical language model (see table 3.2) by running the *idngram2lm* tool.

5. As a last step we then are left with converting the ARPA format LM to the CMU binary form (*.DMP*). This is done by running the *sphinx_lm_convert* tool.

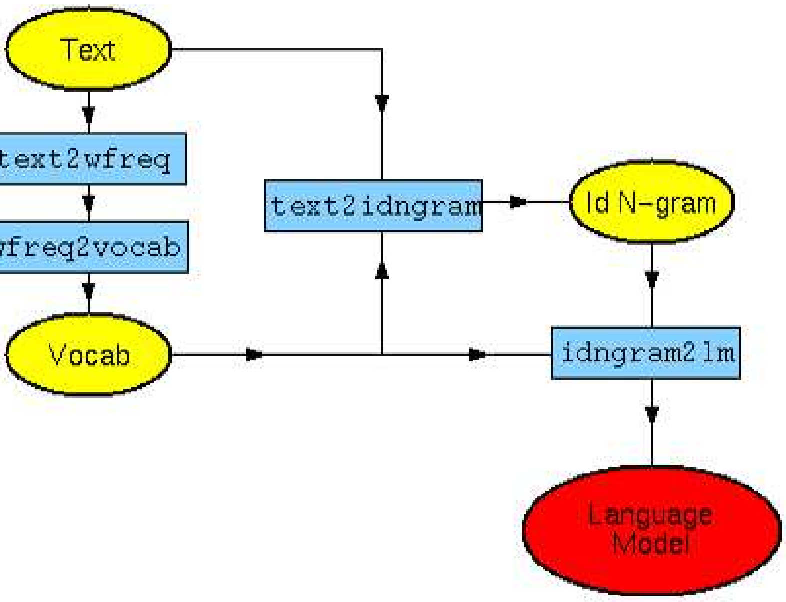


Figure 3.1: CMUCLMT Flowchart [1]

```
1-grams:
        -3.7839  board          -0.1552
        -2.5998  bottom         -0.3207
        -3.7839  bunch          -0.2174
2-grams:
        -0.7782  as  the                -0.2717
        -0.4771  at  all                 0.0000
        -0.7782  at  the                -0.2915
3-grams:
        -2.4450  in  the  lowest
        -0.5211  in  the  middle
        -2.4450  in  the  on
```

Table 3.2: Example for an ARPA standard LM

Chapter 4

Acoustic Model

Acoustic modeling of speech refers to the process of establishing statistical representations for the feature sequences computed from the speech waveform. While acoustic models (AM) include segmental models, neural networks, maximum entropy models, etc. Hidden Markov Models (HMM) are the one most common type of acoustic models. Acoustic modeling also encompasses ”pronunciation modeling”, which describes how a sequence or multi-sequences of fundamental speech units (e.g. phones) are used to represent larger speech units such as words or phrases which are the object of speech recognition. [8]

To build an acoustic model usable by Sphinx4, the Sphinx group has released a specialized training software - SphinxTrain. SphinxTrain requires a certain set of files to work. Below listed are the files with their extensions as well as a sample content [5]:

- *.train.fileids - List of audiofiles for training given as paths relative to the training folder

./20081001/000/003.raw.wav
./20081001/000/004.raw.wav

Table 4.1: Sample fileids

- *.train.transcription - Transcriptions for training. Each line contains the content of a single audiofile. Any given line in this file has to contain the transcription for the audiofile listed on the same line as in the *.fileids file.

Like the transcriptionfile used for training of the LM, utterances have to be delimited by >s <and >/s <tags. In addition, each line has to be followed by the fileid in parentheses. This ID may not contain the entire path but only the filename without any extensions.

>s <FIFTY NINE U >/s <(003)
>s <FORBES AND BIGELOW >/s <(004)

Table 4.2: Sample transcription

- *.dic - Phonetic dictionary Contains one word per line and its mapping to a sequence of phonemes. The same word may be represented multiple times with different phoneme representation.

ATLANTIC AH T L AE N T IH K
ATLANTIC(2) AH T L AE N IH K
ATTENTION AH T EH N SH AH N)

Table 4.3: Sample dictionary

- *.phone - Phoneset file A list of all the phonemes used in the phonetic dictionary.

D
DH
EH
ER

Table 4.4: Sample phones

- *.filler - List of fillers Non-speech sounds are mapped to corresponding non-speech or speech-like sound units. For training of this model only a rudimentary model containing only the start and end silences was used, as the transcription didn't contain any more sophisticated noise transcription.

>s <SIL
>sil <SIL
>/s <SIL

Table 4.5: Filler dictionary

- *.lm.DMP - Language model The binary language model built earlier.

All of these files are then moved into a project folder along with the files automatically created by SphinxTrain.

Chapter 5

Training and implementation

As there aren't many resources for information on how to configure SphinxTrain to achieve certain characteristics, the most important and complicated step of training the AM is to set up the configuration file. SphinxTrain is most commonly used to train on 16kHz, microphone recorded data. However, in this case it had to be set up for 8kHz telephone recorded data. This was done by modifying 3 values in the SphinxTrain configuration file [5]:

```
# Feature extraction parameters
$CFG_WAVFILE_SRATE = 8000.0;
$CFG_NUM_FILT = 31;
$CFG_LO_FILT = 200;
$CFG_HI_FILT = 3500;
```

Other than this change, for the first iteration we left everything else (e.g.: final number of Gaussian densities, number of tied states) unchanged/used recommended setting as per [5]. .

After several errors during training, we found out that training will only be successful if:

1. The .dic,.filler, .phones, and .transcription file have everything capitalized.
2. There is an empty line at the bottom of each file.
3. There is the same number of lines in the .transcription file as in the .fileids file
4. There are no duplicate entries in the .phones file.

A typical training circle turned out to take around 13 hours. After the first successful training, the configuration file of the Sphinx ASR system had to be modified to prepare usage of both the model and 8kHz files. This was done in accordance to [3].

After the first test run, the ASR failed to perform recognition on files containing just 1 word (e.g. ”yes” or ”no”) in 95% of those cases and misrecognized nearly 90% of multiword utterances.

As there are no guidelines in place to find optimal training paramenters, we proceeded to build multiple AMs using different tuning parameters and test them on the data from January 1st 2009.

Number of Gaussian Densities	Number of Tied-States (Senones)
8	3000
8	2000
16	3000
32	8000

Table 5.1: Tune parameters for initial line of testing

Running tests, the AM featuring 16 Gaussian Densities and 3000 Senones turned out to improve recognition the most. This improvement however still resulted in far more misrecognitions than would be tolerable for a live system.

To further try and improve performance we turned to the configration of Sphinx4 itself. The first part of the configuration file defines general properties of the recognition as shown in Table 5.2.

```
<property name=”absoluteBeamWidth” value=”500”/>
<property name=”relativeBeamWidth” value=”1E−80”/>
<property name=”absoluteWordBeamWidth” value=”20”/>
<property name=”relativeWordBeamWidth” value=”1E−60”/>
<property name=”wordInsertionProbability” value=”1E−16”/>
<property name=”languageWeight” value=”7.0”/>
<property name=”silenceInsertionProbability” value=”.1”/>
<property name=”frontend” value=”epFrontEnd”/>
<property name=”recognizer” value=”recognizer”/>
<property name=”showCreations” value=”false”/>
```

Table 5.2: Standard parameters for Sphinx4

As these standard configurations are designed to deal with real-time, clean microphone recorded speech, we tried to find parameters that would better fit the kind of audio in our data. Given that the data was collected using natural speech in natural environments our data was charged with a higher amount of background noise than these parameters would normally allow for. 4 values specifically turned out to be helpful in improving performance.

- *WordBeamWidth* (both absolute and relative) - With increasing value in both parameters, the ASR will take more words into the active word list for any given recognition, thus increasing the chance of including the correct word.
- *wordInsertionProbability* - Word break likelihood. This value sets a probability of which words are included in the initial hypothesis.
- *silenceInsertionProbability* - Likelihood of inserting silence. For noisier data, a smaller value will help avoiding failed recognitions.

After modifying values (see Table 5.3, ASR performance again increased slightly. However, single word utterances remained an issue as they more often than not would fail to be recognized at all and longer utterances were in many cases only partially recognized.

```
<property name="absoluteBeamWidth" value="500"/>
<property name="relativeBeamWidth" value="1E-120"/>
<property name="absoluteWordBeamWidth" value="200"/>
<property name="relativeWordBeamWidth" value="1E-100"/>
<property name="wordInsertionProbability" value="1E-6"/>
<property name="languageWeight" value="8.0"/>
<property name="silenceInsertionProbability" value=".01"/>
<property name="frontend" value="epFrontEnd"/>
<property name="recognizer" value="recognizer"/>
<property name="showCreations" value="false"/>
```

Table 5.3: Modified parameters for Sphinx4

This partial recognition of audio caused us to take a closer look again at the frontend.

```
<propertylist name="pipeline">
<item>audioFileDataSource </item>
<item>dataBlocker </item>
<item>speechClassifier </item>
<item>speechMarker </item>
<item>nonSpeechDataFilter </item>
<item>preemphasizer </item>
<item>windower </item>
<item>fft </item>
<item>melFilterBank </item>
<item>dct </item>
<item>liveCMN </item>
<item>featureExtraction </item>
 </propertylist>
```

Table 5.4: Standard frontend of Sphinx

Of special concern here is the second to last *item* listed. *liveCMN*, where *CMN* stands for čepstral mean normalization, indicates that this configuration of the frontend was designed to deal with realtime audio. However, in our case we want to process pre-recorded audiofiles. In such case, the *batchCMN* is the more desirable item to use. 3 more items related to the processing of live audio had to be removed in order to get the system running again.

```
    <propertylist name="pipeline">
    <item>audioFileDataSource </item>
    <item>dataBlocker </item>
    <item>preemphasizer </item>
    <item>windower </item>
    <item>fft </item>
    <item>melFilterBank </item>
    <item>dct </item>
    <item>batchCMN </item>
    <item>featureExtraction </item>
</propertylist>
```

Table 5.5: Modified frontend of Sphinx

With these alterations the ASR no longer failed to recognize longer utterances and the overall performance increased significantly. After further research regarding performance enhancing configurations when using 8kHz prerecorded audio, we encountered the usage of *Maximum Likelihood Linear Transform (MLLT)* tables. These tables have to be computed during training and are especially helpful when dealing with changing environments and speakers as is the case with our data. To use the newly created tables, the following lines have to be added to the configuration of Sphinx after inserting the item *featureTransform* into the frontend:

```
    <component name="featureTransform"
     type="edu.cmu.sphinx.frontend.feature.FeatureTransform">
    <property name="loader" value="wsjLoader"/>
</component>
```

Table 5.6: Using the MLLT

As a result, the final version of the AM is a 16 Guassian, 3000 Senones 8kHz model including an MLLT transformation table. Comparing the performances of both the originally used ASR and the one based on the newly trained AM reveals a significant boost in accuracy. While the original data contained a total of 73643 utterances with at least one misrecognition, this number was reduced to 50759 utterances using the new AM.

Forced Alignment

Sphinx4 does not come prepared to do forced alignment so implementing this feature took some re-writing of the ASR configuration. Theses changes can be seen in A.4.

After running forced alignment, the timestamps for every word were inserted as shown in table 5.7

20081001/000/013.raw.wav,fifty(0.8;1.35) nine(1.35;1.62) u(1.62;3.07)
20081001/000/014.raw.wav,no(0.47;1.6)
20081001/000/015.raw.wav,yes(0.03;1.41)

Table 5.7: Forced alignment examples.

Bibliography

[1] Philip Clarkson. The cmu-cambridge statistical language modeling toolkit manual. http://www.speech.cs.cmu.edu/SLM/toolkit_documentation.html.

[2] CMU. Let's go!: A spoken dialog system for the general public. http://www.speech.cs.cmu.edu/letsgo/, 2006.

[3] CMU. How to use models from sphinxtrain in sphinx-4. http://cmusphinx.sourceforge.net/sphinx4/doc/UsingSphinxTrainModels.html, 2009.

[4] CMU. Building language model. http://cmusphinx.sourceforge.net/wiki/tutoriallm, 2012.

[5] CMU. Training acoustic model for cmusphinx. http://cmusphinx.sourceforge.net/wiki/tutorialam, 2012.

[6] Antoine Raux et al. Lets go public! taking a spoken dialog system to the real world. Technical report, Language Technologies Institute, Carnegie Mellon University, 2005.

[7] Maxine Eskenazi Gabriel Parent. Toward better crowdsourced transcription: Transcription of a year of the lets go bus information system data. Technical report, Language Technologies Institute, Carnegie Mellon University, 2011.

[8] Microsoft Research. Acoustic modeling. http://research.microsoft.com/en-us/projects/acoustic-modeling/.

Appendix A

Appendix

A.1 Numericals to Words Conversion

```
###############################
#convert bus numbers into words
# 51[a-z] -> fifty one
###############################

import re



#input is a word, if this word matches a bus format, expand it to words
# else return the word
def getbusstr(str):
    matchbus = re.search("([0-9]?)([0-9])([a-zA-Z])", str);
    retstr = "";

    if matchbus:
       ten = matchbus.group(1)
       one = matchbus.group(2)
       letter = matchbus.group(3)

       if(ten=='1'):
               if(one=='0'): retstr += "ten"
               if(one=='1'): retstr += "eleven"
               if(one=='2'): retstr += "twelve"
               if(one=='3'): retstr += "thirteen"
               if(one=='4'): retstr += "fourteen"
               if(one=='5'): retstr += "fifteen"
               if(one=='6'): retstr += "sixteen"
               if(one=='7'): retstr += "seventeen"
               if(one=='8'): retstr += "eighteen"
               if(one=='9'): retstr += "nineteen"
       else:
               if(ten=='2'): retstr += "twenty"
               if(ten=='3'): retstr += "thirty"
               if(ten=='4'): retstr += "fourty"
               if(ten=='5'): retstr += "fifty"
               if(ten=='6'): retstr += "sixty"
               if(ten=='7'): retstr += "seventy"
               if(ten=='8'): retstr += "eighty"
               if(ten=='9'): retstr += "ninety"

               if(one and retstr!=""): retstr += " "

               if(one=='1'): retstr += "one"
               if(one=='2'): retstr += "two"
               if(one=='3'): retstr += "three"
               if(one=='4'): retstr += "four"
               if(one=='5'): retstr += "five"
               if(one=='6'): retstr += "six"
               if(one=='7'): retstr += "seven"
               if(one=='8'): retstr += "eight"
               if(one=='9'): retstr += "nine"

       if(letter):
           if (retstr != ""): retstr += " "
           retstr += letter;
    else:
       retstr = str #if there is no match, return the word

    return retstr

def process(str):
   words = str.split()
   resultstr = ""
   for w in words:
      if (resultstr!=''): resultstr += " "
      resultstr += getbusstr(w)
   return resultstr
```

```
#print process("tata 11B blabla")
#print process("54C")
#print process("1U")
```

```
###############################
#convert numbers into words
# 51 -> fifty one
###############################

import re



#input is a word, if this word matches a bus format, expand it to words
# else return the word
def getbusstr(str):
    matchbus = re.search("(([0-9]?)([0-9]?)([0-9]?)([0-9]))", str);
    retstr = "";

    if matchbus:
       thousand = matchbus.group(1)
       hundred = matchbus.group(2)
       ten = matchbus.group(3)
       one = matchbus.group(4)





       if(thousand=='1'): retstr += "one thousand "
       if(thousand=='2'): retstr += "two thousand "
       if(thousand=='3'): retstr += "three thousand "
       if(thousand=='4'): retstr += "four thousand "
       if(thousand=='5'): retstr += "five thousand "
       if(thousand=='6'): retstr += "six thousand "
       if(thousand=='7'): retstr += "seven thousand "
       if(thousand=='8'): retstr += "eight thousand "
       if(thousand=='9'): retstr += "nine thousand "

       if(hundred=='1'): retstr += "one hundred "
       if(hundred=='2'): retstr += "two hundred "
       if(hundred=='3'): retstr += "three hundred "
       if(hundred=='4'): retstr += "four hundred "
       if(hundred=='5'): retstr += "five hundred "
       if(hundred=='6'): retstr += "six hundred "
       if(hundred=='7'): retstr += "seven hundred "
       if(hundred=='8'): retstr += "eight hundred "
       if(hundred=='9'): retstr += "nine hundred "

       if(ten=='1'):
               if(one=='0'): retstr += "ten "
               if(one=='1'): retstr += "eleven "
               if(one=='2'): retstr += "twelve "
               if(one=='3'): retstr += "thirteen "
               if(one=='4'): retstr += "fourteen "
               if(one=='5'): retstr += "fifteen "
               if(one=='6'): retstr += "sixteen "
               if(one=='7'): retstr += "seventeen "
               if(one=='8'): retstr += "eighteen "
               if(one=='9'): retstr += "nineteen "
       else:
               if(ten=='2'): retstr += "twenty "
               if(ten=='3'): retstr += "thirty "
               if(ten=='4'): retstr += "fourty "
               if(ten=='5'): retstr += "fifty "
               if(ten=='6'): retstr += "sixty "
               if(ten=='7'): retstr += "seventy "
               if(ten=='8'): retstr += "eighty "
               if(ten=='9'): retstr += "ninety "

               if(one=='1'): retstr += "one"
               if(one=='2'): retstr += "two"
               if(one=='3'): retstr += "three"
               if(one=='4'): retstr += "four"
               if(one=='5'): retstr += "five"
               if(one=='6'): retstr += "six"
               if(one=='7'): retstr += "seven"
               if(one=='8'): retstr += "eight"
               if(one=='9'): retstr += "nine"


    else:
       retstr = str #if there is no match, return the word

    return retstr

def process(str):
   words = str.split()
   resultstr = ""
   for w in words:
      if (resultstr!=''): resultstr += " "
      resultstr += getbusstr(w)
   return resultstr

#print process("tata 11B blabla")
#print process("54C")
#print process("1U")
```

A.2 Extraction of Transcriptions

```java
import java.io.File;

public class getgoodtranscriptions {

    public static void main(String[] args) {
        try {

            CsvReader transciption = new
CsvReader("D:/TU/Master/Dip/CU/LGT/letsgo_transcript_2008_2009_v4.csv");

            CsvWriter output = new CsvWriter(new
FileWriter("D:/TU/Master/Dip/CU/LGT/asrconf.csv", true), ',');
            //CsvWriter output2 = new CsvWriter(new
FileWriter("D:/TU/Master/Dip/CU/LGT/correctfiles.txt", true), ',');
            CsvWriter output2 = new CsvWriter(new
FileWriter("D:/TU/Master/Dip/CU/LGT/am_transcription_align.txt", true), ',');
            CsvWriter output3 = new CsvWriter(new
FileWriter("D:/TU/Master/Dip/CU/LGT/lm_transcription_hicon.txt", true), ',');
            CsvWriter output4 = new CsvWriter(new
FileWriter("D:/TU/Master/Dip/CU/LGT/transcription_hicon.txt", true), ',');

            output.write("id");
            output.write("transcription");
            output.endRecord();
            output2.write("transcription");
            output2.endRecord();
            output3.write("transcription");
            output3.endRecord();
            output4.write("transcription");
            output4.endRecord();

            transciption.readHeaders();
            int s = 0;

            while (transciption.readRecord())
            {
                String fileID = transciption.get("id");
                String label = transciption.get("label");
                String asr = transciption.get("asr_output");
                String fconf = transciption.get("first_confidence");
                String ctrans = transciption.get("crowd_transcript");
                String sconf = transciption.get("second_confidence");
                String sec = transciption.get("sec");
                String purpose = transciption.get("purpose");

                Float ficonf = new Float(fconf);


                //String fileID = transciption.get("id");
                //String ctrans = transciption.get("transcription");

                // perform program logic here
                if((label.equals("understandable_correct") ||
label.equals("understandable_incorrect"))){

                //System.out.println(fileID + ":" + cTranscript);
                    ctrans = ctrans.replaceAll("%", " ");
                    s++;
                    //output.write(fileID);
                    //output.write(label);
                    //output.endRecord();
                    output.write(fileID);
                    output.endRecord();
                    output2.write("<s> " + ctrans.toUpperCase() + " </s> " + "(" +
fileID.substring(15, 18) + ")");
                    output2.endRecord();
                    output3.write("<s> " + ctrans.toUpperCase() + " </s> ");
                    output3.endRecord();
                    output4.write(ctrans.toUpperCase());
                    output4.endRecord();
                }

            System.out.printf("%d",s);
            transciption.close();
            output.close();
            output2.close();
            output3.close();
            output4.close();

        }

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

    }

}
```

A.3 ASR

```
 * Copyright 1999-2004 Carnegie Mellon University.

import edu.cmu.sphinx.frontend.util.AudioFileDataSource;

/** A simple Lattice demo showing a simple speech application that generates a Lattice
from a recognition result. */
public class simpleasr {


    /** Main method for running the Lattice demo. */
    public static void main(String[] args) throws IOException,
UnsupportedAudioFileException {
        URL audioURL, configURL;
        String outURL;

        if (args.length > 0) {
            audioURL = new File(args[0]).toURI().toURL();
        } else {
            audioURL = new URL("file:///D:/TU/Master/Dip/017.wav");
            //audioURL = LatticeDemo.class.getResource("10001-90210-01803.wav");
        }

        if (args.length > 1) {
            configURL = new File(args[1]).toURI().toURL();
        } else {
            configURL = new
URL("file:///D:/TU/Master/Dip/sphinx4-1.0beta6/Sphinx/src/apps/edu/cmu/sphinx/demo/latt
ice/configlg.xml");
            //url = new URL("file:///D:/TU/Master/Dip/sphinx4-
1.0beta6/Sphinx/sc/apps/edu/cmu/sphinx/demo/lattice/configlg.xml");
        }

        if (args.length > 2) {
            outURL = args[2];
        } else {
            outURL = "D:/asr.txt";
            //url = new URL("file:///D:/TU/Master/Dip/sphinx4-
1.0beta6/Sphinx/sc/apps/edu/cmu/sphinx/demo/lattice/configlg.xml");
        }

        FileWriter fstream = new FileWriter(outURL,true);
        BufferedWriter out = new BufferedWriter(fstream);

        ConfigurationManager cm = new ConfigurationManager(configURL);

        Recognizer recognizer = (Recognizer) cm.lookup("recognizer");
        recognizer.allocate();

        // configure the audio input for the recognizer
        AudioFileDataSource dataSource = (AudioFileDataSource)
cm.lookup("audioFileDataSource");
        dataSource.setAudioFile(audioURL, null);

        boolean done = false;
        while (!done) {
            /* This method will return when the end of speech
            * is reached. Note that the endpointer will determine
            * the end of speech.
            */
            Result result = recognizer.recognize();

            if (result != null) {
                String resultText = result.getBestResultNoFiller();
                out.write(args[0].substring(args[0].length() - 24));
                out.write(",");
                out.write(resultText.replaceAll(",",";"));
                out.newLine();
                out.close();

            } else {
                done = true;
            }
        }


    }
}
```

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

Philipp Salletmayr

Submitted in partial fulfillment of the
requirements for the degree
of Master of Science

Graz University of Technology & Columbia University in the City of New York

2012

ABSTRACT

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

Philipp Salletmayr

We address the problem of *localized error detection* in Automatic Speech Recognition (ASR) output. Localized error detection seeks to identify which particular words in a user's utterance have been misrecognized. Identifying misrecognized words permits one to create *targeted* clarification strategies for spoken dialogue systems, allowing the system to ask clarification questions targeting the particular type of misrecognition, in contrast to the *"please repeat/rephrase"* strategies used in most current dialogue systems. We present results of machine learning experiments using ASR confidence scores together with prosodic and syntactic features to predict whether 1) an utterance contains an error, and 2) whether a word in a misrecognized utterance is misrecognized. We show that by adding syntactic features to the ASR features when predicting misrecognized utterances the F-measure improves by 13.3% compared to using ASR features alone. By adding syntactic and prosodic features when predicting misrecognized words F-measure improves by 40%.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The ability to clarify information is important for successful dialogue communication. Human conversationalists ask clarification questions in every-day communication when they believe they have misunderstood their interlocutor. Automatic Spoken Dialogue 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, adopting simple statements indicating their lack of understanding followed by requests to the user to repeat or rephrase their input. While this behavior is general 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 detection of just which part of a user utterance has been recognized correctly and which part or parts contain an error. Previous research on error detection in ASR in general and in SDS applications in particular has focused on identifying simply how likely an utterance is to have been recognized correctly or incorrectly 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 from the user.

1.2 Related Work

1.2.1 Reprise vs non-reprise questions

In his study of human clarification strategies, Purver[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 the clarification questions used in the informal human conversations Purver studied contained only about 12% *non-reprise* clarification questions , 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 dialogue systems where speakers are required to specify values for a fixed number of predefined attributes and concepts. However, previous research has 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 and eventually user initiative, such as tutoring systems [Litman and Silliman, 2004] and speech to speech translation systems [Akbacak and others, 2009], SDS which can request more specific information about hypothesized ASR errors become more critical to create.

1.2.2 Detection of erroneous utterances

Handling errors in SDS involves first determining that an error has probably occurred and then choosing an appropriate dialogue strategy to correct it. There has been considerable work on detecting erroneous utterances in ASR systems and, more specifically, in SDS. Bohus and Rudiniki [Bohus and Rudnicky, 2005] analyse tradeoffs between misunderstandings and false rejections in a dialogue system. The authors optimize rejection thresholds using data-driven methods. Lopes et al. [Lopes *et al.*,] also analyse different feature sets for improving confidence score estimation in a dialogue system. Komatani and Okuno [Komatani and Okuno, 2010] use a user's utterance history to determine whether a barge-in user utterance has been correctly recognized. Our use of prosodic features is motivated by Hirschberg et al. [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, using prosodic information. Goldwater et al. [Goldwater and others, 2010] find 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 the characteristics they find characterizing misrecognized words in their data might be used to predict which words are misrecognized in practice in an SDS. Our work 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 dialogue strategies for error recovery. For example, Dzikovska et al. [Dzikovska and others, 2009] describe an approach to dealing with errors in tutoring dialogue systems. Bohus et al. [Bohus *et al.*, 2006] use supervised learning to determine the optimal error recovery policy in a dialogue system, such as providing a help message, repeating a previous prompt, or moving on to the next prompt. Our work on localized error detection is a study towards introducing a new policy type in a dialogue system: *asking a targeted clarification questions.*

1.3 BOLT

The 'Broad Operational Language Translation' *(BOLT)* program is funded by the 'Defense Advanced Research Projects Agency' *(DARPA)* of the 'United States Department of Defense'. BOLT's goals are twofold:

- Translation of informal language genres
- Bilingual, multi-turn conversation (both on text and speech level)

To achieve flexibility in regards 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 Dialogue Systems C. Human-Human Dialogue Systems D. Arabic Dialect Translation E. Grounded Language Acquisition F. Basic Technologies

All participating sites will have 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 Goals

In the work presented here, we seek to identify not only which utterances have been misrecognized but also which *portions* of utterances have been incorrectly transcribed by the recognizer, in order to use this information to formulate targeted reprise questions in a Speech-to-Speech (S2S) translation system. In such systems, two speakers communicate orally in two different languages through two ASR systems and two Machine Translation (MT) systems. An S2S system takes speech input, recognizes it automatically, translates the recognized input into text in another languages, and produces synthesized speech output from the translation for the conversational partner. In the S2S application we target, speakers may 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 intercept speech recognition errors early in the dialogue to avoid translating poorly recognized utterances.

Chapter 2

Materials

2.1 Overview

Presented in this chapter is an overview of materials used in the thesis with regards to speech data sets abailable for system development and research experiments as well as the ASR system used to process the audio data.

2.2 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 [Weiss and others, 2008] form the basis for the development done under BOLT. The data stems from 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 a *January release* and *May release.*

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

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 (2.3) 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 Utt.	3.729	2.664 (71.4%)	1.065 (28.6%)
All Words	26.098	23.720(90.9%)	2.378(9.1%)
Words in err. Utt.	7.48	5.45 (72.8%)	2.03 (27.2%)

Table 2.2: Data composition for January release.

	Total Words	**Correct Words**	**Err. Words**
Dev. 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.

2.3 Dynaspeak

Dynaspeak [Franco and others, 2002] is an ASR engine developed and distributed by SRI International. 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
- Supports continuous speech
- Dynamic grammar compilation
- Speaker independent
- Speaker adaptation
- Dynamic noise compensation

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. 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:

- 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 mis-recignitions (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

3.1 Overview

In the following chapter, all features used throughout experiments will be explained as well as how these features were extracted from avaiable data (see 2).

3.2 Features

An extensive list of used features:

- 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; This is 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; This is 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; This is 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; This is the average absolute base-10 logarithm of the difference between the amplitudes of consecutive periods, multiplied by 20.

- shimmerloc; This is the average absolute difference between the amplitudes of consecutive periods, divided by the average amplitude.
- jitterppq5; This is 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.
- jitterrap; This is 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; This is the average absolute difference between consecutive periods, in seconds.
- jitterloc; This is the average absolute difference between consecutive periods, divided by the average period.
- pctVoicebreaks; This is 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 16.6667 milliseconds are regarded as voice breaks.
- pctUnvoi; This is the fraction of pitch frames that are analysed as unvoiced in the analysed audio.
- 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.
- maxF0_NOSMOOTH; Maximum pitch without cutting outliers (highest and lowest 5%).
- minF0_NOSMOOTH; Minimum pitch without cutting outliers (highest and lowest 5%).
- sdF0; Standard Deviation 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%).

- analysed_dur; Analysed signal duration in ms.
- total_dur; Signal duration in ms. (Redundant)
- VCD2TOT; This is the fraction of pitch frames that are analysed as unvoiced in the analysed audio. (Redundant)
- ENGSTDEV; Standard deviation of energy in signal.
- ENGMEAN; Mean of energy in signal with 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%).
- F0STDEV; Standard deviation of energy in signal with cutting outliers (highest and lowest 5%).
- F0MED; Median of energy in signal with cutting outliers (highest and lowest 5%).
- F0MEAN; Mean of energy in signal with cutting outliers (highest and lowest 5%).
- F0MAX; Maximum of energy in signal with cutting outliers (highest and lowest 5%).
- F0MIN; Minimum of energy in signal with cutting outliers (highest and lowest 5%).
- HIGHTAGNEXT; Content tag for the following word.
- POSTAGNEXT; Stanford part of speech tag for the following word.
- HIGHTAGPREV; Content tag for the preceding word.
- POSTAGPREV; Stanford part of speech tag for the preceding word.
- HIGHTAGTHIS; Content tag for the current word.
- POSTAGTHIS; Stanford part of speech tag for the current word.
- ASRconfidenceAvgAll; Word ASR posterior averaged over entire utterance.
- ASRconfidenceAvg3; Word ASR posterior averaged over preceding, current and next word.
- ASRconfidence; Word ASR posterior for current word.

In our experiments, these features were available for both utterance- and word-level experiments as summarized in Table 3.1. Also denoted in this table are the feature subset affiliations for the acronyms *ASR*, *POS* and *SYN* which will be used to refer to these subsets.

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 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: Features used in the experiments

3.3 Recognition Tagging

The basic goal of this thesis is to be able to reliably classify 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* 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 hypothesis as well as 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 XX) to tag a word in the final hypothesis as either *correct* or *incorrect*. Tables 3.2 and 3.3 present an example as to how words would be classified based on logfile information. Note that deletions of words occurring in the transcript is not accounted for as such information wouldn't be available to a live-system.

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

Table 3.2: Example for tagging of an utterance.

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

Table 3.3: Example for tagging of an utterance.

Utterances are tagged as *incorrect*, if one or more words contained in the utterance are also tagged as such.

3.4 Prosodic Features

For the January released data, we extracted prosodic features from the audio file of each utterance using praat scripts. 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 by design as one goal was to test as many features as possible for their information gain as possible. Functionality of both scripts as well as an example output for one script are presented in tables 3.5 and 3.6. Both scripts are called simultaneously and use information regarding start and end time of words extracted by another script. This script analyzes Dynspeak logfiles and extracts information. One output of this script are start and end times of words. An example of the output of this script regarding word alignment - which serves as the input for both prosodic scripts - can be seen in table 3.4.

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 3.4: 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 3.5: 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 3.6: Example of output for extract_acoustics.pl script.

3.5 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 shows an example of the tagged output for the word-level.

```
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. The table shows POS as well CNT tags for current, previous and next words NULL meaning the information is not available (e.g. no previous/next word in the utterance).

3.6 Feature Exploration

3.6.1 Concept

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 remeasuring the error.
- starting with 1 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 then can go in either of 2 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 or decreases. 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 3 different scores: F-measure for both 'correct' and 'incorrect' labeled words (thus providing 2 scores) and accuracy representing the percentage of correctly classified instances in the dataset.

3.6.2 Results

Interestingly the final set of relevant features was the same for all 3 evaluated scores. This final dataset consisted of the features listed in table 3.8.

logconfidence
logconfidenceAvg3
POSTAGTHIS
POSTAGPREV
POSTAGNEXT
F0MIN
F0MAX
F0MEAN
ENGMAX
ENGMIN
ENGMEAN
ENGSTDEV
VCD2TOT
total_dur
analysed_dur
medianF0_NOSMOOTH
maxF0_NOSMOOTH
nPeriods
meanPeriod
sdPeriod
jitterloc
NHR

Table 3.8: Features used in the experiments

The scores for this dataset are:

Accuracy	0.845
F-measure correct	0.885
F-measure incorrect	0.669

For later experiments, the features

- nPeriods
- meanPeriod
- sdPeriod
- jitterloc
- 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

Modelling

Disclaimer: All training and testing was exclusively performed on TRANSTAC data. Testing features for the Let's Go! data set is part of future work on this project.

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, classification, regression, clustering, association rules, and visualization. [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 and also used due to the ease of implementation into existing software given the Java based base package. Also, for experiment purposes the built-in GUI provided the possibility of quickly visualizing, evaluating and comparing different types of classifiers or parameter sets. Also, featured from WEKA version 3.7 on a package manager was introduced which especially in the experiment stage allowed for fast obtaining and testing using newly proposed classifiers.

4.1.1 ARFF format

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 the 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 next 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 on how final states where achieved in a given machine learning problem to paths in the tree.

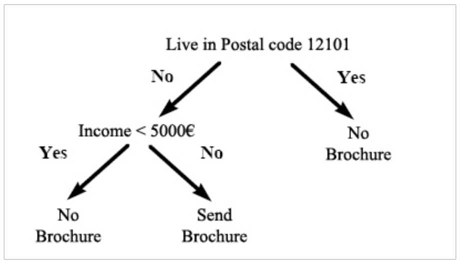


Figure 4.1: Example for a simple DT

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 as only for the most mundane tasks such an approach will result in a robust classifier. 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 analysing 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.

4.2.2 Multiboost Decision Trees

Boosting refers to a technique, which proposes the use of not just a single classifier to solve a classification problem, but to employ an ensemble or *committee* of such classifiers, where each classifier is to be consider ”weaker” 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 that 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.

For this thesis, a more sophisticated boosting algorithm was chosen, which was also easily available in the WEKA framework - the *MultiBoostAB* ([WEBB, 2000]) 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. However, many of the original examples may be repeated in the resulting training set while others may be left out as 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* which got 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. However, with AdaBoost the probability of picking each example is initially set to be 1/N, where N is the total samples available in the training set. These probabilities are then 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 bias of classifications. To benefit from both these important 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.

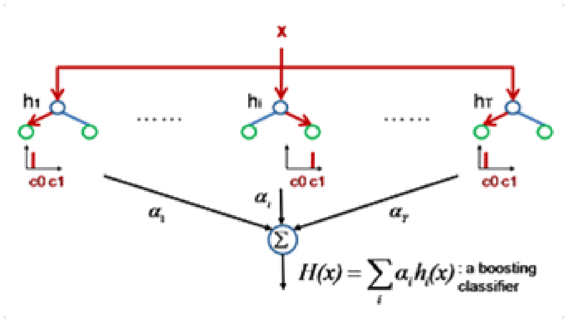


Figure 4.2: Example of a weighted decision committee

4.2.3 Support Vector Machines

Support Vector Machines (SVM) ([Cortes and Vapnik, 1995]) are a class of binary classifiers and are also used in regression problems. 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 **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.

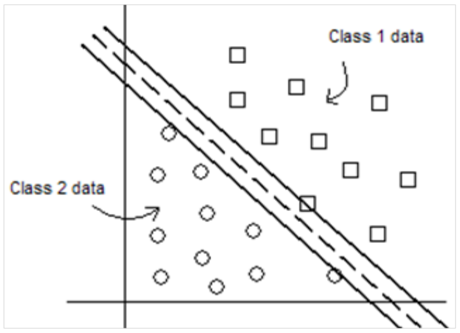


Figure 4.3: Simple linear SVM example

To train SVMs, 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. This eliminates the need to solve a quadratic programming problem and makes the solution analytically computable.

4.3 Training and testing

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. We present results of misrecognized utterance and word prediction experiments on ASR confidence, prosodic, and syntactic features, compared to the majority class baseline. We evaluate the effect of combining prosodic and syntactic features with the ASR confidence features. In these experiments, we perform 10-fold cross-validation on the full dataset using a J48 classifer. Table 4.1 shows precision, recall, and F-measure (also known as F1-measure,[1] 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 .93/.68. In order to create targeted clarifications, we are particularly interested in increasing the F-measure for detection of misrecognized utterances. We observe that by adding syntactic features to ASR features, the F-measure of 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**.

Our ultimate goal is to use the output of utterance misrecognition prediction as an input to word misrecognition prediction (2-stage prediction). We run this experiment on a subset of the data with the words from misrecognized utterances known from the reference

[1]F-measure $= 2 * recall * precision/(recall + precision)$

transcription 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.

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**.

The majority class baseline (predict correct recognition) achieves 72.8% overall accuracy, again failing to detect any of the incorrectly recognized words. Using the ASR confidence features alone, we achieve 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 misrecognized words compared to the classifier that uses only ASR features. These experiments show that the best performing feature combination for predicting misrecognized utterances is ASR+SYN and for words ASR+PROS+SYN. In the next set of experiments we use these feature sets to construct classifiers in 1-stage and 2-stage misrecognition prediction methods.

With this information, we start to look at the different classifiers and try to evaluate which one will be of best use to the later system implementation using the full feature set (ASR+PROS+SYN) for word prediction. To evaluate performance of each classifier, we use the June release of Transtac data (see section TODO). Thus, a very large set of samples for both training and testing is available. 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. This may be caused both by the unbalanced nature of training data (7% of all training samples are of class *incorrect*) as well as 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.

4.4 Results and Discussion

We evaluate 1-stage and 2-stage approaches to misrecognized word prediction. In a 1-stage approach, we predict misrecognition on all words in a 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 a 2-stage approach, we predict utterance misrecognition for each utterance in an ASR hypothesis. We consider an utterance to be misrecognized if the word error rate (WER) of the utterance is > 0. In the second stage, we predict 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 in each column are highlighted in **bold**.

We evaluate word-correctness prediction on the complete dataset using 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 train the utterance classifiers using all utterances in the training set. We train the misrecognized word classifiers using all words in the training set. We experiment with upsampling instances of misrecognized words in the training set to 35%[2] in order to improve performance of the classifier. Upsampling of an unbalanced dataset is a common procedure discussed in [Shriberg and Stolcke, 2002].

We evaluate 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 a 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 a 1-stage method trained on the original and upsampled datasets. We observe that, although the 1-stage method trained on the original dataset achieves higher overall accuracy (94.4%) than the 1-stage method trained on the upsampled dataset (93.7%), the upsampled training set achieves higher recall and F-measure (.60/.62) for predicting misrecognized words compared to original training set methods (.49/.60). Lines 4 and 5 show results for a 2-stage method trained on original and upsampled datasets. Both of the 2-stage methods achieve higher overall accuracy (94.5%) compared to the 1-stage methods. The 2-stage method trained on the original dataset achieves the highest precision for detecting misrecognized words of .85, while the 2-stage method trained on the upsampled dataset achieves the highest F-measure of .63.

All of the experimental methods improve overall accuracy performance by 2.4%-3.3% compared to the majority baseline. The highest performance improvement is achieved by the 2-stage predicting methods. The 2-stage method on the upsampled dataset achieves 52% recall and 76% precision in identifying misrecognized words. 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 as an answer to a clarification for a correct word is likely to support the original hypothesis.

[2]We derived this value empirically.

Chapter 5

Implementation

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 subsystem of which the confidence scoring module is a part of, is dealing with the Human-Machine communication. The goal of this communication is to make sure, that the human input as understood by the machine is as close to the actual input as possible. This is implemented by starting a dialog with the user asking questions with regards to the input as understood by the machine and confirming wether this was the intended meaning or not. Given the greater context of an actual human to human dialog taking place, this clarification dialog has to conform to standards ensuring maximum fluidity of the dialog as perceived by the interacting humans. These standard is enforced in the system by allowing for only a maximum of three 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. As of August 2012, the standard was relaxed to allowing four turns- the initial turn plus three clarification turns.

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 Overview

The proposed system was designed as a pipeline based on a central, multi-layered data structure (see 5.2). This structure is modified by the different components 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 5.1. The pipeline starts with a new speech input being recognized. During the recognition process the ASR, in addition to the 1-best transcription of the input, both the final confusion network as well as the lattice generated are being saved to the data structure. This information is used by the second component to try a re-scoring of the lattice to be able to find a better 1-best solution to the input. This step was however skipped in the final version of the system (August 2012) and just the original 1-best was used. The third component then tries to detect regions in the transcription which in the actual speech-information refer to words which are not covered by the ASR vocabulary (*out of vocabulary-OOV*) and marks those areas accordingly. 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. The next step is then to run the ASR 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 called is then the word-confidence scorer built in 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 then used by the *"Dialog Manager"* to determine wether or not (further) clarification questions are needed before the (combined) result can be handed on to the machine translation *(MT)*.

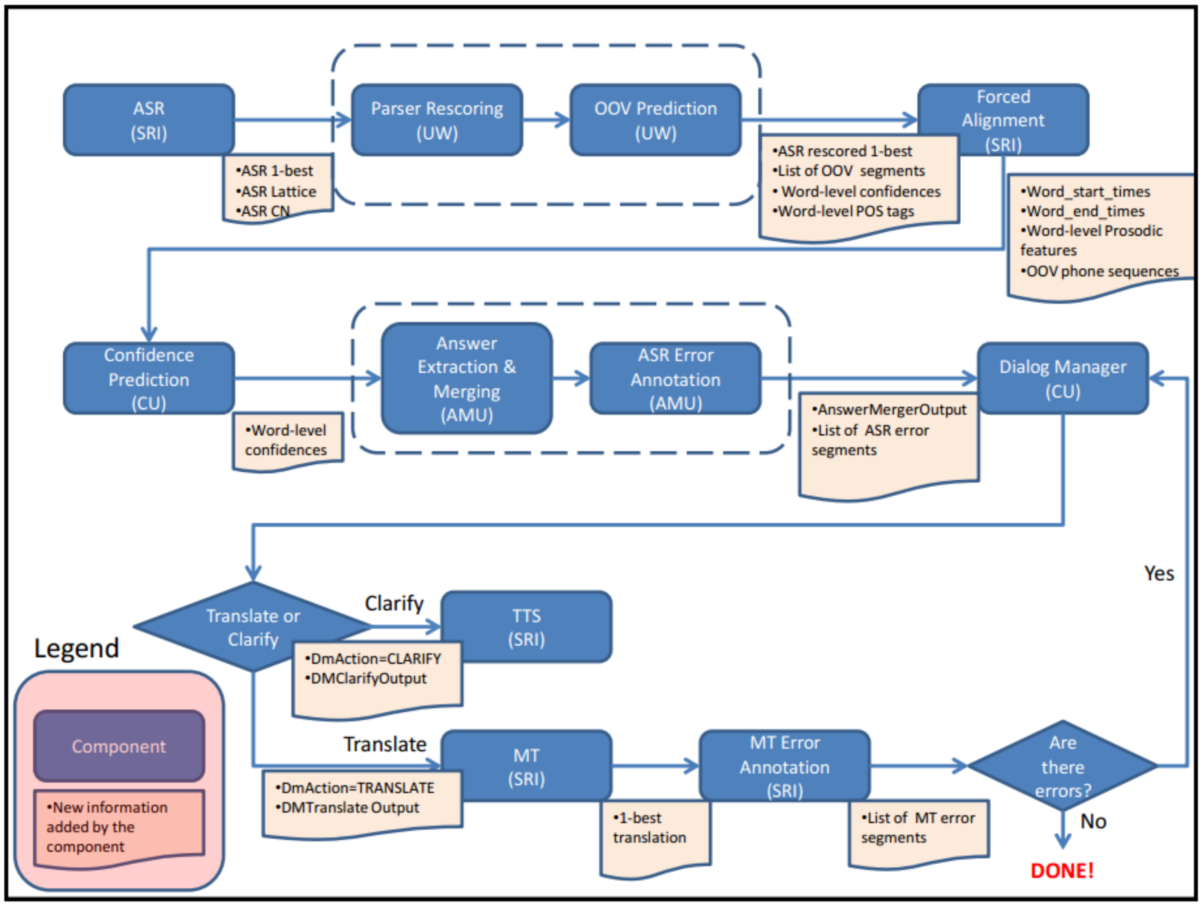


Figure 5.1: SRI's system pipeline

5.2 Google Protocol Buffers

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. When choosing a central, multi-layered data structure serving as the foundation for the pipeline used in the system, several criteria led to the choice of implementing Google Protocol Buffers:

- use of multiple programming and scripting languages throughout the different sites involved in the project, it was a primary goal to find a container supported by all those languages used or would be easy to adapt. Google Protocol Buffers primarily support C++, Java, and Python but community built support packages for different languages a readily available.

- 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. Data is just handed on in that case.

Creating multiple layers in the context of the BOLT system means that we categorize and save data at the following levels:

- 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.

A detailed design of the used buffer structure can be seen in APPENDIX

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 5.3.2) to a format usable by the classifier built in chapter 4. A UML diagram showing the major components of this can be seen in figure 5.2.

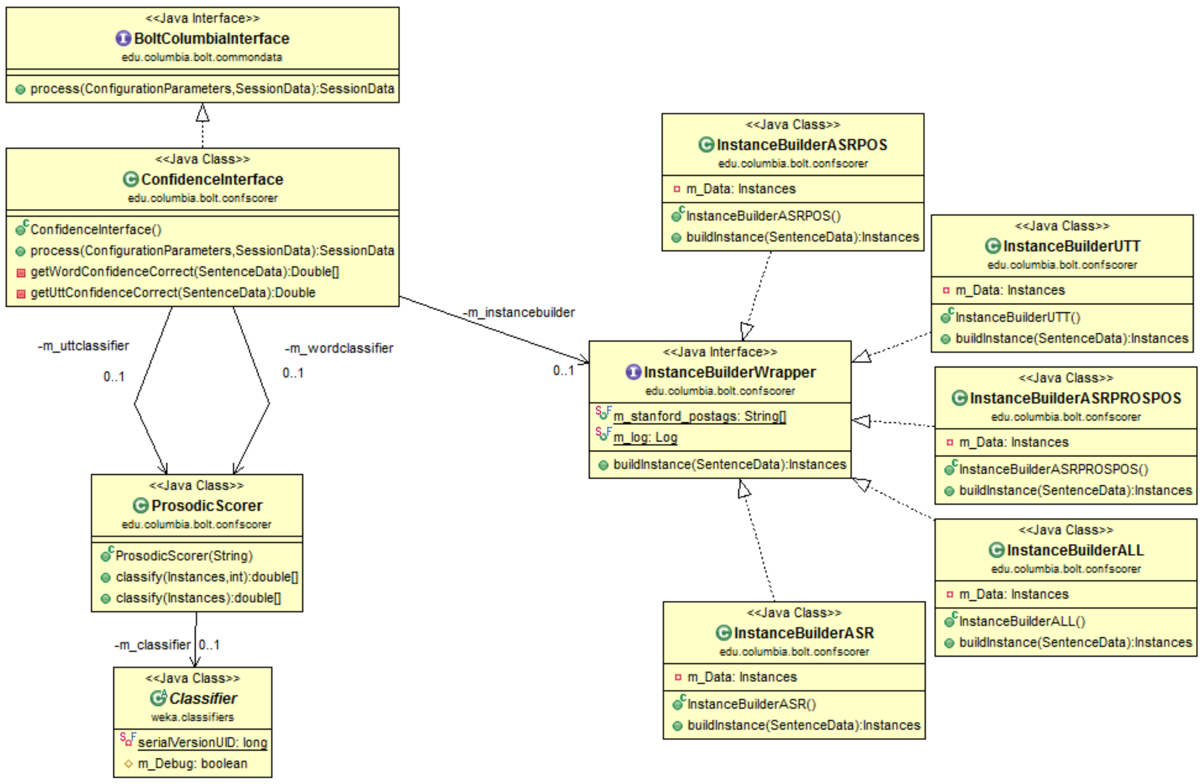

<<Java Interface>>
BoltColumbiaInterface
edu.columbia.bolt.commondata
process(ConfigurationParameters,SessionData):SessionData
<<Java Class>>
ConfidenceInterface
edu.columbia.bolt.confscorer
ConfidenceInterface()
process(ConfigurationParameters,SessionData):SessionData
getWordConfidenceCorrect(SentenceData):Double[]
getUttConfidenceCorrect(SentenceData):Double
-m_uttclassifier
0..1
-m_wordclassifier
0..1
-m_instancebuilder
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
<<Java Interface>>
InstanceBuilderWrapper
edu.columbia.bolt.confscorer
m_stanford_postags: String[]
m_log: Log
buildInstance(SentenceData):Instances
<<Java Class>>
InstanceBuilderASRPOS
edu.columbia.bolt.confscorer
m_Data: Instances
InstanceBuilderASRPOS()
buildInstance(SentenceData):Instances
<<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>>
InstanceBuilderASR
edu.columbia.bolt.confscorer
m_Data: Instances
InstanceBuilderASR()
buildInstance(SentenceData):Instances

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 a way 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.

```
public SessionData process(ConfigurationParameters conf, SessionData sessiondata)
```

This 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.

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

getWordConfidenceCorrect In this method the data arranged 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 any given being correctly classified. The returned information is then the filled out array of confidence values.

```
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 4).

The structure built is an *instances* file usable by WEKA (see 4.1) using the given set of features. 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.

```
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;
        }
```

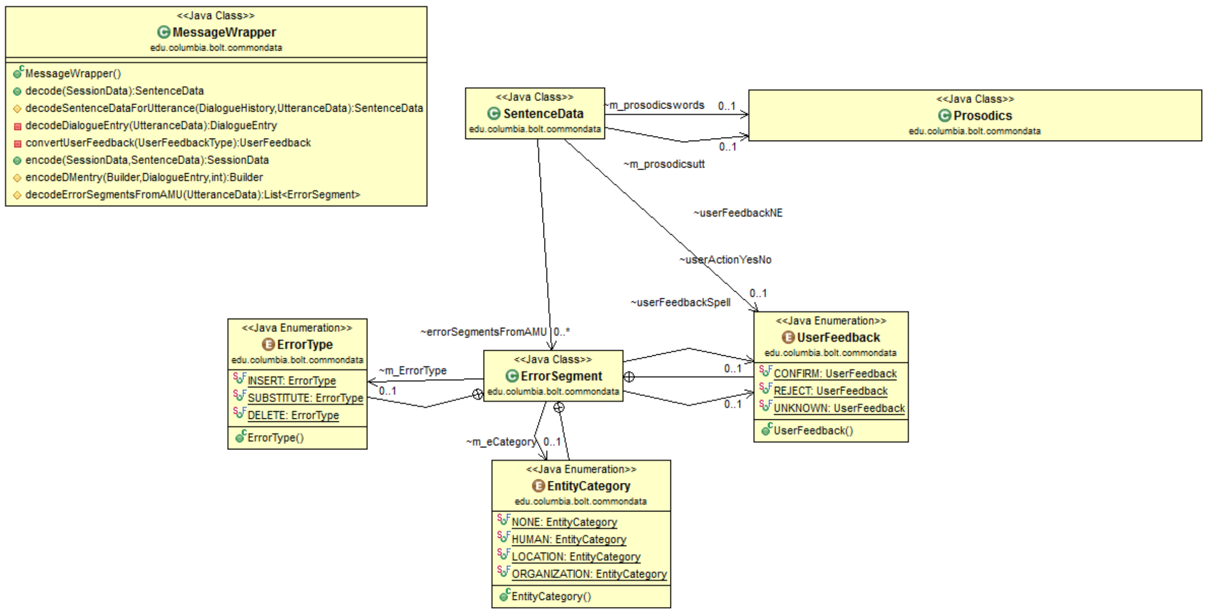
5.3.1.3 ProsodicScorer

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.

```
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 introduction of Google Protocol Buffers to the project, especially for early software tests there had to be a manual solution to the data transfer problem. 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 of this can be seen in figure 5.3.



<<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>
<<Java Class>>
SentenceData
edu.columbia.bolt.commondata
~m_prosodicswords 0..1
0..1
~m_prosodicsutt
<<Java Class>>
Prosodics
edu.columbia.bolt.commondata
~userFeedbackNE
~userActionYesNo
~userFeedbackSpell
0..1
~errorSegmentsFromAMU 0..*
<<Java Enumeration>>
ErrorType
edu.columbia.bolt.commondata
INSERT: ErrorType
SUBSTITUTE: ErrorType
DELETE: ErrorType
ErrorType()
~m_ErrorType
0..1
<<Java Class>>
ErrorSegment
edu.columbia.bolt.commondata
0..1
0..1
<<Java Enumeration>>
UserFeedback
edu.columbia.bolt.commondata
CONFIRM: UserFeedback
REJECT: UserFeedback
UNKNOWN: UserFeedback
UserFeedback()
~m_eCategory 0..1
<<Java Enumeration>>
EntityCategory
edu.columbia.bolt.commondata
NONE: EntityCategory
HUMAN: EntityCategory
LOCATION: EntityCategory
ORGANIZATION: EntityCategory
EntityCategory()

Figure 5.3: UML diagram of the common data 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.

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 both those modules and the Google Protocol Buffer structure.

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

```
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, here we want to change 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. At this layer we then have to write the computed *word confidence* to the field reserved for this information.

```
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 words contained in an utterance. Every word is represented by a certain index which stays the same throughout those List objects. The only information not saved internally in Lists objects are prosodic information which is held by a child-class called *Prosodics*. Also implemented in these structures are the getter and setter methods used to access the saved data.

```
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;
}
        .
        .
public void setProsodicsForWords(List<Double> f0max, List<Double> f0min, List<Double> f0mean, List<
     Double> f0stdev,
                                                      List<Double> engmax, List<Double> engmin, List<
                                                           Double> engmean, List<Double> engstdev, List<
```

```
                                                    Double> vcd2tot)
            throws SentenceDataException {

            checkSize(f0max);
            checkSize(f0min);
            checkSize(f0mean);
            checkSize(f0stdev);
            checkSize(engmax);
            checkSize(engmin);
            checkSize(engmean);
            checkSize(engstdev);
            checkSize(vcd2tot);

            m_prosodicswords.setProsodicsForWords(f0max, f0min, f0mean, f0stdev, engmax, engmin, engmean,
                 engstdev, vcd2tot);
        }
        .
        .

public List<Double> getF0maxWords() {
            return m_prosodicswords.getF0max();
        }

        public List<Double> getF0minWords() {
            return m_prosodicswords.getF0min();
}
}



public class Prosodics {

        //maximum pitch features for each word
        List<Double> f0max;
        //minimum pitch features for each word
        List<Double> f0min;
        .
        .
public void setProsodicsForWords(List<Double> f0max, List<Double> f0min, List<Double> f0mean, List<
     Double> f0stdev, List<Double> engmax, List<Double> engmin, List<Double> engmean, List<Double>
     engstdev, List<Double> vcd2tot) {

                    this.f0max = f0max;
                    this.f0min = f0min;
                    this.f0mean = f0mean;
                    this.f0stdev = f0stdev;
                    this.engmax = engmax;
                    this.engmin = engmin;
                    this.engmean = engmean;
                    this.engstdev = engstdev;
                    this.vcd2tot = vcd2tot;
            }
        .
        .
public List<Double> getF0max() {
                    return f0max;
```

```
                }

        public List<Double> getF0min() {
                return f0min;
        }
        .
        .
}
```

5.4 Results

TBD (awaiting result analysis by AMU)

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.

[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.*,] J. Lopes, M. Eskenazi, and I. Trancoso. Incorporating asr information in spoken dialog system confidence score. In *Computational Processing of the Portuguese Language*, Lecture Notes in Computer Science.

[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.

[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.