UNIVERSIDADE FEDERAL DO PARÁ
INSTITUTO DE CIÊNCIAS EXATAS E NATURAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DESENVOLVIMENTO DE RECURSOS E FERRAMENTAS PARA RECONHECIMENTO DE VOZ EM PORTUGUÊS BRASILEIRO PARA DESKTOP E SISTEMAS EMBARCADOS

RAFAEL SANTANA OLIVEIRA

Belém
2012

UNIVERSIDADE FEDERAL DO PARÁ
INSTITUTO DE CIÊNCIAS EXATAS E NATURAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

RAFAEL SANTANA OLIVEIRA

DESENVOLVIMENTO DE RECURSOS E FERRAMENTAS PARA RECONHECIMENTO DE VOZ EM PORTUGUÊS BRASILEIRO PARA DESKTOP E SISTEMAS EMBARCADOS

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal do Pará como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação.

Belém
2012

RAFAEL SANTANA OLIVEIRA

DESENVOLVIMENTO DE RECURSOS E FERRAMENTAS PARA RECONHECIMENTO DE VOZ EM PORTUGUÊS BRASILEIRO PARA DESKTOP E SISTEMAS EMBARCADOS

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal do Pará como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação.

Banca Examinadora:

Prof. Dr. Aldebaro Barreto da Rocha Klautau Junior
Programa de Pós-Graduação em Ciência da Computação - UFPA - Orientador

Prof. Dr. Nelson Cruz Sampaio Neto
Programa de Pós-Graduação em Ciência da Computação - UFPA - Co-orientador

Prof. Dr. Carlos Alberto Ynoguti
Instituto Nacional de Telecomunicações - Inatel - Membro

Prof. Dr. Eloi Luiz Favero
Programa de Pós-Graduação em Ciência da Computação - UFPA - Membro

Dr. Evandro Bacci Gouvêa
Instituto Nacional Vocalize - São Paulo - Membro

Dados Internacionais de Catalogação na Publicação (CIP)

Biblioteca Central da UFPa

Oliveira, Rafael Santana

Desenvolvimento de Recursos e Ferramentas para Reconhecimento de Voz em Português Brasileiro para Desktop e Sistemas Embarcados / (Rafael Santana Oliveira); Orientador, Aldebaro Barreto da Rocha Klautau Júnior. - 2012

Dissertação (Mestrado) - Universidade Federal do Pará, Instituto de Ciências Exatas e Naturais, Programa de Pós-Graduação em Ciência da Computação, Belém-Pa, 2012.

1. Reconhecimento Automático de Voz. I. Klautau Júnior, Aldebaro Barreto da Rocha, orient. II. Universidade Federal do Pará, Instituto de Ciências Exatas e Naturais, Programa de Pós-Graduação em Ciência da Computação. III. Título

CDD 22. ed. 006.31

Agradecimentos

À Deus, que me guiou e guardou ao longo de mais esta caminhada, agradeço primeiramente;

Ao meu orientador Aldebaro Barreto da Rocha Klautau Júnior e co-orientador Nelson Neto, agradeço pelos inúmeros aconselhamentos e apoio que tornaram possível que este trabalho fosse concluído;

Aos meus pais, José da Silva Oliveira e Conceição Santana Oliveira, agradeço pelo apoio incondicional que me proporcionou mais essa oportunidade de retribuir todo o esforço que tiveram para me dar uma boa educação;

Ao meu irmão, Rayleno Yan Santana Oliveira, melhor escritor que já conheci, agradeço pelas “críticas carinhosas”;

À minha companheira e amiga Rosa Cibele Borges dos Santos, agradeço pelo carinho e paciência que me fizeram olhar sempre para frente mesmo nos momentos de desânimo e pelas inúmeras revisões que tornaram este trabalho mais agradável de ser lido;

Aos meus amigos, principalmente aqueles que estiveram mais próximos durante esta jornada, agradeço por todo o apoio;

Por fim, agradeço à todos os colegas e amigos do LaPS, minha segunda família ao longo desses dois anos, que de forma direta e indireta participaram desta caminhada.

"O trabalho que nunca se começa é
o que mais demora para terminar."
(J.R.R. Tolkien)

Resumo

Nos últimos anos, a utilização de tecnologias de reconhecimento de voz teve crescimento notável. Com os investimentos de empresas como a Google, Microsoft e Nuance, uma gama de aplicativos com interface de voz, recursos e ferramentas vem sendo disponibilizados a comunidade. Contudo, para o Português Brasileiro (PB), esses recursos ainda são escassos. Nesse contexto, este trabalho teve como objetivo a disponibilização de ferramentas e recursos livres para o desenvolvimento de aplicativos com suporte a reconhecimento de voz para *desktop* e sistemas embarcados. Na primeira etapa deste, uma interface de programação de aplicativos compatível a especificação JSAPI foi desenvolvida para o *engine* Coruja tendo como objetivo sua disponibilização na plataforma Java. Visando plataformas móveis, na segunda etapa deste trabalho foram desenvolvidos recursos específicos para os decodificadores do grupo CMUSphinx. Modelos acústicos foram treinados e a viabilidade de conversão de modelos do formato do HTK para o Sphinx foi estudada. Como forma de validação dos recursos desenvolvidos, a nova versão do Coruja foi utilizada na refatoração do projeto SpeechOO. Um aplicativo com suporte a reconhecimento de voz foi construído para a plataforma Android 2.2 utilizando o decodificador Pocketsphinx, aliado a recursos desenvolvidos neste trabalho.

Palavras-chave: Reconhecimento automático de voz, JSAPI, CMUSphinx.

Abstract

In recent years, the use of speech recognition technology has expanded considerably. Duo to investments from companies like Google, Microsoft and Nuance, applications with speech interface, resources and tools has been made available to the community. However, there are few resources for Brazilian Portuguese (BP). Therefore, this work aimed to provide tools and resources for developing applications with support for speech recognition for desktop and embedded systems. First, an application programming interface (API) Java Speech API compliant was proposed to the speech recognition engine Coruja, aiming to make it available on Java development platform. Targeting mobile platforms, in the second part of this work were developed specific resources to the CMUSphinx group decoders. Acoustic models were trained and the viability of model conversion from HTK to Sphinx format was investigated. To validate the developed resources, the new version of Coruja was used to refactoring the SpeechOO project. An application with speech recognition support was built for the Android 2.2 platform using the Pocketsphinx decoder combined with resources develop in this work.

Keywords: Automatic Speech Recognition, JSAPI, CMUSphinx.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Publicações

No decorrer do curso de mestrado foram elaboradas três publicações, sendo uma em conferência, uma em simpósio e uma em *workshop*. Estas publicações estão ordenadas em ordem da mais importante e recente até a mais antiga, revelando a evolução gradual dos estudos até consolidação da proposta sugerida neste trabalho.

1- Oliveira, R.; Batista, P.; Neto, N.; Klautau, A. *Baseline Acoustic Models for Brazilian Portuguese Using CMU Sphinx Tools*. 12th International Conference on Computational Processing of the Portuguese Language - PROPOR 2012, Coimbra - Portugal, 2012.

2- Dantas, J.; Oliveira, R.; Santos H.; Neto, N.; Klautau, A. *Um Sistema para Melhorar a Usabilidade de um Gerenciador de Correio Eletrônico Baseado em Reconhecimento de Fala*. The 8th Brazilian Symposium on Information and Human Language Technology - STIL 2011, Mato Grosso - Brasil, 2011.

3- Oliveira, R.; Batista, P.; Neto, N.; Klautau, A. *Recursos para Desenvolvimento de Aplicativos com Suporte a Reconhecimento de Voz para Desktop e Sistemas Embarcados*. XII Workshop de Software Livre - WSL 2011, Porto Alegre - Brasil, 2011.

Lista de Abreviaturas

API *Application programming interface*

ASR *Automatic speech recognition*

CETUC Centro de Estudos em Telecomunicações

CD *Context dependent*

CI *Context independent*

CMU *Carnegie Mellon University*

DTW *dynamic time-warping*

HMM *Hidden markov models*

JNI *Java Native Interface*

JSAPI *Java Speech API*

JSGF *Java Speech Grammar Format*

LVCSR *Large Vocabulary Continuous Speech Recognition*

MFCC *Mel-frequency cepstral coefficients*

PB Português Brasileiro

PPGCC Programa de Pós-Graduação em Ciência da Computação

SAPI *Speech API*

TTS *Text-to-Speech*

UFPA Universidade Federal do Pará

WER *Word Error Rate*

Capítulo 1

Introdução

Desde a criação dos computadores existem esforços constantes a fim de facilitar a interação homem-máquina, objetivando assim tornar essa relação o mais natural possível. Nesse contexto, a utilização de tecnologias de processamento de voz como interface de comunicação entre o homem e o computador vem ganhando destaque. Em se tratando especificamente de reconhecimento automático de voz [Rabiner e Juang 1993, Huang et al. 2001] (ASR, de *automatic speech recognition*), com a melhoria constante do poder de processamento dos computadores pessoais, essa tecnologia vem sendo cada vez mais utilizada em um vasto número de aplicações como: sistemas para atendimento automático, aplicativos para controle de computadores pessoais como o Simon [Simon 2012], celulares com discagem por comando de voz e sistemas de ditado como o Via Voice da IBM e o Dragon Naturally Speaking da Nuance.

A adoção da tecnologia de reconhecimento de voz não só torna mais cômoda a tarefa de controlar o computador, como também permite que portadores de alguma limitação física ou necessidade especial possam fazer uso dessa ferramenta, possibilitando o acesso a informação por parte desse grupo de usuários que não são atendidos pelos meios típicos de interação com o computador (o *mouse* e o teclado).

Dada sua reconhecida importância, esforços constantes tem sido investidos nos últimos anos visando o desenvolvimento da tecnologia ASR. Dominado no passado por empresas especializadas, o mercado atualmente conta com participantes como a Microsoft e a Google, a investir fortemente no suporte de ASR no Windows [Odell e Mukerjee 2007] e no Chrome [Chrome 2012], por exemplo.

O surgimento dos *tablets* e *smartphones* e consequentemente de sistemas operacionais móveis como Symbian, iOS e Android, foi acompanhado pelo crescimento vertiginoso do poder de processamento dos dispositivos móveis. Esse fato permite que hoje uma gama de aplicativos possam ser desenvolvidos para esses aparelhos. Recentemente, empresas como a Google, Dextra, Apple Inc. e a Nuance lançaram uma série de aplicativos para plataformas móveis com funcionalidades de ASR que atraíram a atenção da comunidade. Dentre esses aplicativos estão os assistentes pessoais controlados por voz Siri [Siri 2011], Iris [Iris 2011] e o DragonGo! [DragonGO 2012], agentes inteligentes controlados por

voz. Além dos aplicativos mencionados, várias funcionalidades desses dispositivos já podem ser controladas por voz como: construir e enviar mensagens de texto, fazer pesquisas na internet, interagir com redes sociais, dentre outras.

1.1 Motivação

Apesar da notável importância e do crescimento da área de reconhecimento de voz em nível mundial, no Brasil as atividades tanto por parte da academia quanto da indústria ainda não alcançaram a dimensão necessária para que as mesmas tragam benefícios significativos à sociedade. Em consequência disso, não apenas desenvolvedores de *software* como também pesquisadores acabam se mantendo afastados dessa área de estudo. Dessa forma, os avanços na área de reconhecimento de voz para o Português Brasileiro (PB) acabam acontecendo de forma mais lenta do que acontecem para outros idiomas, e a sociedade acaba demorando a receber benefícios significativos proporcionados pelos avanços dessa área. As dificuldades oriundas da falta de recursos disponíveis prejudicam tanto o trabalho de grupos de pesquisa no meio acadêmico, interessados em dividir conhecimento e avançar no estado da arte da área, como também desenvolvedores de software e usuários com interesse de fazer uso da voz como interface em seus produtos e aplicações.

Acreditando que o desenvolvimento do estado da arte em ASR em PB depende da disponibilidade de recursos específicos para o nosso idioma e intimamente vinculado ao FalaBrasil [FalaBrasil 2012], projeto iniciado em 2009 no laboratório de processamento de sinais da Universidade Federal do Pará (LaPS) com objetivo de desenvolver e disponibilizar recursos e aplicativos para ASR em PB, e portanto, alinhado ao seu interesse na disseminação da tecnologia de voz em PB, essa pesquisa buscou através do desenvolvimento de recursos de domínio público, incentivar a pesquisa na área por parte de outros centros de pesquisa, disponibilizando tanto recursos práticos quanto recursos voltados para a academia, que podem servir de referência enquanto desenvolvendo trabalhos na área de reconhecimento de voz. Especificamente, este trabalho objetivou disponibilizar ferramentas e recursos para o desenvolvimento de aplicativos com suporte a ASR em PB, tanto para *desktop* quanto para sistemas embarcados.

Para o desenvolvimento de aplicativos com suporte a reconhecimento de voz se faz necessário o uso de um *engine* ASR para o idioma alvo, que consiste em um sistema de reconhecimento completo, composto por um decodificador e recursos específicos para um determinado idioma. Para *desktop*, existem soluções comerciais da Microsoft [MLDC 2012] e Nuance [Nuance 2012], contudo ambas não oferecem suporte ao PB. Em 2010, a Microsoft disponibilizou *engines* e ferramentas em versão beta para desenvolvimento de aplicativos específicos ao PB e PE [MLDC 2012]. Entretanto, tratam-se de sistemas de caráter proprietário. Outro *engine* ASR comercial para PB é o IBM ViaVoice, que foi descontinuado. Para *desktop*, a única *engine* ASR para PB livre disponível para *desktop* é o Coruja [Silva et al. 2010], que será discutido posteriormente.

Para dispositivos móveis, a Google disponibiliza uma API [Schuster 2010] (*Application Programming Interface*) que permite que desenvolvedores adicionem funcionali-

dades de ASR em vários idiomas a seus aplicativos, inclusive o PB. Essa API, permite que a aplicação acesse o serviço de reconhecimento provido pela Google, enviando o áudio e recebendo como resposta o resultado do reconhecimento, em um sistema de reconhecimento de voz distribuído (DSR, *Distribuited Speech Recognition*). Naturalmente, a utilização desse serviço exige que haja conexão com a internet. A Nuance também oferece serviço similar [DragonMobileSDK 2012]. Entretanto, não se trata de um recurso livre.

Existem também decodificadores livres que, com os devidos recursos, podem ser utilizados para construir *engines* para ASR em PB. O Julius [Lee 2009] e Sphinx-4 [Walker et al. 2004] são exemplos para *desktop*. Para sistemas embarcados, o decodificador Pocketsphinx [Huggins-Daines et al. 2006] é um dos mais populares.

Enquanto descrevendo recursos práticos para ASR, é oportuno citar as especificações *Speech* API [SAPI 2012] (SAPI) da Microsoft e *Java Speech* API [JSAPI 2012] (JSAPI) da Sun Microsystems, que padronizam o acesso a *engines* de voz e são atualmente as mais utilizadas pela comunidade para o desenvolvimento de aplicativos.

Pode-se notar que, poucos são os *engines* disponíveis para o desenvolvimento de aplicativos com suporte a ASR em PB, principalmente em se tratando de recursos livres. Contudo, a partir de decodificadores livres, é possível construir sistemas para ASR em PB desde que os recursos necessários para tanto sejam desenvolvidos. Outra forma de disponibilizar *engines* para ASR em PB é aumentar a abrangência de recursos existentes, como o Coruja, disponibilizando-o em plataformas ainda não suportadas.

1.2 Metodologia e contribuições

Tendo em vista a disponibilização de *engines* para ASR em PB para *desktop* e sistemas embarcados, este trabalho se dividiu em duas etapas. A primeira delas teve como foco a disponibilização do Coruja na plataforma de desenvolvimento Java. Para isso, este trabalho propôs a implementação de uma API em Java, chamada JLaPSAPI, para ser adicionada ao pacote de distribuição padrão do Coruja. Com o objetivo de adequar o Coruja a uma especificação largamente utilizada pela comunidade, a API proposta foi desenvolvida de acordo com a especificação JSAPI, tornando o Coruja uma opção de *engine* para ASR em PB para *desktop*.

O Coruja, tem suas funcionalidades de ASR baseadas no decodificador Julius, que por sua vez, não oferece suporte a plataformas móveis. Nesse contexto, visando a disponibilização de um *engine* para ASR em PB para plataformas móveis, esse trabalho buscou por alternativas ao decodificador Julius que oferecessem o suporte desejado. Por ser um decodificador livre e principalmente por contar com uma comunidade de desenvolvedores ativa, o decodificador Pocketsphinx foi escolhido para a continuação do trabalho. Como os modelos acústicos desenvolvidos até então pelo grupo FalaBrasil se basearam na ferramenta HTK [Young et al. 2006] e devido ao Pocketsphinx não oferecer suporte a esses modelos acústicos, a segunda etapa deste trabalho focou na construção de modelos

acústicos no formato suportado pelos decodificadores do grupo CMUSphinx[1], visando a construção de um sistema para ASR em PB baseado no decodificador Pocketsphinx.

O Sphinx, é um pacote de ferramentas de domínio público largamente utilizada na comunidade. Suas ferramentas vêm sendo utilizadas para a construção de sistemas para ASR em vários idiomas [Varela et al. 2003, Satori et al. 2009, Gulic et al. 2011] e o desempenho de seus decodificadores foi investigado em uma série de trabalhos [Vertanen 2006, Samudravijaya e Barot 2003, Ma et al. 2009]. Por oferecer decodificadores tanto para *desktop* quanto para sistemas embarcados, o Sphinx garante maior versatilidade quanto a plataforma de desenvolvimento.

Desde sua criação, os modelos acústicos desenvolvidos pelo grupo FalaBrasil foram baseados no HTK. A realização da segunda etapa deste trabalho exigiu, além da adequação de alguns recursos (p.e. dicionário fonético) utilizados para o treinamento de modelos acústicos, um investimento considerável de tempo para que fosse adquirido *know-how* sobre o pacote de ferramentas para treinamento de modelos acústicos do grupo CMUSphinx, o SphinxTrain [SphinxTrain 2012]. Tendo em vista o alto investimento necessário para que essa migração ocorresse dentro do nosso grupo de pesquisa, parte deste trabalho foi dedicado ao estudo da viabilidade de se converter modelos acústicos do formato do HTK para o Sphinx, a fim de prover informação útil para outros grupos de pesquisa. Durante a pesquisa, apenas um *software* com essa proposta foi encontrado, disponibilizado pelo grupo CMUSphinx. Entretanto, não foi encontrada qualquer avaliação sobre o mesmo, nem sequer relatos de desenvolvedores que tivessem conseguido utilizar a referida ferramenta. Nesse contexto, parte deste trabalho foi dedicado a prover tal avaliação, além de um relato sobre a utilização dessa ferramenta.

A metodologia de avaliação e validação dos recursos desenvolvidos ao longo das etapas deste trabalho consistiu em: primeiramente, por se tratar de um recurso de ordem prática, a nova versão do Coruja disponibilizada neste trabalho foi utilizada na refatoração do SpeechOO [Colen e Batista 2010], um projeto livre, desenvolvido com base na versão do Coruja que ainda não oferecia suporte a linguagem Java e não se adequava a uma especificação. Os modelos acústicos treinados utilizando o SphinxTrain, tiveram seus desempenhos avaliados para as tarefas de ditado e comando e controle. Para a segunda dessas tarefas, uma base de áudio, formada por comandos restritos ao contexto de uma aplicação de correio eletrônico foi desenvolvida. Já os modelos acústicos gerados a partir da conversão de formato do HTK para o do Sphinx foram avaliados apenas para a tarefa de ditado, e foram então comparadas as performances dos modelos convertidos em relação a dos modelos originais. Como prova de conceito, uma aplicação para a plataforma Android 2.2 foi construída utilizando os modelos acústicos desenvolvidos neste trabalho.

Em resumo, as principais contribuições deste trabalho são:

- Uma API em Java compatível com a especificação JSAPI para o *engine* para ASR em PB Coruja, desenvolvida visando sua disponibilização na poderosa plataforma

[1] http://cmusphinx.sourceforge.net/

de desenvolvimento Java e a adequação do mesmo a uma das especificações mais difundidas na comunidade de desenvolvimento;

- Modelos acústicos para os decodificadores Sphinx, que possibilitam o desenvolvimento de aplicativos com suporte a ASR em PB para *desktop* com o Sphinx-4 e principalmente para dispositivos móveis com Pocketsphinx;
- O *baseline* de treinamento utilizado no trabalho, além do dicionário fonético e *scripts* de apoio desenvolvidos durante a etapa de treinamento dos modelos acústicos;
- Uma base de áudio livre, desenvolvida neste trabalho para servir como referência para avaliação de desempenho de modelos acústicos no contexto de uma aplicação de correio eletrônico;
- Uma avaliação sobre a ferramenta para conversão de modelos acústicos disponibilizada pelo grupo CMUSphinx, além do *baseline* seguido para a utilização dessa ferramenta.

1.3 Trabalhos relacionados

Em [Ynoguti e Violaro 2001] foi construído um sistema de reconhecimento de fala contínua para um vocabulário médio. Uma base de áudio composta por 40 locutores adultos (20 homens e 20 mulheres), que pronunciaram 200 frases com 694 palavras diferentes foi construída. Nos testes para o modo independente de locutor, e para um vocabulário de aproximadamente 700 palavras, a taxa de erro por palavra obtida foi de 18,42%, com um tempo médio de reconhecimento por volta de 2 minutos em uma máquina com processador AMD-K6 de 350 MHz e 64 MB de memória RAM. Já nos testes para o modo dependente de locutor, com os locutores divididos por sexo, foram atingidas as taxas de erro por palavra de 13,93% para os locutores masculinos e 16,36% para os locutores do sexo feminino.

Em [Santos e Alcaim 2002, Fagundes e Sanches 2003], sistemas para ASR em PB foram desenvolvidos, entretanto, o vocabulário utilizado em ambos trabalhos eram muito reduzidos.

Silva et al. (2005), desenvolveu um sistema de reconhecimento automático de voz para grandes vocabulários (LVCSR, de *large vocabulary continuous speech recognition*) independente de locutor com mais de 60.000 palavras foi discutido. A melhor taxa de erro por palavra obtida foi de 37.42% Os resultados foram obtidos com uma quantidade relativamente pequena de dados de áudio extraídos do corpus Spoltech[2].

Em [Ênio 2008], um aplicativo demonstrativo com suporte a reconhecimento de dígitos foi construído utilizando um sistema ASR baseado no decodificador Sphinx-4. O

[2]http://www.ldc.upenn.edu/Catalog/catalogEntry.jsp?catalogId=LDC2006S16

modelo acústico utilizado no trabalho foi treinado com as ferramentas do HTK e então convertido para o formato do decodificador Sphinx-4. Não foram publicados resultados experimentais sobre o sistema criado e além disso, o trabalho não fornece informações suficientes para que o processo de conversão seja reproduzido.

Em [Teruszkin e Vianna 2006], um corpus de áudio proprietário gravado por um único locutor (a quantidade de horas de áudio não foi divulgada) foi utilizado para treinar modelos acústicos estocásticos dependentes de locutor. Um corpus de texto também foi elaborado para treinar modelos de linguagem. A melhor taxa de acerto para 60 mil palavras obtida pelo sistema foi de 81%, quando reconhecendo sentenças contendo de 9 a 12 palavras, com perplexidade dos modelos de linguagem variando entre 250 e 350 e tempo de processamento menor que um minuto por sentença. Todos os testes foram executados em um computador com processador Dual Intel (XeonTM 3.0 MHz) e 2 GB de RAM.

Em [dos Santos et al. 2010], um sistema de referência para o reconhecimento de fala contínua foi desenvolvido utilizando o corpus West Point Brazilian Portuguese Speech. Para o treinamento, 90% do corpus foi utilizado, e os 10% restantes foram utilizados nos testes de validação. A melhor taxa de acerto obtida foi de 93,3%. O tempo médio de reconhecimento não foi informado.

Durante a pesquisa realizada neste trabalho, não foram encontradas referências a trabalhos que utilizaram as ferramentas do grupo CMUSphinx para o treinamento de modelos acústicos para o PB.

1.4 Organização do trabalho

Capítulo 2. Reconhecimento automático de voz. Este capítulo apresenta um breve histórico sobre a tecnologia ASR e descreve os principais blocos dos sistemas ASR construídos neste trabalho. São descritas ainda as ferramentas utilizadas para o treinamento e teste dos modelos acústicos construídos.

Capítulo 3. Adequação do Coruja a JSAPI. Este capítulo apresenta a JLaPSAPI, API desenvolvida para o *engine* Coruja visando a disponibilização deste na plataforma de desenvolvimento Java, bem como a contribuição deste trabalho para o projeto SpeechOO.

Capítulo 4. Modelos acústicos para o Sphinx. Este capítulo descreve os *baselines* de treinamento e conversão utilizados neste trabalho para desenvolver modelos acústicos para os decodificadores Sphinx.

Capítulo 5. Resultados experimentais. Este capítulo aborda os resultados obtidos com a avaliação modelos dos acústicos construídos. É apresentado um aplicativo com suporte a ASR em PB desenvolvido para a plataforma Android 2.2 utilizando recursos disponibilizados neste trabalho.

Capítulo 6. Considerações finais. Por fim, neste capítulo é feita uma síntese do trabalho apresentando. Os resultados obtidos são avaliados e as ações previstas para o futuro são comentadas.

Capítulo 2

Reconhecimento automático de voz

Este capítulo provê uma introdução aos sistemas ASR. Primeiramente, um breve histórico sobre a tecnologia ASR e seu estado da arte são apresentados. Na sequência é descrita a arquitetura dos sistemas ASR construídos neste trabalho, descrevendo seus principais blocos.

2.1 Breve histórico sobre a tecnologia ASR

De acordo com [da Cunha e Velho 2003], os sistemas ASR têm sido estudados desde os anos 50 [Rabiner e Juang 1993, Huang et al. 2001, da Cunha e Velho 2003]. Nos Laboratórios Bell na mesma época, foi desenvolvido o primeiro reconhecedor de dígitos isolados com suporte a apenas um locutor. Ainda na década de 50, foi introduzido o conceito de redes neurais, mas devido a muitos problemas práticos, a ideia não foi seguida no âmbito de ASR. Durante os anos 70, a técnica predominante foi a *dynamic time-warping* (DTW) [Sakoe e Chiba 1978], um algoritmo que mede a similaridade entre duas sequências após alinha-las ao longo do tempo. Com o passar dos anos, a pesquisa foi evoluindo e muitas limitações técnicas foram sendo superadas.

Inicialmente, os sistemas de reconhecimento de voz tentavam aplicar um conjunto de regras gramaticais e sintáticas à fala [Jelinek e Frederick 1998]. Caso as palavras ditas caíssem dentro de um certo conjunto de regras, os sistemas poderiam determinar quais eram aquelas palavras. Para isso, cada palavra precisava ser dita separadamente, com uma pequena pausa entre elas. Devido à características como sotaques, dialetos e regionalismos os sistemas baseados em regras não tiveram muito sucesso.

Nos anos 80 vários pesquisadores iniciaram estudos para reconhecimento de palavras conectadas utilizando métodos estatísticos, com maior destaque para os modelos ocultos de Markov (HMMs) [Rabiner 1989, Juang e Rabiner 1991].

Atualmente, o estado da arte em reconhecimento de voz é representado pelo uso de HMMs, incrementado por técnicas como aprendizado discriminativo [Woodland e Povey 2002], seleção de misturas de gaussianas [Lee et al. 2001], maximização de mar-

gem [Cheng 2011], dentre outras. São também encontrados na literatura sistemas híbridos bastante robustos para a tarefa de reconhecimento de fala contínua[Cohen et al. 1992, Schwenk 1999], baseados no uso de HMMs em conjunto com redes neurais artificiais.

2.2 Principais blocos de um sistema ASR

Um sistema ASR típico, é composto por quatro blocos: *frontend*, modelo acústico, modelo de linguagem e decodificador, como indicado na Figura 2.1, que também mostra o dicionário fonético.

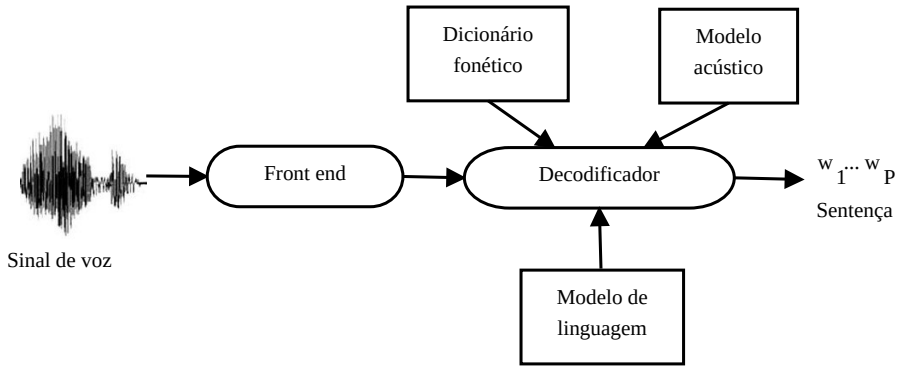


Figura 2.1: Principais blocos de um sistema ASR.

2.3 O *frontend*

O *frontend* é o parte do sistema ASR responsável por realizar a primeira fase no processamento da voz, que é a conversão analógico-digital, onde a onda analógica é traduzida em dados digitais. Devido a natureza aleatória do sinal de voz, utilizá-lo na sua forma original não é uma prática recomendável. Sendo assim, durante a etapa de *frontend* o sinal acústico é parametrizado em um processo de filtragem.

No processo de *frontend* convencional, tem-se a segmentação do sinal de voz em segmentos curtos (janelas, ou "frames") de 20 a 25 milisegundos, com o deslocamento da janela de análise sendo tipicamente de 10 milisegundos. Então, cada *frame* é convertido em um vetor $\mathbf{x}$ de dimensão L (tipicamente, $L = 39$). É assumido aqui que T *frames* estão organizados em uma matriz $\mathbf{X}$ de $L \times T$, representando uma sentença completa.

Existem várias alternativas no que diz respeito à parametrização do sinal de voz [Picone 1993, Junqua e Haton 1996, Huang et al. 2001]. Entretanto, a análise *Mel-frequency cepstral coefficients* (MFCCs) [Davis e Merlmestein 1980] tem provado ser eficiente e é geralmente empregada na construção de sistemas ASR [Huang et al. 2001].

2.4 Modelo acústico

O modelo acústico é a parte do sistema ASR que busca através de *features* extraídas do sinal acústico criar um modelo matemático que represente esse sinal, de forma que dado segmentos desconhecidos, estes possam ser mapeados de forma correta para palavras no caso de reconhecimento de palavras isoladas, ou para fonemas no caso de reconhecimento de fala contínua. Este trabalho tratou apenas de sistemas de reconhecimento de fala contínua. Sendo assim, cada fonema[1] presente nos modelos acústicos criados, foi representando por uma HMM. No estado da arte, modelos acústicos são representados por HMMs conectadas em sequência.

Uma cadeia de Markov [Rabiner 1989] consiste em uma máquina de estados finita que modifica seu estado a cada unidade de tempo. Em resumo, uma cadeia oculta de Markov consiste em um modelo matemático formado por uma cadeia de estados conectados entre si, onde para cada transição entre estados existe uma probabilidade de ocorrência associada. Além disso, para cada estado existe um processo estocástico vinculado, conhecido como processo de observação de saída, que pode ser discreto ou contínuo. A observação de saída representa uma ocorrência do fenômeno sendo modelado. Para o caso de sistemas para reconhecimento de fala contínua essas observações são fonemas. Para aplicações em ASR, a topologia "left-right" é adotada para estruturar as HMMs, onde as únicas transições permitidas são de um estado para ele mesmo ou para o estado seguinte.

Apesar de possível, a abordagem que define uma HMM para cada fonema (monofone) representado no modelo acústico não representa uma boa modelagem para fala. Devido ao fato dos articuladores do trato vocal não se moverem de uma posição para outra imediatamente na maioria das transições de fones, se faz necessário o uso de estratégias que levem em consideração essa característica do trato vocal. Nesse sentido, durante o processo de criação de sistemas que modelam a fala fluente, busca-se um meio de modelar os efeitos contextuais causados pelas diferentes maneiras que alguns fones podem ser pronunciados em sequência [Ladefoged 2001]. A solução encontrada é o uso de HMMs dependentes de contexto (trifones), que modelam o fato de um fone sofrer influência dos fones vizinhos. Por exemplo, supondo a notação do trifone *a-b+c*, temos que *b* representa o fone central ocorrendo após o fone *a* e antes do fone *c*. Essa é a notação tipicamente utilizada no HTK.

Segundo Huang et al. (2001), existem basicamente duas estratégias para a criação de modelos trifones: *internal-word* e *cross-word*. As diferenças entre as mesmas consiste em que, no caso da *internal-word* as coarticulações que extrapolam a duração das palavras não são consideradas, sendo assim, menos modelos são necessários. Já no caso do *cross-word*, a modelagem é mais precisa, visto que a mesma considera a coarticulação entre o final de uma palavra e o início da seguinte. Contudo, com a utilização dessa estratégia o número de modelos trifones gerados cresce muito, o que dificulta o trabalho do decodificador e gera uma necessidade de mais dados para treino. Alguns exemplos de transcrição podem ser conferidos na Tabela 2.1.

A partir do momento que se trabalha com modelos acústicos baseados em trifones, o

[1] O fonema é a menor unidade sonora que descreve um som.

Tabela 2.1: Exemplos de transcrições utilizando trifones.

	arroz com bife
Monofones	sil a R o s k o∼ b i f i sil
Internal-Word	sil a+R a-R+o R-o+s o-s k+o∼ k-o∼ b+i b-i+f i-f+i f-i sil
Cross-Word	sil sil-a+R a-R+o R-o+s o-s+k s-k+o∼ k-o∼+b o∼-b+i b-i+f i-f+i f-i+sil sil

número de HMMs cresce bastante. Por exemplo, assumindo um modelo acústico com 39 fonemas, aplicando a estratégia *cross-word* para a migração de monofones para trifones, aproximadamente 59.319 modelos seriam criados. Esse número elevado de modelos leva a um dos problemas clássicos da modelagem acústica que é a insuficiência de dados para estimar os modelos. O método de compartilhamento de parâmetros (ou "sharing") visa combater esse problema, melhorando a robustez dos modelos. Em muitos sistemas, o compartilhamento é implementado no nível de estado, ou seja, o mesmo estado pode ser compartilhado por HMMs diferentes.

Umas das técnicas para compartilhamento de estados é a *data-driven*, na qual ocorre a clonagem dos monofones seguida pela conversão para trifones, por fim todos os estados centrais dos trifones derivados do mesmo monofone são vinculados conforme a Figura 2.2. Nesta abordagem, assume-se que o contexto do trifone não afeta o estado central do modelo.

Outra técnica para compartilhamento de estados, descrita em [Bahl et al. 1994], consiste no uso de uma árvore de decisão fonética para unir os estados que são acusticamente semelhantes. Esse método envolve a construção de uma árvore binária utilizando um procedimento de otimização sequencial *top-down*, onde questionamentos são anexados a cada nó. As perguntas são relacionadas ao contexto fonético dos fones vizinhos ao fone analisado, como ilustra a Figura 2.3. Por exemplo, as questões são do tipo "o fonema à esquerda é nasal?", e dependendo da resposta (sim/não), uma das possíveis direções é seguida. Ao final do processo, todos os estados no mesmo nó folha são agrupados.

2.5 Modelagem da lingua

Apenas as informações acústicas não são suficientes para que um sistema ASR consiga obter um bom desempenho. Para que melhores resultados sejam alcançados, informações referentes a estrutura do idioma estudado devem ser incluídas no sistema. Essas informações podem ser providas ao sistema através de gramáticas livres de contexto ou através de modelos de linguagem.

Uma gramática provê ao reconhecedor regras que definem o que pode ser reconhecido. Mais especificamente uma gramática consiste em um conjunto de variáveis seguidas de expressões regulares descrevendo quais palavras podem ser reconhecidas e em que ordem. Assim, o reconhecedor busca pela sequência de palavras mais provável dentro dos limites

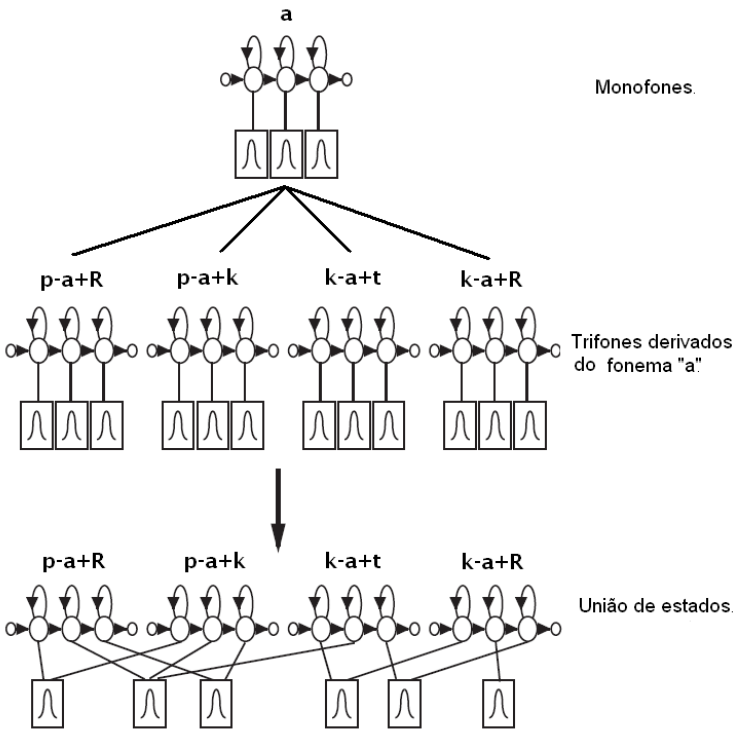


Figura 2.2: Compartilhamento de estados para fins de aumento da robustez da estimativa dos modelos HMM.

impostos pela gramática. Gramáticas são bem mais simples de se construir e apresentam melhores resultados, já que o espaço de busca durante o reconhecimento e bastante limitado.

Enquanto trabalhando com grandes vocabulários, a utilização de gramáticas se torna inviável, pois o esforço necessário para se construir uma gramática com todas as possíveis frases de um idioma, torna essa tarefa impraticável. Supondo um vocabulário de tamanho V, no reconhecimento de uma sequência de N palavras existem V^N possibilidades. Nesse sentido, faz-se uso de modelos de linguagem. Esses modelos buscam caracterizar a língua, tentando capturar suas regularidades e assim condicionar as combinações de palavras durante o reconhecimento, dado menor probabilidade aquelas frases cuja ocorrência não é permitida dado as regras gramaticais do idioma. O uso de modelos de linguagem leva a ganhos de desempenhos consideráveis [Lippmann 1997], proporcionando a redução no espaço de busca, o que implica na redução do tempo de reconhecimento. Além disso, a utilização de modelos de linguagem permite a resolução de ambiguidades acústicas [Pessoa et al. 1999].

De modo geral, um modelo de linguagem estima, de uma forma confiável, a probabilidade de ocorrência de uma determinada sequência de palavras $W = w_1, w_2, ...w_k$, onde k é o número de palavras na sentença. A probabilidade $P(W_1^k)$ é definida na equação 2.2

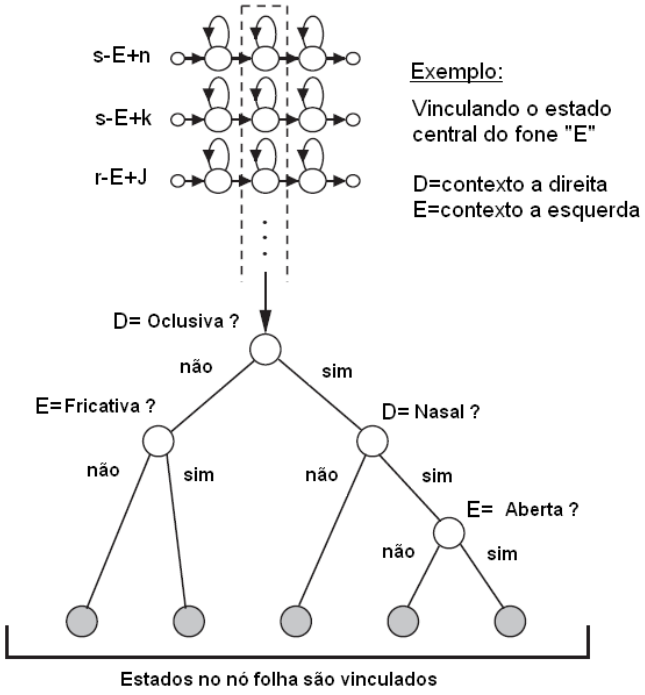


Figura 2.3: Compartilhamento de estados utilizando árvore de decisão fonética.

$$P(W_1^k) = P(w_1)P(w_2|w_1)P(w_3|w_1,w_2)...P(w_k|w_1,...,w_k-1) \quad (2.1)$$

$$P(W_1^k) = P(w_1)\prod_{i=2}^{k} P(w_i|w_1,...,w_i-1) \quad (2.2)$$

2.6 Decodificador

Até este ponto, foram apresentados os modelos matemáticos para construção de modelos acústicos e de linguagem que fornecem as informações necessárias para então a tarefa de reconhecimento da fala seja realizada. Essa etapa final, que consiste na transcrição de amostras de voz desconhecidas para sua forma textual, é desempenhada pelo decodificador. O processo de decodificação é controlado por uma rede de palavras construída a partir do modelo de linguagem. Essa rede, é composta por um conjunto de nós (palavras) conectados, onde cada conexão possui uma probabilidade de ocorrência (transição). De posse de tal rede, do dicionário fonético e do modelo acústico, pode-se determinar, dada uma amostra de voz desconhecida, a probabilidade de se percorrer qualquer um dos possíveis caminhos da rede. A tarefa do decodificador é encontrar dentre os possíveis caminhos aqueles que são mais prováveis.

O processo de busca é descrito pela Equação 2.3, onde, dada a sequência de obser-

vação $O_1^T = o_1, o_2 ... o_T$, busca-se pela sequência de palavras $W_1^N = w_1, w_2 ... w_N$ que maximize $\hat{W}_1^N$. O termo $P(O_1^T|W_1^N)$ pode ser expandido em função do modelo acústico, assim tem-se S_1^T como uma hipótese de sequência de estados, onde busca-se pelo W_1^N que maximize $P(O_1^T, S_1^T|W_1^N)$ levando em conta todas as possíveis sequências de estados. O somatório pode ser substituído por uma maximização, em um processo conhecido como aproximação de Viterbi [Jelinek 1976], em que se utiliza apenas o caminho mais provável. Essa busca é realizada através de programação dinâmica (DP - *Dynamic Programming* [Vintsyuk 1968]).

$$\begin{aligned} \hat{W}_1^N &= \arg\max_{W_1^N} \{P(W_1^N) P(O_1^T|W_1^N)\} \\ &= \arg\max_{W_1^N} \{P(W_1^N) \sum_{S_1^T} P(O_1^T, S_1^T|W_1^N)\} \\ &\simeq \arg\max_{W_1^N} \{P(W_1^N) \max_{S_1^T} (P(O_1^T, S_1^T|W_1^N))\} \end{aligned} \tag{2.3}$$

2.7 Métricas de avaliação

Após a etapa de decodificação vem a análise dos resultados. A medida de desempenho utilizada na maioria das aplicações que usam reconhecimento de voz é a taxa de erro por palavra (WER). Como tipicamente as transcrições correta e reconhecida possuem um número diferente de palavras, as mesmas são alinhadas através de programação dinâmica [Bellman 1957]. Dessa forma, de acordo com Huang et al. (2001), a WER pode ser definida como:

$$WER = \frac{S + D + I}{N} \tag{2.4}$$

onde N é o número de palavras na sentença de entrada, S e D e I são o número de erros de substituição, deleção e inserção na sentença reconhecida, respectivamente, quando comparada com a transcrição correta.

Outra métrica de avaliação é o fator de tempo-real (xRT). O fator xRT é calculado dividindo o tempo que o sistema despende para reconhecer uma sentença pela sua duração. Assim, quanto menor for o fator xRT, mais rápido será o reconhecimento.

2.8 Ferramentas utilizadas

Ao longo deste trabalho, uma serie de modelos acústicos foram treinados. O pacote de ferramentas SphinxTrain [SphinxTrain 2012] foi utilizado para o treinamento dos modelos acústicos para os decodificadores Sphinx. Já para o treinamento dos modelos acústicos para serem convertidos foi utilizado o pacote de ferramentas HTK. Para

a conversão dos modelos acústicos do formato do HTK para o Sphinx a ferramenta HTK2Sphinx [HTK2Sphinx 2012], distribuída pela grupo CMUSphinx, foi utilizada.

Para a avaliação dos modelos acústicos treinados para o Sphinx foram utilizados os decodificadores Sphinx3 e Pocketsphinx, ambos *softwares* livres. O Pocketsphinx, decodificador voltado para plataformas móveis, também foi utilizado para a construção de um sistema ASR que fez parte de um aplicativo desenvolvido para a plataforma Android 2.2. Para a avaliação dos modelos acústicos treinados para conversão, foi utilizado o decodificador Julius.

2.9 Conclusão

Este capítulo apresentou um breve histórico sobre a tecnologia ASR e descreveu os principais blocos dos sistemas ASR construídos neste trabalho. Foram também descritas as ferramentas utilizadas ao longo deste. O capítulo seguinte descreve a API em Java proposta para o *engine* Coruja, visando a disponibilização deste na plataforma de desenvolvimento Java.

Capítulo 3

Adequação do Coruja a JSAPI

Com o intuito de tornar o *engine* ASR Coruja uma ferramenta mais abrangente, este trabalho propôs a adição de uma API em Java compatível com a especificação JSAPI ao pacote original do Coruja, visando possibilitar sua utilização a partir da linguagem de programação Java. Este capítulo apresenta como se deu o processo de adequação do *engine* Coruja a especificação Java Speech API (JSAPI). Primeiramente, a especificação JSAPI é descrita, e na sequência o Coruja é apresentado. Em seguida, é descrita a JLaPSAPI, a API proposta neste trabalho para permitir o uso do Coruja através da especificação JSAPI. Como prova de conceito, é apresentado o projeto SpeechOO, um projeto livre que usa o recurso desenvolvido nesta etapa do trabalho.

3.1 Java Speech API

Para o desenvolvimento de aplicativos com suporte a reconhecimento automático de voz (ASR), é indispensável a presença de um *engine* de reconhecimento. Tendo um *engine* disponível, uma API facilita o trabalho do programador durante o processo de desenvolvimento. Cada reconhecedor possui sua própria API, mas existem especificações desenvolvidas para uso geral.

A Java Speech API [JSAPI 2012] é a especificação da Sun Microsystems que permite que desenvolvedores incorporem tecnologia de voz as suas aplicações desenvolvidas no ambiente de programação Java. A JSAPI especifica uma interface de programação independente de plataforma que suporta sistemas para ditado, comando e controle e síntese de voz. Deste ponto em diante, o termo JSAPI será utilizado neste trabalho fazendo referência apenas as funcionalidades de reconhecimento da API.

Do ponto de vista arquitetural, a JSAPI se encontra entre a aplicação e o *engine* ASR, disponibilizando um conjunto de classes e interfaces escritas em Java que representam a visão do programador sobre o *engine*, como pode ser visto na Figura 3.1.

Apesar da versão 1.0 da Java Speech API ter sido lançada pela Sun há mais de uma década, esta nunca comercializou qualquer tipo de implementação da JSAPI. As implemen-

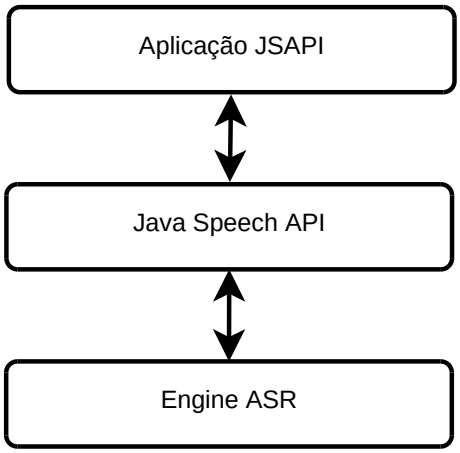


Figura 3.1: Arquitetura de alto nível de uma aplicação que usa a JSAPI.

tações existentes foram criadas por companhias que trabalham com ASR e TTS, como a IBM. A única implementação para ASR de código livre encontrada é o Sphinx-4 [Walker et al. 2004].

3.2 O Coruja

Em 2009, com o intuito de promover o amplo desenvolvimento de aplicações com suporte a reconhecimento de voz em PB, o grupo de pesquisa FalaBrasil da Universidade Federal do Pará (UFPA), disponibilizou a comunidade de desenvolvedores seu sistema para ASR, o Coruja [Silva et al. 2010], um *engine* de voz específico para o PB.

Esse *software*, é composto pelo decodificador livre Julius [Lee e Kawahara 2009]; modelos acústico e de linguagem para o reconhecimento de voz em PB; e uma API própria, implementada na linguagem de programação C/C++ chamada LaPSAPI. De modo geral, o Coruja pode ser visto como sendo o decodificador Julius, preparado para realizar ASR em PB, controlado a partir de uma API que facilita a operação desse decodificador.

A LaPSAPI foi projetada para esconder do programador detalhes de baixo nível relacionados à operação do Julius, permitindo o controle em tempo-real das funcionalidades deste e também da interface de áudio do sistema. Essa API pode ser utilizada tanto na plataforma Linux, a partir de implementações em C++, quanto na plataforma Windows, através de qualquer linguagem da plataforma Microsoft.NET (C#, Visual Basic, J#, entre outras), garantindo certa flexibilidade tanto quanto a plataforma, quanto a linguagem de programação. A Figura 3.2 ilustra o processo de interação entre uma aplicação e o Coruja.

Prover flexibilidade quanto a linguagem de programação utilizada para acessar o Coruja, foi uma das formas escolhidas pelos idealizadores do projeto para que este *engine* conseguisse atender ao maior número de desenvolvedores quanto possível, visando a disseminação desse recurso. Entretanto, a LaPSAPI não oferece suporte nativo a uma das linguagens de programação mais utilizadas pela comunidade de desenvolvimento, a lin-

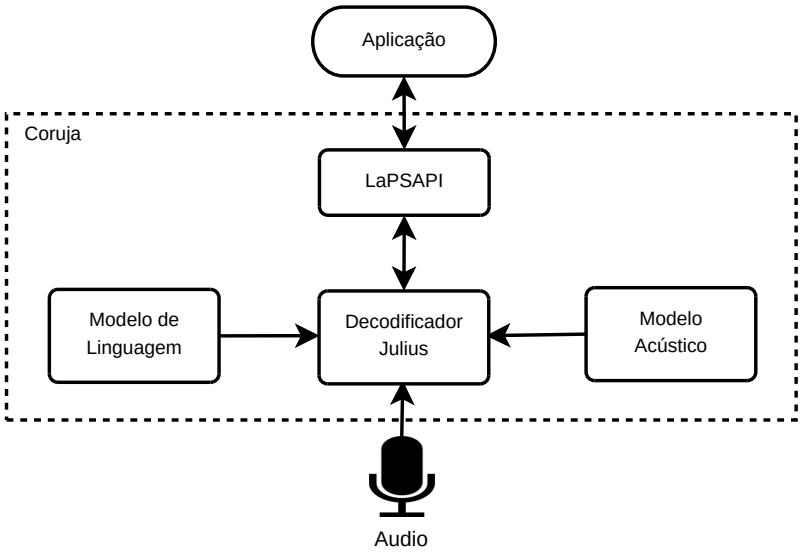


Figura 3.2: Modelo de interação entre uma aplicação e o Coruja.

guagem Java.

Além disso, a LaPSAPI não segue nenhuma das especificações para desenvolvimento de aplicativos com suporte a ASR consagradas na comunidade, o que significa que o código fonte escrito para usar a LaPSAPI, e consequentemente o Coruja, não pode ser utilizado para controlar outro *engine* ASR, implicando em re-trabalho para o desenvolvedor caso uma troca de *engine* seja necessária ao longo de um projeto.

3.3 JLaPSAPI: Uma API em Java para o Coruja

Com base no exposto, este trabalho propôs a criação da JLaPSAPI, uma API inteiramente em Java e compatível com a especificação JSAPI para o Coruja, visando não só a disponibilização desse *engine* na plataforma de programação Java, como também sua adequação a uma especificação largamente utilizada na comunidade.

De um ponto de vista arquitetural, a JLaPSAPI opera sobre a LaPSAPI para controlar o decodificador Julius, evitando assim a re-implementação de funcionalidades básicas já implementadas na atual versão do Coruja. Devido a API proposta ter sido implementada em Java e a LaPSAPI ter sido escrita em C/C++, houve a necessidade de se utilizar um *wrapper* para prover a comunicação entre essas duas APIs. Essa comunicação foi provida pela Java Native Interface (JNI) [Liang 1999].

Nessa nova arquitetura, o acesso ao Coruja é feito através de código JSAPI. Como mostrado na Figura 3.3, o programador tem agora, a possibilidade de alternar entre o Coruja e qualquer outro *engine* que siga a especificação JSAPI, como o Sphinx-4 por exemplo, sem a necessidade de alterar o código da sua aplicação Java.

Atualmente, a JLaPSAPI conta com um conjunto reduzido de métodos e eventos (vide

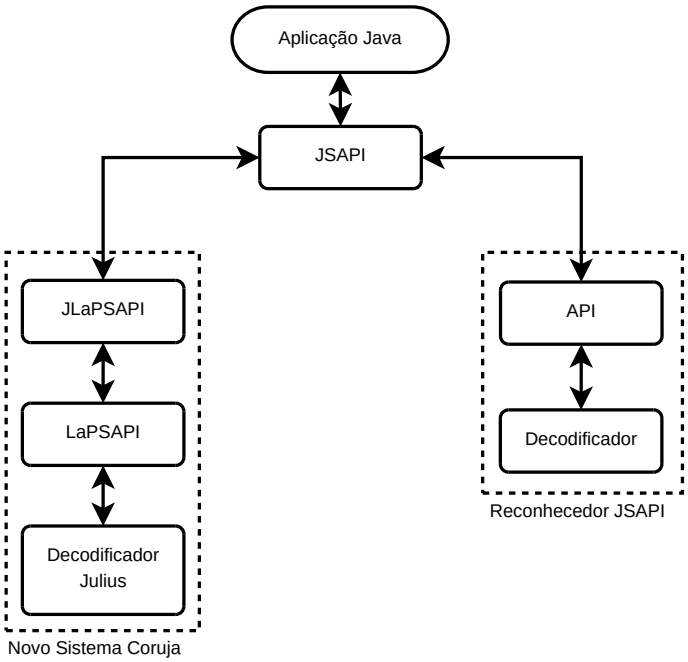


Figura 3.3: Modelo de interação entre uma aplicação Java e o Coruja através da especificação JSAPI.

Tabela 3.1). Entretanto, deve-se notar que esses recursos são suficientes para a construção de aplicativos para *desktop* com suporte a ASR em PB. Para uma melhor compreensão das funcionalidades providas pela JLaPSAPI, a seguir são descritos seus métodos e eventos.

Métodos e Eventos	**Descrição Básica**
createRecognizer	Cria uma instância do *engine*
allocate	Aloca os recursos do *engine*
deallocate	Desaloca os recursos do *engine*
loadJSGF	Carrega uma gramática de um arquivo
setEnabled	Ativa ou desativa reconhecimento de uma gramática
addResultListener	Escolhe classe ouvinte do resultado por *engine* ou por gramática
resume	Inicia o reconhecimento do *engine*
pause	Pausa o reconhecimento do *engine*
resultAccepted	Recebe o resultado do reconhecimento

Tabela 3.1: Métodos e eventos suportados pela JLaPSAPI.

O método *createRecognizer* cria uma instância da classe *Recognizer* especificada pela JSAPI. Essa classe possibilita o controle dos aspectos (p.e, alterar propriedades e chamar métodos) do decodificador utilizado, no caso do Coruja, o Julius. Já o método *allocate* aloca os recursos do decodificador Julius. Similarmente, o método *deallocate* desaloca esses recursos.

Através do método *loadJSGF* é possível carregar uma gramática livre de contexto[1] no formato especificado pela Java Speech Grammar Format (JSGF) [JSG 2012]. Para que isso seja possível, um conversor gramatical do formato da JSGF para o formato suportado pelo decodificador Julius foi construído. É importante salientar que a atual versão do conversor ainda não suporta regras gramaticais recursivas, facilidade suportada pelo decodificador Julius. A gramática carregada pode ser habilitada ou desabilitada com o método *setEnabled*.

O método *resume* abre o *stream* de áudio e inicia o reconhecimento e o método *pause* provoca o fechamento do *stream* de áudio e pausa o reconhecimento. A captura do resultado do reconhecimento é realizado por uma instância da classe *ResultAdapter*, que é anexada a aplicação por meio do método *addResultListener*. O resultado do reconhecimento é recebido pelo método *resultAccepted*. No Anexo A, é apresentando um exemplo de uso do Coruja a partir da JSAPI.

3.4 SpeechOO: ditado no LibreOffice

A fim de validar na prática a nova versão do Coruja disponibilizada neste trabalho e demonstrar suas vantagens, a mesma foi utilizada na refatoração do projeto livre, SpeechOO [SpeechOO 2010].

O SpeechOO é uma extensão para o programa Writer do pacote de ferramentas de escritório LibreOffice, que permite a utilização deste a partir da interface de voz. Esse projeto vem sendo desenvolvido pelo Laboratório de Processamento de Sinais da UFPA em parceria com o Centro de Competência em Software Livre da Universidade de São Paulo desde setembro de 2010. Em sua versão atual, o SpeechOO possui apenas a funcionalidade de ditado, que reconhece a fala do usuário e adiciona o texto reconhecido ao corpo do documento no Writer.

Desenvolvido na linguagem Java, o projeto inicial do SpeechOO teve suas funcionalidades de ASR desenvolvidas com base na versão do Coruja que ainda não oferecia suporte a referida linguagem. Isso era possível através da utilização de um *wrapper* para prover a comunicação entre o código em Java da aplicação e o código em C/C++ da API do Coruja, como pode ser visto na Figura 3.4.

Deve-se ressaltar que devido a utilização de um *wrapper*, o projeto do SpeechOO recebeu uma camada adicional de implementação; código extra, não relacionado a lógica da aplicação em si. Dessa forma, o projeto do SpeechOO ganhou complexidade, o que dificultava tanto a continuidade do seu desenvolvimento quanto sua manutenção.

Em consequência da utilização de um *wrapper*, deve-se notar que, a camada de implementação do projeto do SpeechOO responsável por controlar as funcionalidades de ASR providas pelo Coruja se tornou mais complexa, devido ao código extra adicionado a essa camada referente ao uso do *wrapper*, recurso indispensável até então em virtude

[1] A gramática livre de contexto atua como o modelo de linguagem, provendo ao reconhecedor as palavras e frases que podem ser ditas

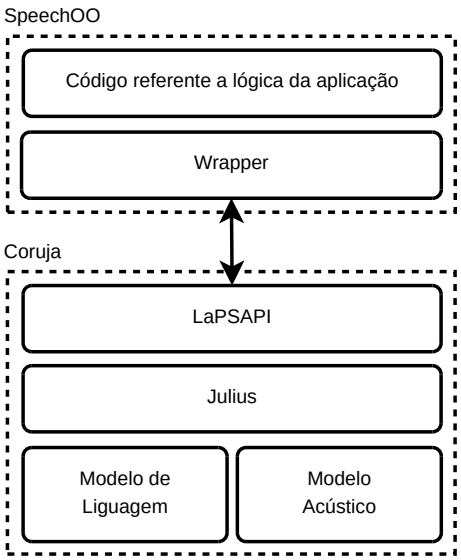


Figura 3.4: Arquitetura do SpeechOO baseada na versão do Coruja sem suporte a JSAPI.

da limitação imposta pelo Coruja quanto a linguagem de programação Java, dificultando assim o desenvolvimento e a manutenção do projeto do SpeechOO.

Nesse contexto, parte do presente trabalho foi dedicada a refatoração do SpeechOO. Com base nos recursos aqui desenvolvidos, uma nova arquitetura foi proposta para o *software* (Figura 3.5), onde sua camada de implementação referente ao código do *wrapper* foi totalmente substituída por código JSAPI. Com o controle das funcionalidade de ASR do Coruja sendo feito a partir de código da especificação, o projeto do SpeechOO não só se tornou mais limpo livre do código do *wrapper*, como também se tornou independente do *engine* ASR utilizado, permitindo que o Coruja seja substituído por outro *engine* ASR, tanto para PB quanto para qualquer outra língua, o que facilitará uma já prevista internacionalização do SpeechOO.

3.5 Conclusão

Este capítulo apresentou a JLaPSAPI, uma API que possibilita o desenvolvimento de aplicativos para *desktop* em Java com suporte a ASR em PB usando o *engine* Coruja e de forma indireta o decodificador Julius. A adição dessa API ao pacote padrão do Coruja, possibilita a utilização deste a partir da poderosa linguagem de programação Java e o adequa a especificação JSAPI, uma das especificações mais utilizadas para o desenvolvimento de aplicativos com suporte a ASR. Com a JLaPSAPI, desenvolvedores podem agora ter acesso as funcionalidades providas pelo Coruja através de código da especificação JSAPI, tornando suas aplicações independentes do *engine* de reconhecimento de voz utilizado. Como prova de conceito, o projeto SpeechOO foi refatorado de forma a utilizar a nova versão do Coruja aqui disponibilizada [JLaPSAPI 2012].

A contribuição desenvolvida nessa etapa do trabalho aumentou a abrangência do Co-

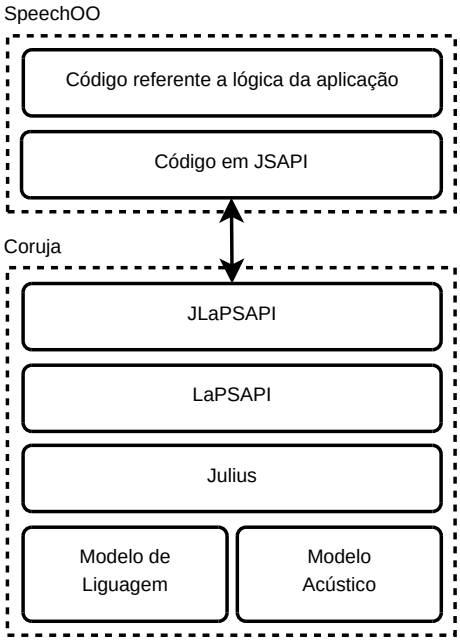


Figura 3.5: Arquitetura proposta baseada na versão do Coruja com suporte a JSAPI.

ruja, tornando-o um recurso mais rico para o desenvolvimento de aplicativos com suporte a ASR em PB. Entretanto, sua utilização continua restrita ao desenvolvimento de aplicativos para *desktop*, devido a uma limitação do decodificador Julius, que não oferece suporte a plataformas móveis. O capítulo seguinte descreve o trabalho desenvolvido para tornar possível o desenvolvimento de aplicativos com suporte a ASR em PB para dispositivos móveis, utilizando recursos livres providos pelo grupo FalaBrasil aliados a ferramentas disponíveis à comunidade.

Capítulo 4

Modelos acústicos para o Sphinx

Desde sua criação, o grupo FalaBrasil disponibiliza recursos para ASR em PB baseados no HTK, um poderoso pacote de ferramentas para treinamento de modelos acústicos que vem sendo utilizado em larga escala por vários outros grupos de pesquisa para o desenvolvimento de trabalhos na área de ASR para diversas línguas. Infelizmente para a comunidade, os decodificadores disponibilizados com o pacote do HTK não podem ser livremente distribuídos e as opções livres, como o Julius, ainda não alcançaram a mesma performance obtida pelos decodificadores do HTK. Devido ao número escasso de decodificadores disponíveis, é difícil desenvolver recursos para ASR com o HTK que possam ser utilizados em diversas plataformas e linguagens de programação.

Um dos objetivos deste trabalho foi disponibilizar recursos para ASR em PB para dispositivos móveis. Devido as limitações do HTK e do Julius descritas anteriormente, neste trabalho foi iniciado um estudo sobre outro pacote de ferramentas para treinamento de modelos acústicos que provê maior versatilidade quanto a plataforma de desenvolvimento, o Sphinx.

Este capítulo apresenta os esforços investidos visando o desenvolvimento de recursos para ASR em PB utilizando as ferramentas do projeto Sphinx. São descritos os *baselines* de treinamento e conversão usados neste trabalho para construir modelos acústicos no formato dos decodificadores disponibilizados pelo grupo CMUSphinx, com o objetivo de prover recursos para ASR em PB tanto para *desktop* quanto para sistemas embarcados. Primeiramente, são descritos os recursos utilizados durante o treinamento como: corpora de voz e dicionário fonético, bem como o treinamento em si. Em seguida, é apresentando o *software* disponibilizado pelo grupo CMUSphinx usado para conversão de modelos acústicos do formato do HTK para o Sphinx, aqui chamado de HTK2Sphinx. São relatadas as limitações e dificuldades de utilização do *software* e por fim, é apresentada a estratégia criada para utilização desse recurso.

4.1 Corpora de voz e dicionário fonético

Treinar modelos acústicos é uma tarefa demorada que envolve várias etapas. A coleta de dados de treino é uma delas. Neste trabalho, bases de voz transcritas livres e proprietárias, foram utilizadas possibilitando que um maior esforço fosse dedicado a etapa de treinamento propriamente dita, em detrimento a etapa preliminar de coleta de dados. A seguir, as bases de voz que compuseram os dados de treino e o dicionário fonético utilizado para o treinamento dos modelos acústicos são descritos.

4.1.1 O corpus West Point

O West Point Brazilian Portuguese Speech é um corpus de áudio para PB criado pelo governo dos EUA com intuito de desenvolver modelos acústicos para sistemas de reconhecimento de voz. O corpus é distribuído no catálogo da LDC (LDC2008S04) e consiste de sentenças lidas por 60 mulheres e 68 homens, nativos e não-nativos. As sentenças gravadas resumem-se a 296 frases e expressões. O corpus West Point original possui algumas restrições, como ausência de transcrições fonéticas e ortográficas, além da existência de arquivos de áudio com falhas, como ruídos e fala não clara. Assim, apenas um sub-conjunto da base, sugerido em [Santos et al. 2010] foi utilizado para o treinamento dos modelos acústicos. No total, 7.920 arquivos com locutores nativos foram usados, correspondendo a 8 horas de áudio, amostrados em 16.000 Hz com 16 bits por amostra.

4.1.2 O corpus LapsStory

A LapsStory, desenvolvida em [Neto et al. 2010], é uma base de áudio para PB composto por arquivos extraídos de *audiobooks*, manualmente segmentados em arquivos menores com aproximadamente 30 segundos cada, amostrados em 16.000 Hz e quantizados em 16 bits. O corpus LapsStory é composto por 8 locutores sendo 5 do sexo masculino e 3 do feminino totalizando 16 horas e 17 minutos de áudio. Devido ao fato de alguns dos *audiobooks* serem protegidos por direitos autorais, apenas parte da LapsStory é distribuída publicamente [FalaBrasil 2012].

4.1.3 O corpus do CETUC

O Centro de Estudos em Telecomunicações (CETUC) [CETUC 2012], através do Professor Doutor Abraham Alcaim, gentilmente cedeu ao LaPS, para fins de pesquisa exclusivamente, seu corpus de áudio para PB. Esse corpus, é composto por áudios de 1.000 sentenças, gravados por 101 locutores, totalizando aproximadamente 143 horas de áudio. A Tabela 4.1 sumariza os dados das bases de áudio descritas.

Corpus	Horas	Locutores	Palavras	Ambiente acústico
West Point	8	128	484	não controlado
LapsStory	16.28	8	8257	controlado
CETUC	142.83	101	3528	não controlado

Tabela 4.1: Bases de áudio utilizadas para o treinamento dos modelos acústicos para o Sphinx.

4.1.4 Dicionário fonético

O dicionário fonético utilizado neste trabalho foi o UFPAdic. Esse dicionário, publicado por [Siravenha et al. 2008], é formado por 65.532 palavras selecionadas entre as mais frequentes no corpus CETENFolha [CETENFolha 2012], que é um corpus composto por textos extraídos do jornal Folha de S. Paulo e compilado pelo Núcleo de Linguística Computacional da Universidade de São Paulo, Brasil. Devido a restrições do *software* para treinamento de modelos acústicos disponibilizados pelo grupo CMUSphinx, o alfabeto fonético original do UFPAdic foi reformulado para que não contivesse caracteres especiais e letras maiúsculas. No Anexo B a versão modificada do alfabeto fonético pode ser encontrada. Um conversor foi construído para auxiliar na tarefa de re-formatação do UFPADic. Esse recurso encontra-se disponível na página do projeto FalaBrasil [FalaBrasil 2012]. A seguir, o processo de treinamento dos modelos acústicos gerados neste trabalho é descrito.

4.2 Treinamento de modelos acústicos para o Sphinx

Parte dos modelos acústicos desenvolvidos neste trabalho foram construídos utilizando as ferramentas para treinamento de modelos acústicos do pacote SphinxTrain. Esse pacote, é composto por ferramentas escritas na linguagem C, que implementam os algoritmos de treinamento, e por *scripts* em Perl que auxiliam na utilização dessas ferramentas. Esses *scripts* Perl, fornecem uma receita padrão para o treinamento de modelos acústicos, e permitem a fácil adaptação dessa receita a partir da alteração dos parâmetros de treinamento via arquivo de configuração (sphinx_train.cfg).

Assim, desenvolvedores que desejam utilizar o pacote SphinxTrain para o treinamento de modelos acústicos, tem a sua disposição duas possíveis abordagens. Fazer uso das ferramentas implementas em C diretamente, ou utilizá-las de maneira indireta através da receita padrão, oferecida pelos *scripts* Perl.

Ambas abordagens possuem seus prós e contras. A utilização dos *scripts* Perl permite que desenvolvedores sem experiência com o ambiente de treinamento oferecido pela CMUSphinx alcancem resultados de forma mais rápida, abstraindo detalhes de configuração das ferramentas de treinamento que tipicamente dificultam esse contato inicial com a tarefa de treinar modelos acústicos. Entretanto, esse mascaramento do processo pode inviabilizar pesquisas mais aprofundadas, fazendo com que nesses casos, o uso direto das ferramentas em C seja uma alternativa mais interessante. Por se tratar de um primeiro

esforço para o desenvolvimento de recursos para PB com o SphinxTrain, a abordagem baseada nos *scripts* Perl foi adotada neste trabalho.

4.2.1 Preparação dos dados

Como descrito no Capítulo 2, o primeiro passo para o treinamento de modelos acústicos é a extração de parâmetros do sinal de voz. A abordagem MFCC foi utilizada para a parametrização do corpus de treino. Mais especificamente, a configuração de *frontend* utilizada consiste dos 12 coeficientes MFCC [Davis e Merlmestein 1980] mais a componente de energia C0, computados a cada 10 mili-segundos para um *frame* de 25 mili-segundos. Tanto para modelos contínuos quanto semi-contínuos, esses 12 coeficientes são complementados por um conjunto de coeficientes dinâmicos, calculados em tempo de execução durante o processo de treinamento e decodificação. Adicionalmente, uma normalização pela média *cepstral* [Young et al. 2006] também é realizada. Juntos, os coeficientes citados formam o vetor de *features*. Os arquivos de configuração utilizados para a extração de parâmetros neste trabalho estão presentes no Anexo C.

O tipo de *feature* utilizada no Sphinx é configurado pelo parâmetro *feat* das ferramentas de treinamento e decodificação. Esse parâmetro, define os coeficientes dinâmicos que devem ser calculados e a disposição dos mesmos no vetor de *features*. Para os modelos contínuos, foi utilizada a *feature 1s_c_d_dd*, formada por um vetor composto pelos 12 coeficientes MFCC mais a componente de energia C0, seguido por sua primeira e segunda derivada nessa respectiva ordem. Totalizando 39 coeficientes.

Já para os modelos semi-contínuos, a *feature* usada foi a s2_4x, composta por 4 vetores de parâmetros. O primeiro deles consiste nos 12 coeficientes MFCC; o segundo vetor é composto por 24 elementos, onde os 12 primeiros são a primeira derivada (*short term*[1]) dos coeficientes MFCC, dada por $\delta_s(t) = \vec{C}(t+2) - \vec{C}(t-2)$, onde $\vec{C}$ é o vetor de coeficientes MFCC, e os 12 restantes são a primeira derivada (*long term*) dos coeficientes, definida por $\delta_l(t) = \vec{C}(t+4) - \vec{C}(t-4)$; o terceiro vetor possui 3 elementos, a componente de energia C0, sua primeira derivada (*short term*) e a sua segunda derivada; o último vetor é dado pela segunda derivada dos 12 coeficientes MFCC. Totalizando 51 coeficientes.

4.2.2 Estimando modelos independentes de contexto

Tipicamente, a estratégia utilizada para a criação de modelos acústicos é partir de modelos mais simples e refinar de maneira gradual esses modelos a fim de torná-los mais robustos [Woodland e Young 1993]. Com base nessa abordagem, inicialmente foram treinados modelos independentes de contexto (CI) para 39 monofones, sendo 38 fonemas e 1 modelo para o silêncio. A topologia com 3 estados emissores, *left-to-right* sem *skip transitions* e com *self-loops* foi utilizada para todas as HMMs treinadas.

[1] Os termos *short* e *long term* foram retirados de comentários presentes no código fonte do Sphinx

Para os modelos contínuos, a abordagem *flat-start* [Young et al. 2006] foi adotada para inicialização da média e variância da Gaussiana de cada estado em cada HMM com o valor da média e variância global dos dados de treino. Para os modelos semi-contínuos, foram utilizados 4 *codebooks* com 256 *codewords* cada. As *codewords* foram encontradas usando o algoritmo de clusterização *k-means*. O peso das misturas e probabilidades de transição de cada estado em cada HMM foram inicializadas equiprováveis.

Os modelos CI foram então treinados utilizando múltiplas iterações do algoritmo de Baum-Welch [Welch 2003]. Os parâmetros das HMMs foram re-estimados até que a diferença entre a *likelihood* da iteração atual e da iteração anterior fosse menor ou igual ao limiar definido (0.1) ou até que o número máximo de iterações (10) fosse atingido.

4.2.3 Estimando modelos dependentes de contexto

Para o treinamento dos modelos dependentes de contexto (CD), uma lista contendo todos os possíveis trifones foi automaticamente gerada com base no dicionário fonético utilizado. Em seguida, os trifones presentes na lista não representados nos dados de treino foram excluídos do treinamento. Os modelos CD foram então criados, inicializados e re-estimados, esse último segundo os mesmos critérios utilizados para os modelos CI.

4.2.4 Vinculando estados

Após o treinamento inicial dos modelos CD, os estados de todos os trifones (vistos e não vistos durante o treinamento) foram vinculados com base em uma árvore de decisão, construída a partir de um conjunto de questões linguísticas geradas automaticamente por um algoritmo [Singh et al. 1999] *data-driven*, disponibilizado no pacote SphinxTrain, visando criar HMMs mais robustas através do compartilhamento de dados, diminuindo os efeitos negativos causados pela insuficiência de dados para o treinamento de alguns dos trifones. Antes que os estados fossem de fato vinculados, a árvore de decisão passa por um processo de *pruning* até que os número de estados vinculados definido no arquivo de configuração do treinamento fosse atingido.

Os modelos CD com os estados vinculados foram re-estimados com o algoritmo de Baum-Welch até a convergência. Se ao final da re-estimação o número de Gaussianas por mistura escolhido para o modelo ainda não tivesse sido atingido, cada Gaussiana era então divida em duas e então os parâmetros do modelo eram re-estimados. Esse processo se repetia até que número de Gaussianas por mistura especificado via arquivo de configuração fosse alcançado. A Figura 4.1 mostra o diagrama do *baseline* de treinamento dos modelos acústicos criados nesse trabalho.

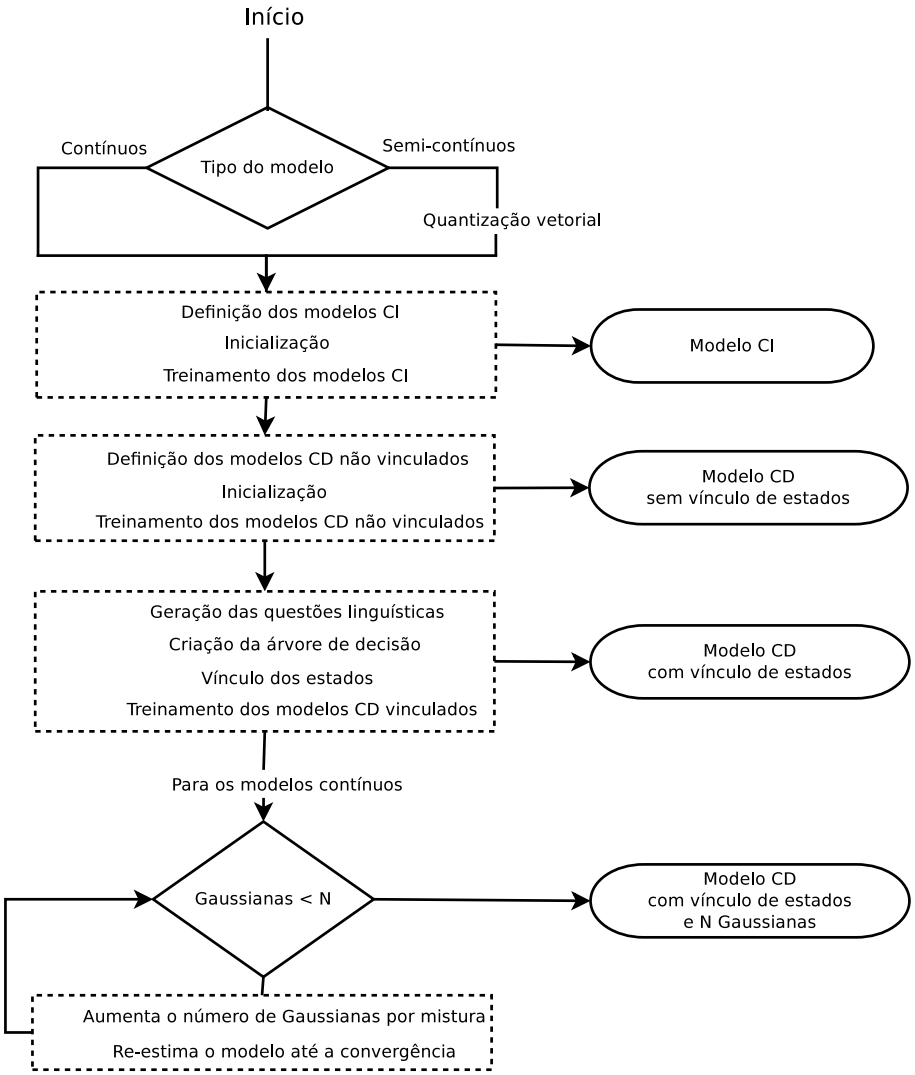


Figura 4.1: Fluxograma do *Baseline* seguido para a criação dos modelos acústicos.

4.3 Conversão de modelos acústicos do HTK para o Sphinx

Treinar modelos acústicos é uma tarefa complexa. Felizmente, hoje esse trabalho é facilitado graças a ferramentas como o SphinxTrain e o HTK. De certa forma, conseguir chegar a um modelo acústico de boa performance exige, além de recursos de qualidade, conhecimento sobre a ferramenta utilizada para construir esse modelo. Adquirir tal maturidade exige um grande investimento de tempo.

Diante do crescimento do projeto da CMUSphinx, que hoje disponibiliza decodificadores tanto para *Desktop* quanto para plataformas móveis, é compreensível que grupos de pesquisa iniciem esforços com a finalidade de disponibilizar modelos acústicos que possam ser utilizados por esses decodificadores. Devido ao alto investimento que uma migração entre ferramentas de treinamento pode ocasionar, é natural que esses grupos de pesquisa busquem formas alternativas de desenvolver modelos acústicos compatíveis com

os decodificadores do grupo CMUSphinx.

Como dito anteriormente, até a realização deste trabalho todos os recursos disponibilizados pelo grupo FalaBrasil se baseavam no pacote HTK. Dada essa realidade, e acreditando que outros grupos de pesquisa pudessem compartilhar da mesma, parte desse trabalho se dedicou ao estudo da possibilidade de conversão de modelos acústicos do formato do HTK para o Sphinx.

Durante a pesquisa, apenas um *software* com essa proposta foi encontrado, disponibilizado pelo grupo CMUSphinx. Entretanto, não foi encontrada qualquer avaliação sobre o mesmo, nem sequer relatos de desenvolvedores que tivessem conseguido utilizar a referida ferramenta. Nesse contexto, parte deste trabalho foi dedicada a prover tal avaliação, além de um relato sobre a utilização dessa ferramenta.

4.3.1 Conversor de modelos acústicos da CMUSphinx

O *software* HTK2Sphinx [HTK2Sphinx 2012] é um pacote de *scripts* escritos na linguagem de programação Python que converte modelos acústicos do formato do HTK para o formato suportado pelo decodificador Sphinx3. Desenvolvido por W.J. Maaskant em 2007, o conversor só foi publicado na página do projeto CMUSphinx em 2010. Uma nota do autor, presente no pacote de distribuição da ferramenta, deixa claro que o conversor não é genérico o suficiente para que qualquer modelo do HTK possa ser convertido pelo mesmo, pelo contrário, apenas um conjunto específico de modelos pode ser convertido pela ferramenta. Algumas das limitações do conversor são causadas por diferenças entre os *softwares* HTK e Sphinx, entretanto, a maioria delas se deve a limitações de implementação. Dentre as limitações relacionadas ao uso do conversor, as principais são:

- o vetor de *features* utilizado para treinar o modelo que será submetido a conversão deve ser compostos por 39 parâmetros;
- todas as HMMs do modelo devem ter uma matriz de transição como a representada na Tabela 4.2. Os asteriscos representam probabilidades maiores ou iguais a zero. O número de estados pode se variado, desde que o primeiro se mantenha não emissor;

	1	2	3	4	5
1	0	1	0	0	0
2	0	*	*	*	*
3	0	*	*	*	*
4	0	*	*	*	*
5	0	0	0	0	0

Tabela 4.2: Estrutura da matriz de transição dos modelos para conversão

- todos os estados devem ter o mesmo número de Gaussianas por mistura;

- as definições das HMMs dos monofones base de cada trifone devem existir no arquivo de definição do modelo e;
- o conversor não oferece suporte a bifones.

Em consequência dessas limitações, os modelos treinados com o HTK para posterior conversão, precisam seguir um *baseline* de treinamento particular visando atender a todos os requisitos exigidos. Nas seções seguintes, o *baseline* utilizado para o treinamento do modelo acústico criado para conversão é descrito.

4.3.2 Compatibilizando os *frontends* de treino e teste

Um requisito básico para se obter sucesso em uma tarefa de ASR é garantir que a mesma configuração de *frontend* seja utilizada nas etapas de treinamento e decodificação. Atender a esse requisito enquanto trabalhando com a mesma ferramenta de extração de parâmetros nas duas etapas mencionadas é uma tarefa simples, entretanto, quando se trabalha com ferramentas diferentes, tornar essas configurações equivalentes pode exigir certo esforço.

Nos *softwares* HTK e o Sphinx, a etapa de *frontend* (extração de parâmetros e a montagem da *feature*) é executada de formas diferentes. No HTK, a ferramenta HCopy é utilizada para realizar a extração de coeficientes e a computação dos coeficientes dinâmicos antes que o treinamento ou a decodificação sejam iniciados. Dessa forma, os arquivos gerados pelo HCopy com a parametrização dos sinais de voz de entrada carregam todos os dados que formam a *feature* escolhida pelo desenvolvedor, não havendo então necessidade de que qualquer processamento adicional seja realizado nas etapas que seguem a de aplicação do *frontend*. Já no caso do Sphinx, a ferramenta sphinx_fe realiza apenas a extração de coeficientes, deixando a computação dos coeficientes dinâmicos para ser realizada em tempo de execução, durante as etapas de treinamento e decodificação.

Tendo em vista a necessidade de se trabalhar tanto no HTK quanto no Sphinx com configurações equivalentes de *frontend* e levando em consideração a necessidade de se trabalhar com uma *feature* composta por 39 parâmetros, um dos requisitos do conversor, três abordagens foram propostas como solução para esse problema.

A primeira abordagem estudada para fazer com que as bases de treino e teste fossem parametrizadas de formas equivalentes, consistiu em analisar as ferramentas HCopy e sphinx_fe a fim de fazer com que ambas produzissem resultados equivalentes. Para isso, um estudo sobre a implementação dessas ferramentas foi conduzido com o propósito de encontrar uma correspondência entre seus parâmetros de configuração. Em resumo, o objetivo aqui foi encontrar o conjunto de parâmetros e valores no sphinx_fe que correspondesse a um conjunto específico de parâmetros usado no HCopy. Nesse ponto do trabalho, a documentação encontrada sobre a ferramenta sphinx_fe não foi suficiente para deixar claro alguns aspectos sobre a implementação do *frontend* do Sphinx. Como o sphinx_fe é uma ferramenta de código aberto, muita informação pôde ser obtida a partir da leitura do

seu código fonte, entretanto, devido essa ser uma tarefa que demanda muito tempo, esse estudo não foi concluído.

A fim de possibilitar que outros grupos de pesquisa possam concluir o trabalho que foi iniciado, a Tabela 4.3 apresenta os parâmetros que foram mapeados durante a realização deste trabalho.

sphinx_fe	valor	HCopy	valor
alpha	0.97	preemcoef	0.97
nfilt	26	numchans	26
dither	no	adddither	0.0
ncep	13	numceps	12
lowerf	1	lofreq	1
upperf	8000	hifreq	8000
wlen	0.0250	windowsize	250000.0
samprate	16000	sourcerate	625
lifter	22	ceplifter	22
frate	100	targetrate	100000.0
remove_dc	yes	zmeansource	T

Tabela 4.3: Parâmetros equivalentes nas ferramentas sphinx_fe e HCopy

Como prova de conceito, abaixo dois vetores de parâmetros são mostrados, cada um contendo 13 valores referentes aos coeficientes *cepstrais* extraídos dos primeiros 25 milisegundos do arquivo art001a.wav do corpus Constituição [Neto et al. 2010] amostrado em 16.000 Hz. O primeiro deles foi extraído usando o HCopy e o segundo o sphinx_fe. As configurações usadas nas duas ferramentas podem ser visualizadas no Anexo G.

```
            [C0]    [C1]    [C2]  [C3]   [C4]   [C5]   [C6]   [C7]  [C8]  [C9]  [C10]  [C11] [C12]

HCopy:    -37.416 -14.512 -5.265 3.112 -2.152 -6.080 -5.954 -0.983 2.185 8.546 -2.491 -2.599 42.078

sphinx_fe: -28.592 -14.952 -3.692 2.191 -1.503 -6.418 -5.230 -1.411 2.570 8.451 -1.180 -0.619 3.144
```

A segunda abordagem proposta consistiu em utilizar o HCopy no lugar do sphinx_fe para parametrizar a base de teste e assim garantir que as configurações de *frontend* utilizadas durante o treinamento e a decodificação fossem equivalentes. Para isso, um conversor de formato foi criado para converter os arquivos parametrizados no formato gerado pelo HCopy para o do sphinx_fe. De posse dos arquivos convertidos, para a utilização dos mesmos nos decodificadores da CMUSphinx, os parâmetros *feat* e *ncep* devem receber os respectivos valores *1s_c* e 39. Essa configuração informa ao decodificador que a *feature* no arquivo de parâmetros está disposta em um vetor de tamanho 39 e que nenhum coeficiente dinâmico deve ser calculado. Dessa forma, o decodificador pode trabalhar com as *features* extraídas pelo HCopy, desde que nenhuma normalização seja realizada em tempo de execução.

Deve-se notar, que essa abordagem é solução apenas quando o modo de decodificação utilizado é o *offline*, visto que, na decodificação *online*, a extração de parâmetros é

realizada em tempo de execução, usando a mesma implementação do sphinx_fe. Logo, não seria possível fazer uso do modelo acústico convertido nesse modo de decodificação. Em virtude dessa limitação, uma terceira abordagem foi proposta visando permitir que o modelo acústico convertido pudesse ser utilizado também no modo de decodificação *online*.

A terceira abordagem seguiu uma linha de raciocínio inversa a que foi proposta na segunda. Nessa abordagem, a ferramenta sphinx_fe foi utilizada para parametrizar a base de treino. Para isso, além de um conversor de formato de arquivos parametrizados no sphinx_fe para o do HCopy, foi necessário isolar parte do código do decodificador Sphinx3 para que fosse possível encontrar os valores dos coeficientes dinâmicos que compõe a *feature*, que no Sphinx só são computados em tempo de execução. A *feature* utilizada foi a *1s_c_d_dd* com 39 parâmetros. A Tabela 4.4 mostra a configuração de *frontend* utilizada. Para que os arquivos convertidos pudessem ser utilizados no HTK, o *targetkind MFCC_E_D_A_Z* foi informado em seus cabeçalhos, que corresponde a uma *feature* com a mesma estrutura da que foi utilizada para a extração dos parâmetros com o sphinx_fe. Tendo garantido uma forma de trabalhar nas fases de treinamento e decodificação com configurações de *frontend* equivalentes, o passo seguinte foi o de treinamento do modelo acústico para ser convertido.

Parâmetro	Valor
alpha	0.97
dither	yes
doublebw	no
nfilt	40
ncep	13
lowerf	133.33334
upperf	6855.4976
nfft	512
wlen	0.0256
transform	legacy
feat	1s_c_d_dd
agc	none
cmn	current
varnorm	no

Tabela 4.4: Configuração de *frontend* utilizada para parametrizar a base de treino com o sphinx_fe.

4.3.3 Treinando um modelo acústico para o HTK

Tipicamente, o *baseline* adotado para o treinamento de modelos acústicos para o HTK [Young et al. 2006] segue de perto o tutorial de treinamento descrito no HTKBook. Contudo, ao treinar modelos para conversão pela ferramenta htk2s3conv, alguns passos

do referido tutorial não devem ser seguidos a risca. A fim de evidenciar as diferenças entre o *baseline* de treinamento sugerido no HTKBook e o que foi seguido nesse trabalho para criar um modelo apto a conversão, essa seção descreve o treinamento realizado ressaltando os aspectos impares dos *baselines* e também as medidas adotadas para atender aos requisitos do conversor utilizado.

Os corpus descritos na Seção 4.1 compuseram a base de treino utilizada nessa parte do trabalho. Como explicado anteriormente, a etapa de *frontend* referente ao treinamento do modelo acústico para conversão se deu conforme a terceira abordagem descrita. A configuração utilizada foi a mesma descrita na Seção 4.2.1 para modelos contínuos.

Da mesma forma como foi feito no treinamento com o SphinxTrain, o modelo acústico treinado foi refinado iterativamente. A abordagem *flat-start* foi adotada para inicializar 39 modelos monofones (38 fonemas e um modelo de silêncio) com topologia composta por 3 estados emissores, *left-to-right* sem *skip transitions* e com *self-loops*. Iniciando com modelos baseados em monofones e com uma Gaussiana por mistura, as HMMs foram gradualmente expandidas de forma a ter múltiplas Gaussianas por mistura ainda durante o treinamento dos modelos CI. Isso foi necessário devido a duas exigências da ferramenta htk2s3conv: a primeira delas é a necessidade do conversor de conhecer a definição dos modelos monofone usados como fonema base para construção dos trifones presentes no modelo acústico alvo da conversão; a segunda é a necessidade de que todas as HMMs do modelo tenham o mesmo número de Gaussianas por mistura. Após a expansão do número de Gaussianas, as HMMs foram re-estimadas até a convergência pelo algoritmo de Baum-Welch. No tutorial do HTKBook, um modelo adicional para o *short-pause* é construído através da cópia do estado central do modelo de silêncio, com apenas um estado emissor. Devido a topologia da HMM do *short-pause* diferir das demais HMMs do modelo, esse passo não foi seguido para a criação do modelo para conversão.

Para o treinamento dos modelos CD, uma lista contendo todos os trifones possíveis dada a lista de monofones foi gerada. É comum enquanto treinando modelos acústicos para os decodificadores do HTK, que a partir desse ponto apenas as HMMs dos trifones façam parte do arquivo de definição do modelo. Entretanto, como os decodificadores do projeto CMUSphinx precisam da informação dos monofones, suas definições foram mantidas até o modelo final. Para isso, os monofones foram mantidos na lista de HMMs que deveriam ser re-estimadas, mesmo sabendo que esses modelos não seriam mais re-estimados. Após o treinamento dos modelos CD, uma árvore de decisão fonética específica para PB [Neto et al. 2010], foi usada para compartilhar os estados dos trifones com características fonéticas similares. Após o processo de vínculo, o número de componentes por mistura nas distribuições foi gradualmente incrementado, finalizando o processo de treinamento.

4.3.4 Convertendo o modelo acústico treinado

De posse do modelo acústico, prosseguir com a conversão é uma tarefa simples. Para utilizar o conversor basta executar a linha de comando abaixo, onde: o arquivo *hmmdefs*

é o arquivo de definição do modelo acústico treinado pelo HTK; a *hmmlist* consiste na lista de HMMs do modelo e *output* se refere ao diretório onde as definições do modelo convertido para o formato do Sphinx3 devem ser escritas. O modelo convertido pode então ser usado nos decodificadores Sphinx3 e Pocketsphinx, visto que o último oferece suporte a modelos nesse formato.

```
./convert.py hmmdefs hmmlist output
```

4.4 Conclusão

Neste capítulo foram apresentados os *baselines* de treinamento e conversão seguidos para criar modelos acústicos para PB no formato dos decodificadores do grupo CMUSphinx. Com os modelos acústicos construídos é possível criar aplicativos com suporte a ASR em PB tanto para *desktop*, com o Sphinx-4, quanto para dispositivos móveis, com o Pocketsphinx, que oferece suporte a dois dos principais sistemas operacionais móveis do mercado, o Android e iOS. Por não ter havido qualquer entendimento prévio a respeito, os modelos acústicos treinados utilizando a base de áudio da CETUC ainda não estão disponíveis a comunidade. Contudo, modelos acústicos construídos com o restante dos corpus encontram-se disponíveis na página do grupo FalaBrasil. No capítulo seguinte, os modelos acústicos construídos ao longo dessa etapa do trabalho são avaliados e um aplicativo para a plataforma Android 2.2 com suporte a ASR em PB é apresentado como prova de conceito.

Durante o trabalho com a ferramenta HTK2Sphinx, um *bug* que prejudicava a conversão de modelos dependentes de contexto foi identificado e em colaboração com o grupo CMUSphinx, esse *bug* foi corrigido e hoje uma versão atualizada da ferramenta já se encontra disponível.

Capítulo 5

Resultados experimentais

Este capítulo apresenta os resultados experimentais dos modelos acústicos treinados especificamente para os decodificadores do grupo CMUSphinx e os modelos acústicos obtidos através do *baseline* de conversão descrito na Seção 4.3. Primeiramente é descrita a base de áudio LapsMail, que foi construída visando a realização de teste com gramática controlada. Em seguida, são descritos os resultados obtidos nos testes realizados com os modelos acústicos treinados e convertidos. Finalmente, é apresentada uma aplicação para a plataforma Android 2.2 com suporte a ASR em PB.

5.1 O corpus LapsMail

Neste trabalho foi iniciada a construção de um corpus de voz chamado LapsMail. Esse corpus foi projetado para representar um conjunto básico de comandos necessários para controlar uma aplicação de correio eletrônico. A ideia é estabelecê-lo como referência para a avaliação de sistemas ASR para PB dentro desse contexto.

Atualmente, o corpus LapsMail consiste de 86 sentenças, incluindo 43 comandos e 43 nomes falados por 25 voluntários (21 homens e 4 mulheres), o que corresponde a 84 minutos de áudio. Ao todo são observadas 95 palavras distintas. Segue abaixo três sentenças que fazem parte do corpus. A lista completa das sentenças pode ser visualizada no Anexo D.

```
(1) <s> abrir caixa de entrada </s>
(2) <s> responder ao remetente </s>
(3) <s> ana carolina </s>
```

O corpus LapsMail foi gravado usando um microfone de alta qualidade (Shure PG30), amostrado em 16.000 Hz e quantizado em 16 bits. O ambiente acústico não foi controlado, existindo a presença de ruído ambiente nas gravações. A base de áudio LapsMail encontra-se publicamente disponível [FalaBrasil 2012].

5.2 Avaliação dos modelos acústicos para decodificadores Sphinx

Ao todo foram construídos 48 modelos acústicos (42 contínuos e 6 semi-contínuos) utilizando o procedimento descrito na Seção 4.2. Foram treinados modelos acústicos com 500, 1.000, 2.000, 4.000, 6.000 e 8.000 estados vinculados e para cada um desses valores o número de Gaussianas por mistura nos modelos contínuos foi variado de forma a criar modelos com 1, 2, 4, 8, 16, 32 e 64 Gaussianas.

Os modelos acústicos contínuos e semi-contínuos foram testados usando os decodificadores Sphinx3 e Pocketsphinx, respectivamente. Os parâmetros de decodificação de ambos decodificadores não foram variados durante os experimentos. A Tabela 5.1 mostra os valores utilizados para os principais parâmetros de decodificação.

Tabela 5.1: Parâmetros de decodificação para Sphinx3 e Pocketsphinx.

Parâmetro	Sphinx3	Pocketsphinx
Pruning beam width	1e-55	1e-48
Word beam width	1e-35	7e-29
Phone beam width	1e-50	1e-48
Word insertion penalty	0.7	0.65
Language model scale factor	9.5	6.5

Os modelos acústicos construídos nessa etapa do trabalho foram avaliados para as tarefas de ditado e comando e controle. Seus desempenhos foram medidos em termos da taxa de erro por palavras (WER) e do fator de tempo real (xRT). O fator xRT serve como medida de velocidade e é obtido através da divisão do tempo que sistema ASR leva para reconhecer uma frase pela duração da mesma.

Todos os testes foram realizados em um computador com processador Intel Xeon (E5450 3.0 GHz) com 4 GB de RAM. A seguir, são apresentados os resultados da avaliação dos modelos para tarefas de ditado.

5.2.1 Avaliação dos modelos acústicos para ditado

Este experimento buscou medir a viabilidade da construção de um sistema ASR para PB utilizando os modelos acústicos treinados especificamente para os decodificadores do grupo CMUSphinx e estimar seu desempenho em tarefas de ditado. Um modelo de linguagem trigrama e o corpus de referência LapsBenchmark foram utilizados nos testes.

O corpus LapsBenchmark [Neto et al. 2010] é uma base de áudio desenvolvida com o intuito de servir como referência para a avaliação do desempenho de sistemas ASR em ambientes ruidosos. Atualmente, o corpus LapsBenchmark é composto por 35 locutores entre homens e mulheres, com 20 sentenças faladas por cada um deles, o que

corresponde a 54 minutos de áudio. Ao todo são observadas 2.731 palavras distintas. As gravações dessa base foram feitas em ambiente acústico não controlado, usando computadores pessoais e microfones comuns. A taxa de amostragem utilizada foi de 16.000 Hz e cada amostra foi representada com 16 bits. O corpus LapsBenchmark encontra-se publicamente disponível [FalaBrasil 2012].

Já o modelo de linguagem trigrama utilizado foi construído e disponibilizado por [Neto et al. 2010]. Esse modelo foi treinado com 710.000 sentenças extraídas dos corpus CETENFolha e LapsStory [Neto et al. 2010] e possui perplexidade 170, medida com um conjunto de 10.000 sentenças de teste selecionadas randomicamente do corpus CETENFolha (não vistas durante a fase de treinamento do modelo).

Tendo sido mantidos inalterados os parâmetros de decodificação e os demais componentes do sistema ASR, os resultados obtidos nessa etapa do trabalho apontam em particular para o desempenho dos modelos acústicos aqui construídos.

Os resultados dos experimentos mostraram que o aumento do número de estados vinculados resultou em diminuição da WER, tanto para modelos contínuos quanto semicontínuos. Entretanto, a partir de 4.000 estados o desempenho dos modelos se mostrou equivalente. Outra característica observada foi que os modelos com números mais elevados de estados vinculados apresentaram maior fator xRT, o que pode ser justificado pelo fato do número de cálculos de verossimilhança por observação feitos pelo decodificador aumentar de acordo com o número de estados vinculados do modelo acústico.

Como esperado, o aumento do número de Gaussianas por mistura resultou em redução contínua da WER. Ganhos puderam ser registrados até a transição de 16 para 32 Gaussianas. Contudo, valores acima de 16 Gaussianas por mistura implicaram em um aumento significativo do fator xRT. Os gráficos da WER e do fator xRT são mostrados nas Figuras 5.1 e 5.2, respectivamente.

Estipulou-se que os melhores valores de WER devem ser procurados entre os modelos acústicos que obtiveram fator xRT em torno de um. Assim, o modelo acústico com 2.000 estados vinculados e 8 Gaussianas por mistura foi escolhido como um dos melhores avaliados, atingindo uma WER de 16,41%.

A fim de demonstrar a evolução desse modelo acústico a cada iteração do *baseline* de treino modulado na Figura 4.1, as Figuras 5.3 e 5.4 mostram a evolução da WER e do fator xRT ao longo de cada etapa do treinamento, até que a configuração com 2.000 estados vinculados e 8 Gaussianas por mistura fosse atingida.

Como pode ser observado, o desempenho obtido com o modelo acústico baseado em monofones (CI) é muito inferior a do modelo baseado em trifones (CD) com uma Gaussiana por mistura, com uma diferença de WER acima de 50%. No intervalo que compreende os modelos acústicos com uma Gaussiana por mistura até 8, é possível perceber uma queda acentuada da WER sem crescimento considerável do fator xRT. Entretanto, a partir de 16 Gaussianas, a redução da WER é acompanhada de aumentos consideráveis do fator xRT, ratificando 8 Gaussianas por mistura como valor ideal para um modelo com 2.000 estados vinculados.

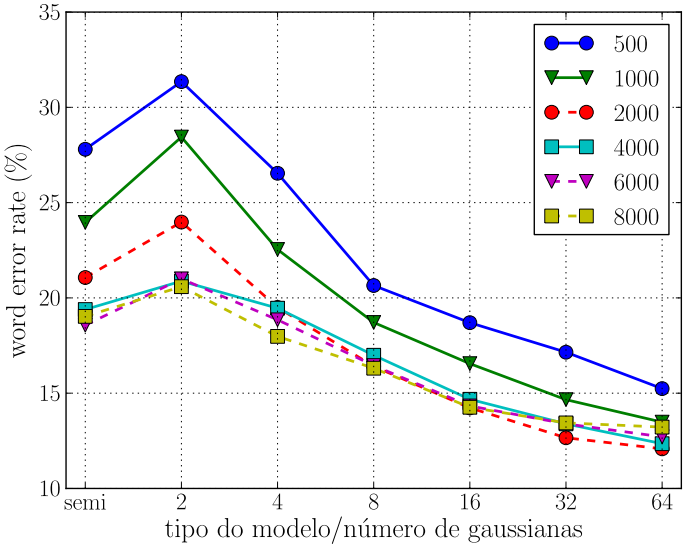


Figura 5.1: WER (%) para a tarefa de ditado usando o corpus LapsBenchmark.

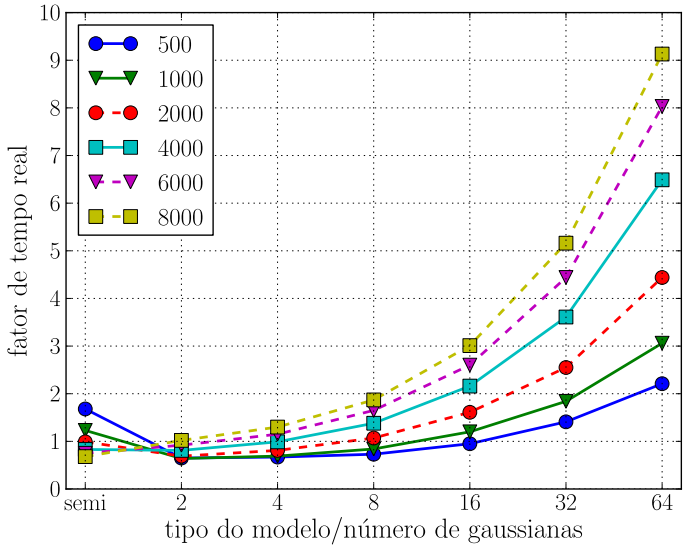


Figura 5.2: Fator xRT para a tarefa de ditado usando o corpus LapsBenchmark.

A seção seguinte apresenta os resultados referentes ao desempenho dos modelos acústicos treinados em uma tarefa de comando e controle.

5.2.2 Avaliação dos modelos acústicos para comando e controle

O objetivo dos experimentos conduzidos nesta etapa do trabalho foi avaliar o desempenho dos modelos acústicos construídos em aplicações de comando e controle. Para

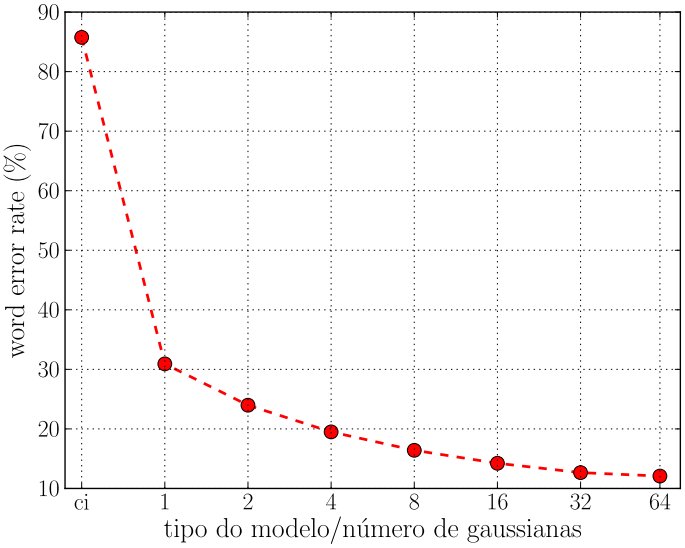


Figura 5.3: Evolução da WER (%) ao longo das etapas de treinamento.

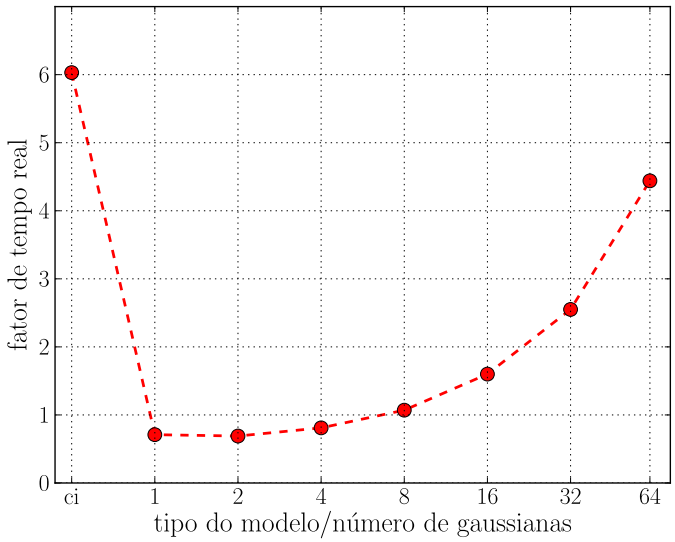


Figura 5.4: Evolução do fator xRT ao longo das etapas de treinamento.

tanto, a base de áudio LapsMail e uma gramática livre de contexto no formato especificado pela JSGF foram utilizadas. Essa gramática pode ser visualizada no Anexo E.

Primeiramente, o modelo acústico com 2.000 estados vinculados e 8 Gaussianas por mistura, selecionado na Seção 5.2.1, foi avaliado. Os resultados obtidos mostraram um desempenho ruim, com uma WER de 6,21%, superior ao que era esperado, levando em consideração o vocabulário reduzido do corpus LapsMail (apenas 95 palavras).

A partir da análise dos resultados utilizando todos os 48 modelos acústicos treinados,

foi constatado que 165 dos 185 erros contabilizados foram causados por falhas no reconhecimento das sentenças referentes aos nomes próprios presentes no corpus LapsMail. A complexidade associada a tarefa de reconhecer nomes próprios foi discutida em [Sethy et al. 2002, Maskey et al. 2004], onde abordagens foram propostas para melhorar o desempenho de sistemas ASR para essa tarefa específica. No entanto, a tarefa de reconhecer nomes próprios não foi estudada neste trabalho.

Dessa forma, um novo experimento foi conduzido onde as sentenças referentes aos nomes próprios presentes no corpus LapsMail não foram incluídas. As 43 sentenças restantes referentes a comandos, totalizando 47 minutos de áudio, foram utilizadas para avaliar o desempenho dos modelos acústicos. A Figura 5.5) ilustra os resultados encontrados.

De forma similar ao que foi observado para a tarefa de ditado, os valores mais baixos de WER foram obtidos a medida que o número de Gaussianas por mistura foi incrementado. Contudo, aqui os valores de WER mais baixos foram observados para modelos com números menores de estados vinculados, diferente do que havia acontecido para a tarefa de ditado, possivelmente devido ao tamanho reduzido do vocabulário da base de teste. O melhor resultado registrado foi obtido pelo modelo acústico com 500 estados vinculados e 64 Gaussianas por mistura, com apenas 0,67% de WER.

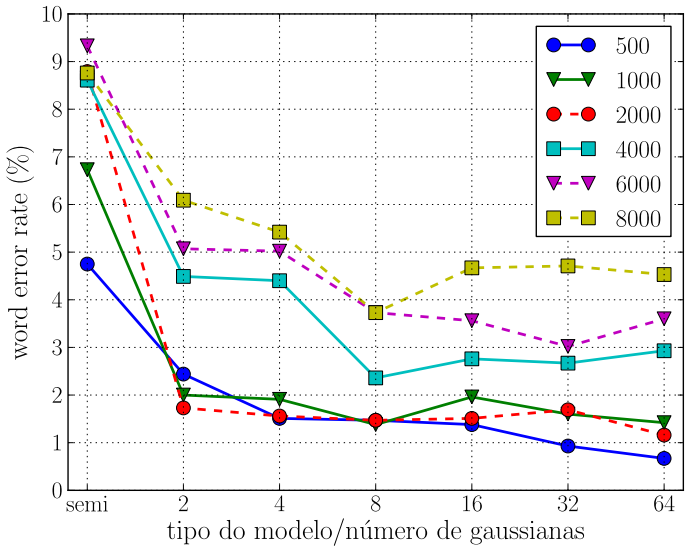


Figura 5.5: WER (%) para a tarefa de comando e controle usando parte do corpus LapsMail.

A seguir serão apresentados os resultados obtidos com os modelos acústicos treinados com o pacote HTK e convertidos para serem usados nos decodificadores Sphinx.

5.3 Avaliação dos modelos convertidos

Este experimento foi realizado visando avaliar a qualidade dos modelos acústicos construídos através do *baseline* de conversão descrito na Seção 4.3, a fim de mensurar possíveis perdas de desempenho relacionadas ao processo de conversão. Dessa forma, o desempenho dos modelos acústicos convertidos foi comparado com os modelos acústicos "originais".

Assim, primeiramente, foram treinados um total de 5 modelos acústicos com diferentes números de Gaussianas por mistura (1, 2, 4, 8 e 16), de acordo com o procedimento descrito na Seção 4.3.3. Em seguida, esses modelos foram convertidos com o *software* HTK2Sphinx.

Para os testes de decodificação foram utilizados os *softwares* Sphinx3 e Julius, o corpus LapsBenchmark e um modelo de linguagem trigrama. A escolha do Julius se justifica pelo fato dos modelos acústicos treinados não definirem uma HMM para o *short-pause*, o que inviabiliza sua utilização no decodificador HDecode.

Assim como nos experimentos anteriores, os parâmetros de decodificação não foram variados ao longo dos testes. A configuração utilizada no decodificador Julius pode ser visualizada no Anexo F. Já para o Sphinx3 foi adotada a mesma configuração utilizada nos testes para ditado (vide Tabela 5.1).

As Figuras 5.6 e 5.7, mostram o contraste de desempenho entre os modelos convertidos e originais em relação ao modelo com 2.000 estados vinculados e 8 Gaussianas por mistura obtido através do *baseline* de treinamento do grupo CMUSphinx.

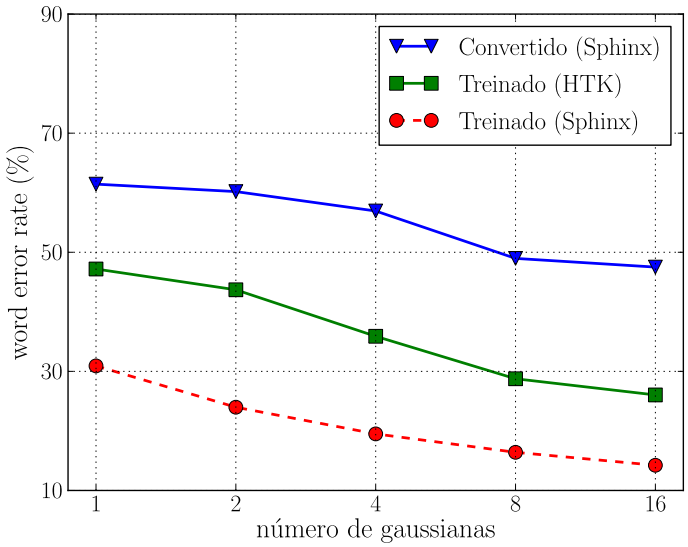


Figura 5.6: WER (%) para a tarefa de ditado com os modelos convertidos

O melhor valor de WER encontrado com os modelos originais foi de 26,02%, obtido pelo modelo acústico com 16 Gaussianas por mistura e fator xRT igual a 1,15. Deve-se

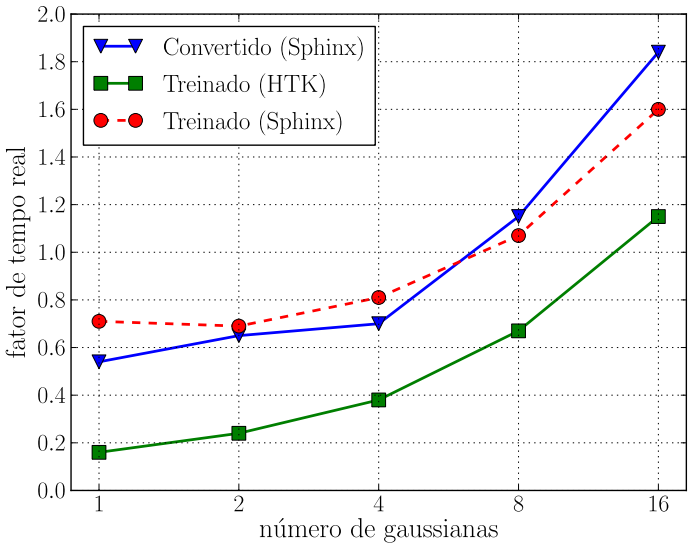


Figura 5.7: Fator xRT para a tarefa de ditado com os modelos convertidos

ressaltar, mais uma vez, que os modelos acústicos treinados não apresentam uma HMM para o *short-pause*, cuja utilização é recomendada enquanto desenvolvendo modelos acústicos para os decodificadores compatíveis com o formato do HTK. Dessa forma, a ausência do *short-pause* pode ter ocasionado perda de desempenho.

De posse dos resultados obtidos com os modelos acústicos convertidos, foi possível constatar um aumento acentuado dos valores de WER (18,57% em média) em relação ao desempenho dos modelos acústicos originais. Também foi possível observar uma alteração sensível em relação aos valores do fator xRT, que para os modelos convertidos foram mais elevados. Entretanto, devido ao algoritmo de busca dos dois decodificadores utilizados serem diferentes, não se pode afirmar que essa diferença de desempenho tenha sido ocasionada pelo processo de conversão.

Pode-se concluir que os modelos convertidos apresentaram um desempenho muito ruim considerando tarefas de ditado, já que a melhor taxa de erro foi de 47,51%, obtida pelo modelo acústico com 16 Gaussianas por mistura para um fator xRT de 1,84.

Devido ao baixo desempenho dos modelos acústicos obtidos através do *baseline* de conversão, o modelo acústico utilizado para o desenvolvimento do aplicativo que é descrito na seção seguinte foi escolhido dentre os modelos obtidos por meio do *baseline* de treinamento do grupo CMUSphinx.

5.4 Rotas: Um aplicativo com suporte a ASR para Android

A fim de demonstrar que os modelos acústicos desenvolvidos neste trabalho podem ser utilizados para o desenvolvimento de aplicativos com suporte a ASR em PB para dispositivos móveis, um aplicativo demonstrativo com funcionalidades básicas de ASR foi construído como prova-de-conceito para a plataforma Android 2.2.

O Rotas, é um *software* que permite que o usuário consulte a informação sobre o itinerário dos ônibus da cidade de Belém do Pará usando para isso comandos de voz. As funcionalidade de ASR em PB são providas pelo decodificador Pocketsphinx aliado a recursos aqui desenvolvidos, tais como: modelo acústico, dicionário fonético e uma gramática livre de contexto. Essa gramática é composta basicamente pelos logradouros atendidos pelo transporte público de Belém e alguns pontos turísticos, como o Entroncamento e Hangar - Centro de Convenções.

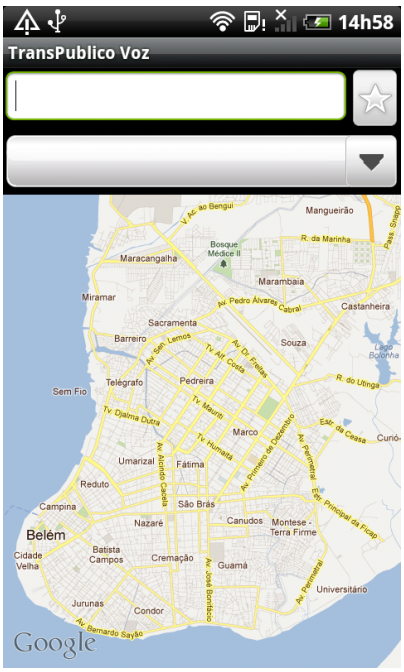
O funcionamento do aplicativo é simples. Para fazer uma consulta, o usuário precisa pressionar o botão marcado com a estrela na tela inicial (vide Figura 5.8a) do aplicativo e então ditar a consulta. O Pocketsphinx é então acionado e o resultado do reconhecimento é utilizado para realizar uma consulta no banco de dados das linhas de ônibus que atendem o logradouro informado na consulta. Os resultados são então mostrados em uma lista, mostrada na Figura 5.8b, para que o usuário possa selecionar uma opção. A rota do ônibus selecionado é então apresentada na interface do Google Maps (vide Figura 5.8c).

5.5 Conclusão

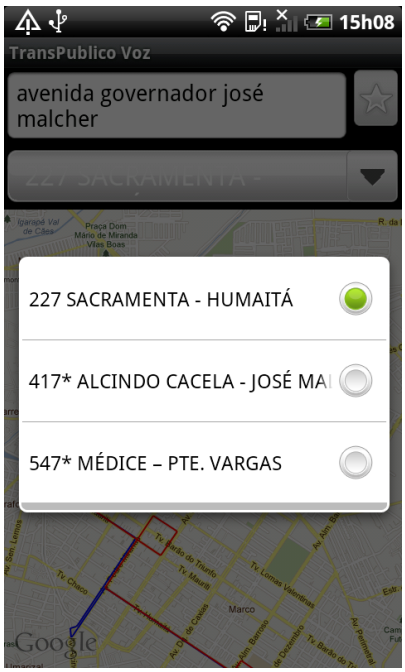
Este capítulo apresentou os resultados dos testes realizados com os modelos acústicos construídos a fim de tornar possível a utilização dos decodificadores do grupo CMUSphinx no desenvolvimento de aplicativos com suporte a ASR em PB, visando principalmente plataformas móveis com o decodificador Pocketsphinx.

Os resultados obtidos pelos modelos acústicos construídos a partir do *baseline* de treinamento do grupo CMUSphinx foram aceitáveis, tanto para a tarefa de ditado quanto comando e controle. Já os modelos acústicos criados a partir do *baseline* de conversão atingiram resultados muito ruins, apresentando uma perda de performance significativa em relação ao desempenho obtido pelos modelos originais.

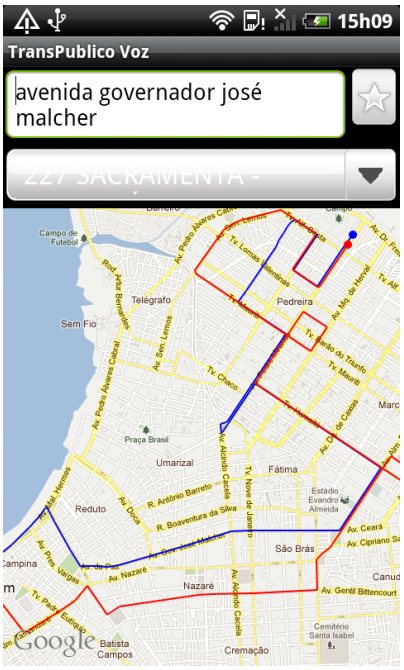
Por fim, como prova de conceito, foi apresentado o aplicativo Rotas, um *software* para a plataforma Android 2.2 com funcionalidades de ASR em PB providas pelo decodificador Pocketsphinx aliado a recursos desenvolvidos neste trabalho. O aplicativo Rotas encontra-se disponível na página do grupo FalaBrasil [FalaBrasil 2012].



(a) Tela inicial



(b) Lista de ônibus



(c) Rota escolhida pelo usuário

Figura 5.8: Telas do aplicativo Rotas.

Capítulo 6

Considerações Finais

Como exposto, nos últimos anos a tecnologia ASR vem sendo cada vez mais utilizada em um vasto número de aplicações e plataformas. Com os investimentos de empresas consolidadas no mercado como a Nuance, a Google, a Microsoft, dentre outras, esses sistemas ASR vem se mostrando cada vez mais robustos, o que propicia um grau cada vez maior de aceitação por parte dos usuários. Contudo, para o PB, esses avanços não acontecem na mesma proporção que para outros idiomas. Dessa forma, recursos para ASR em PB ainda são escassos, principalmente em se tratando de recursos livres.

Nesse contexto, neste trabalho foram desenvolvidos recursos visando aumentar tanto a disponibilidade de ferramentas para o desenvolvimento de aplicativos com suporte a ASR em PB para *desktop* e sistemas embarcados, como também fornecer material que possa servir de base para a realização de pesquisas na área de ASR.

Durante sua primeira etapa, este trabalho focou no desenvolvimento da JLaPSAPI, uma API em Java adequada a especificação JSAPI para o *engine* Coruja. A adição da JLaPSAPI ao pacote de distribuição do Coruja além de ter aumentado sua abrangência, permitindo que o mesmo seja utilizado na plataforma de desenvolvimento Java, também o adequou a uma das especificações para desenvolvimento de aplicativos com suporte a ASR mais difundidas na comunidade. Hoje, a JLaPSAPI é utilizada no SpeechOO, projeto livre desenvolvido em parceria entre o Laboratório de Processamento de Sinais da UFPA e o Centro de Competência em Software Livre da Universidade de São Paulo. A JLaPSAPI encontra-se disponível na página do grupo FalaBrasil.

Os esforços empreendidos na segunda etapa deste trabalho, visaram a disponibilização de recursos para o desenvolvimento de aplicativos com suporte a ASR em PB para dispositivos móveis. Por disponibilizar decodificadores tanto para sistemas embarcados quanto para *desktop*, o objetivo do segundo momento deste trabalho foi desenvolver modelos acústicos e demais recursos necessários utilizando o pacote de ferramentas do grupo CMUSphinx. De acordo com a pesquisa realizada até a produção deste, nenhum trabalho, visando o treinamento de modelos acústicos para PB, foi desenvolvido visando os decodificadores Sphinx. Assim, os relatos aqui apresentados podem servir de ponto de partida para outros grupos de pesquisa.

Os resultados mostraram que para a tarefa de ditado, o melhor sistema ASR desenvolvido utilizou um modelos acústico contínuo com 2.000 estados vinculados e 8 Gaussianas por mistura obtendo 16,41% de WER para um xRT em torno de um. Já para a tarefa de comando e controle, o melhor resultado registrado foi obtido com o modelo acústico contendo 500 estados vinculados e 64 Gaussianas por mistura, com apenas 0,67% de WER. Para a realização dos experimentos para a tarefa de comando e controle, uma base de áudio chamada LaPSMail foi construída e encontra-se disponível na página do grupo FalaBrasil.

Durante a realização dos experimentos tendo como finalidade a conversão de modelos acústicos do formato do HTK para o Sphinx, a ausência de uma documentação completa sobre os algoritmos implementados pelas ferramentas da CMUSphinx, principalmente sobre seu *frontend*, dificultaram a tarefa de compatibilizar as configurações de *frontend* utilizadas no HTK e no Sphinx. Apesar disso, com a abordagem proposta neste trabalho foi possível realizar a conversão, entretanto os modelos acústicos convertidos apresentaram performance muito inferior a dos modelos "originais", apresentando WER em média 18,57% superior. A melhor taxa de erro foi de 47,51%, obtida pelo modelo acústico com 16 Gaussianas por mistura para um fator xRT de 1,84.

Como prova de conceito, um aplicativo para plataforma Android 2.2 com suporte a ASR em PB, foi construído utilizando um modelo acústico construído neste trabalho, além de outros recursos livres necessários. Como forma de incentivo, o código fonte desse aplicativo encontra-se disponível em [FalaBrasil 2012].

6.1 Trabalhos futuros

A JSAPI é uma especificação extensa e neste trabalho apenas uma parte das funcionalidades providas pela mesma foram implementadas na JLaPSAPI. A implementação do restante das funcionalidades está prevista para trabalhos futuros. Essa expansão deve acontecer em paralelo ao crescimento do projeto SpeechOO.

Neste trabalho, foram avaliados vários modelos acústicos. Nos testes realizados, apenas parâmetros referentes a composição dos modelos em si foram investigados. Contudo, sabe-se que para uma melhor avaliação de qualquer sistema ASR se faz necessário encontrar a melhor configuração de parâmetros de decodificação para cada sistema. Sendo assim, em trabalhos futuros o *tuning* dos parâmetros de decodificação será investigado.

Referências Bibliográficas

Bahl, L.R., P.V. Souza, P.S. Gopalakrishnan, D. Nahamoo e M.A. Picheny (1994), Context dependent modeling of phones in continuous speech using decision trees, *em* 'DARPA Speech and Natural Language Processing Workshop', pp. 264–270.

Bellman, Richard (1957), *Dynamic Programming*, Princeton University Press.

CETENFolha (2012), 'CETENFolha: Um corpus de texto para Português Brasileiro'. `acdc.linguateca.pt/cetenfolha/`. Visitado em Março de 2012.

CETUC (2012), 'Um Corpus de Voz em Português Brasileiro'. `http://www.cetuc.puc-rio.br/pos-novo.htm`. Visitado em Março de 2012.

Cheng, Chih-Chieh (2011), Online Learning of Large Margin Hidden Markov Models for Automatic Speech Recognition, Tese de doutorado, UCSD.

Chrome (2012), 'Chrome'. `www.google.com/chrome`. Visitado em Março de 2012.

Cohen, M., H. Franco, N. Morgan, D. Rumelhart e V. Abrash (1992), 'Hybrid neural network/hidden markov model continuous speech recognition', *Proceedings of the International Conference on Spoken Language Processing* .

Colen, W. D. e P. Batista (2010), 'Veja mamãe, sem as mãos! SpeechOO, uma extensão de ditado para o BrOffice.org', *11th Fórum Internacional Software Livre* .

da Cunha, A. M. e L. Velho (2003), Métodos probabilísticos para reconhecimento de voz, Relatório técnico, Laboratório VISGRAF - Instituto de Matemática Pura e Aplicada.

Davis, S. e P. Merlmestein (1980), 'Comparison of Parametric Representations for Monosyllabic Word Recognition in Continuously Spoken Sentences', *IEEE Trans. on ASSP* **28**, 357–366.

dos Santos, Fabiano Weimar, Dante Augusto Couto Barone e André Gustavo Adami (2010), 'A Baseline System for Continuous Speech Recognition of Brazilian Portuguese Using the West Point Brazilian Portuguese Speech Corpus', *The International Conference on Computational Processing of Portuguese (PROPOR)* .

DragonGO (2012), 'Assistente Pessoal Controlado por Voz da empresa Nuance'. `http://www.nuancemobilelife.com/apps/dragon-go-android`. Visitado em Março de 2012.

DragonMobileSDK (2012), 'Dragon Mobile Software Development Kit'. http://www.nuance.com/for-developers/dragon-mobile-sdk/index.htm. Visitado em Março de 2012.

Fagundes, R. e I. Sanches (2003), 'Uma nova abordagem fonético-fonológica em sistemas de reconhecimento de fala espontânea', *Revista da Sociedade Brasileira de Telecomunicações* **95**, 225–239.

FalaBrasil (2012), 'Reconhecimento de Voz para o Português Brasileiro'. http://www.laps.ufpa.br/falabrasil/. Visitado em Março de 2012.

Gulic, Matija, Drazen Lucanin e Ante Simic (2011), 'A digit and spelling speech recognition system for the croatian language', *34th International Convention on Information and Communication Technology. Eletronics and Microeletronics* pp. 23–27.

HTK2Sphinx (2012), 'Conversor de modelos acústicos do HTK para o Sphinx'. http://cmusphinx.sourceforge.net/2010/08/python-htk-converter/. Visitado em Março de 2012.

Huang, X., A. Acero e H. Hon (2001), *Spoken Language Processing*, Prentice-Hall.

Huggins-Daines, David, Mohit Kumar, Arthur Chan, Alan W Black, Mosur Ravishankar e Alex I. Rudnicky (2006), 'Pocketsphinx: A Free, Real-Time Continuous Speech Recogntion System for Hand-Held Devices', *Proceedings of ICASSP* pp. 185–188.

Iris (2011), 'Assistente pessoal controlado por voz para android'. https://play.google.com/store/apps/. Visitado em Março de 2012.

Jelinek, F. (1976), Continuous speech recognition by statistical methods, *em* 'Proceedings. of. the IEEE, vol. 64 no. 4', pp. 532–555.

Jelinek e Frederick (1998), 'Métodos estatísticos para reconhecimento de voz', *The MIT Press* .

JLaPSAPI (2012), 'API em Java Compatível com a Especificação JSAPI para o Coruja'. http://www.laps.ufpa.br/falabrasil/jlapsapi. Visitado em Março de 2012.

JSAPI (2012), 'Java Speech API'. java.sun.com/products/java-media/speech/. Visitado em Março de 2012.

JSG (2012), 'Java Speech Grammar Format Specification'. java.sun.com/products/java-media/speech/forDevelopers/JSGF/. Visitado em Março de 2012.

Juang, H. e R. Rabiner (1991), 'Hidden Markov models for speech recognition', *Technometrics* **33**(3), 251–272.

Junqua, J.-C. e J.-P. Haton (1996), *Robustness in Automatic Speech Recognition*, Kluwer.

Ladefoged, P. (2001), *A Course in Phonetics*, 4ª edição, Harcourt Brace.

Lee, Akinobu (2009), *The Julius Book*, 0.0.2 ed. - rev 4.1.2.

Lee, Akinobu e Tatsuya Kawahara (2009), 'Recent development of open-source speech recognition engine julius', *Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)* .

Lee, Akinobu, Tatsuya Kawahara e Kiyoshiro Shikano (2001), 'Gaussian mixture selection using context-independent HMM', *In Proceedings IEEE-ICASSP* .

Liang, Sheng (1999), *The Java^{TM}Native Interface Programmer's Guide and Specification*, Addison-Wesley.

Lippmann, R.P. (1997), 'Speech recognition by machines and humans', *Speech Communication* **22**, 1–15.

Ma, Guangguang, Wenli Zhou, Jing Zheng, Xiaomei You e Weiping Ye (2009), 'A comparison between HTK and Sphinx on Chinese Mandarin', *International Joint Conference on Artificial Intelligence* .

Maskey, S., M. Bacchiani, B. Roark e R. Sproat (2004), 'Improved name recognition with meta-data dependent name networks', *Proceedings of ICASSP* .

MLDC (2012), 'Microsoft Language Development Center Portugal'. `http://www.microsoft.com/portugal/mldc/default.mspx`. Visitado em Março de 2012.

Neto, Nelson, Carlos Patrick, Aldebaro Klautau e Isabel Trancoso (2010), 'Free tools and resources for Brazilian Portuguese speech recognition', *The Brazilian Computer Society* **16**, 53–68.

Nuance (2012), 'Nuance Communications, Inc.'. `http://www.nuance.com`. Visitado em Março de 2012.

Odell, J. e K. Mukerjee (2007), 'Architecture, user interface, and enabling technology in Windows Vista's speech systems', *IEEE Transactions on Computers* **56, no. 9**, 1156–1168.

Pessoa, L., F. Violaro e P. Barbosa (1999), 'Modelos da língua baseados em classes de palavras para sistema de reconhecimento de fala contínua', *Revista da Sociedade Brasileira de Telecomunicações* **14**(2), 75–84.

Picone, J. (1993), 'Signal modeling techniques in speech recognition', *Proceedings of the IEEE* **81**(9), 1215–47.

Rabiner, L. (1989), 'A tutorial on hidden Markov models and selected applications in speech recognition', *Proceedings of the IEEE* **77**(2), 257–86.

Rabiner, L. e B. Juang (1993), *Fundamentals of Speech Recognition*, PTR Prentice Hall, Englewood Cliffs, N.J.

Sakoe, H. e S. Chiba (1978), 'Dynamic programming algorithm optimization for spoken word recognition', *IEEE Trans. on ASSP* **26**(1), 43–49.

Samudravijaya, K. e M. Barot (2003), 'A comparison of public domain software tools for speech recognition', *Proceedings of Workshop on Spoken Language Processing* pp. 125–131.

Santos, Fabiano, Dante Barone e Andre Adami (2010), 'A Baseline System for Continuous Speech Recognition of Brazilian Portuguese Using The West Point Brazilian Portuguese Speech Corpus', *International Conference on Computational Processing of the Portuguese Language* .

Santos, S. e A. Alcaim (2002), 'Um sistema de reconhecimento de voz contínua dependente da tarefa em língua portuguesa', *Revista da Sociedade Brasileira de Telecomunicações* **17**(2), 135–147.

SAPI (2012), 'Microsoft Speech API'. `www.microsoft.com/speech/`. Visitado em Março de 2012.

Satori, Hassan, Hussein Hiyassat, Mostafa Harti e Noureddine Chenfour (2009), 'Investigation Arabic speech recognition using CMU Sphinx system', *International Arab Journal of Information Technology* **6**.

Schuster, Mike (2010), 'Speech Recognition for Mobile Devices on Google', *PRICAI 2010: Trends in Artificial Intelligence* pp. 8–10.

Schwenk, H (1999), Using boosting to improve a hybrid HMM/neural network speech recognizer, *em* 'ICASSP', pp. 1009–12.

Sethy, A., S. Narayanan e S. Parthasarthy (2002), 'A syllable based approach for improved recognition of spoken names', *Proceedings of the ISCA Pronunciation Modeling Workshop* .

Silva, Enio, Luiz Baptista, Helane Fernandes e Aldebaro Klautau (2005), 'Desenvolvimento de um sistema de reconhecimento automático de voz contínua com grande vocabulário para o Português Brasileiro', *XXV Congresso da Sociedade Brasileira de Computação* .

Silva, Patrick, Pedro Batista, Nelson Neto e Aldebaro Klautau (2010), 'An open-source speech recognizer for Brazilian Portuguese with a windows programming interface', *The International Conference on Computational Processing of Portuguese (PROPOR)* .

Simon (2012), '*Software* Livre para Reconhecimento de Voz que Substitui o *Mouse* e o Teclado'. `http://www.simon-listens.org/index.php?id=122&L=1`. Visitado em Março de 2012.

Singh, Rita, B. Raj e R. M. Stern (1999), 'Automatic Clustering and Generation of Contextual Questions for Tied States in Hidden Markov Models', *Proceedings of ICASSP* pp. 117–120.

Siravenha, Ana, Nelson Neto, Valquíria Macedo e Aldebaro Klautau (2008), 'Uso de Regras Fonológicas com Determinação de Vogal Tônica para Conversão Grafema-Fone em Português Brasileiro', *7th International Information and Telecommunication Technologies Symposium* .

Siri (2011), 'Assistente Pessoal Controlado por Voz para IPhone'. `http://www.apple.com/iphone/features/siri.html`. Visitado em Março de 2012.

SpeechOO (2010), 'SpeechOO: Ditado no LibreOffice'. `code.google.com/p/speechoo/`. Visitado em Março de 2012.

SphinxTrain (2012), 'Pacote de Ferramentas para Treinamento de Modelos Acústicos do grupo CMUSphinx'. `http://sourceforge.net/projects/cmusphinx/files/sphinxtrain/1.0.7/`. Visitado em Março de 2012.

Teruszkin, R. e F.G. Vianna (2006), 'Implementation of a large vocabulary continuous speech recognition system for Brazilian Portuguese', *Journal of Communication and Information Systems* **21**, 204–218.

Varela, Armando, Heriberto Cuayáhuitl e Juan Nolazco-Flores (2003), 'Creating a Mexican Spanish version of the CMU Sphinx-III speech recognition system', *Progress in Pattern Recognition, Speech and Image Analysis* pp. 251–258.

Vertanen, Keith (2006), Baseline WSJ Acoustic Models for HTK and Sphinx: Training Recipes and Recognition Experiments, Relatório técnico, University of Cambridge.

Vintsyuk, T.K. (1968), 'Speech discrimination by dynamic programming', *Kibernetika (Cybernetics)* **4**, 81–88.

Walker, W., P. Lamere, P. Kwok, B. Raj, R. Singh, E. Gouvea, P. Wolf e J. Woelfel (2004), Sphinx-4: A Flexible Open Source Framework for Speech Recognition, Relatório técnico, Sun Microsystems Inc.

Welch, Lloyd R. (2003), 'Hidden Markov Models and the Baum-Welch Algorithm', *IEEE Information Theory Society Newsletter* **53**, 10–12.

Woodland, P. e D. Povey (2002), 'Large scale discriminative training of hidden Markov models for speech recognition', *Computer Speech and Language* **16**, 25–47.

Woodland, P. e S. Young (1993), 'The HTK Tied-State Continuous Speech Recognizer', *In: Proc. Eurospeech'93, Berlim* .

Ynoguti, Carlos Alberto e Fábio Violaro (2001), 'Desenvolvimento de um Conjunto de Ferramentas para Pesquisas em Reconhecimento de Fala', *Telecomunicações (Santa Rita do Sapucaí)* **v.4**, 36–43.

Young, S., D. Ollason, V. Valtchev e P. Woodland (2006), *The HTK Book*, Cambridge University Engineering Department, version 3.4.

Ênio (2008), 'Desenvolvimento de um Reconhecedor Automático de Voz com Suporte a Grandes Vocabulários para o Português Brasileiro', Universidade Federal do Pará, Instituto de Tecnologia. Trabalho de conclusão de curso.

Apêndice A

Exemplo de uso do Coruja a partir da JSAPI

```
package br.ufpa.laps.jlapsapi.recognizer;

import javax.speech.Central;
import javax.speech.EngineModeDesc;
import javax.speech.recognition.Recognizer;
import javax.speech.recognition.RecognizerModeDesc;
import javax.speech.recognition.Result;
import javax.speech.recognition.ResultAdapter;
import javax.speech.recognition.ResultEvent;
import javax.speech.recognition.ResultToken;
import javax.speech.recognition.RuleGrammar;
import javax.speech.recognition.DictationGrammar;

import java.io.FileReader;

public class SimpleRecognition extends ResultAdapter {

        static Recognizer rec;

        public void resultAccepted(ResultEvent e) {
                try {
                        Result r = (Result) (e.getSource());
                        ResultToken tokens[] = r.getBestTokens();
                        for (int i = 0; i < tokens.length; i++){
                                System.out.print(tokens[i].getSpokenText()+" ");
                        }
                }catch (Exception e1) {
                        e1.printStackTrace();
                }
        }

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

```
                        RecognizerModeDesc rmd = (RecognizerModeDesc) Central
                                .availableRecognizers(new EngineModeDesc("Coruja","general",
                                  null, null)).firstElement();
                        rec = Central.createRecognizer(rmd);
                        rec.allocate();
                        rec.addResultListener(new SimpleRecognition());

                        rec.resume();
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

Apêndice B

Alfabeto fonético

Fonema	Exemplos
Vogais orais	
a	j**a**tob**á**, c**a**p**a**cete, c**a**beç**a**, lu**a**
em	**é**, paj**é**, p**e**le, f**e**rro, v**e**lho
e	capac**e**te, r**e**solv**e**r, r**e**sp**e**ito
i	just**i**ça, pa**í**s, láp**i**s, **i**d**i**ota, el**e**
om	**ó**pio, j**o**gos, s**o**zinho, f**o**rte
o	j**o**go, g**o**lfinho, c**o**rp**o**
u	ra**u**l, c**u**lpa, ba**ú**, c**u**r**u**r**u**, log**o**
Fricativas vozeadas	
z	ca**s**a, coi**s**a, qua**s**e, e**x**ato
v	**v**o**v**ó, **v**amos, a**v**ião
zm	**g**eladeira, trove**j**ar
Africadas	
ts	**t**ia, paco**t**e, cons**t**ituin**t**e
dz	**d**ia, cidad**d**e, **d**isco
Plosivas	
b	**b**ar**b**a, a**b**sinto
d	**d**a**d**os, a**d**ministrar
t	pa**t**o, consti**t**uinte
k	**c**as**c**a, **qu**ero, **q**uanto
g	**gu**erra, **g**ato, a**g**uentar, a**g**nóstico
p	**p**a**p**ai, **p**sicólogo, a**p**to

Vogais nasais	
aa	**an**dar, t**am**par, c**an**ção, c**a**ma
ee	**en**tão, t**em**po, b**e**m, m**e**nos
ii	n**i**nho, t**in**ta, lat**i**na, **im**porta
oo	**on**da, campe**õ**es, s**o**mos , h**o**mem
uu	**um**, m**u**ito, **um**bigo
Semi vogais	
w	fáci**l**, vo**l**tar, e**u**, q**u**ase
j	pa**i**, fo**i**, caracó**i**s, micróbi**o**
ww	nã**o**, cã**o**
jj	mu**i**to, be**m**, parabé**n**s, compõ**e**
Líquidas	
l	**l**aranja, **l**eitão
lm	ca**lh**ar, co**lh**eita, me**lh**or
rm	ca**rr**o, **r**ua, **r**ato, ca**r**ga, ge**r**me
xm	casa**r**, ce**r**to, a**r**pa, a**r**co
r	ga**r**oto, f**r**ango, po**r** exemplo
Fricativas não-vozeadas	
f	**f**esta, **f**an**f**arrão, a**f**ta, a**f**luente
s	**s**apo, ca**ç**ar, cre**sc**er, se**ss**ão, lápi**s**, capa**z**, ca**s**ca, e**xc**esso
sm	**ch**á, **x**aveco, ca**ch**orro
Consoantes nasais	
m	**m**a**m**ãe, e**m**ancipar
n	**n**ome, ate**n**uar, enca**n**ação
jm	casi**nh**a, gali**nh**a

Apêndice C

Extração de parâmetros com Sphinx

Arquivo de configuração para extração de parâmetros para modelos contínuos

```
-alpha 0.97                    -nfft 512
-dither yes                    -wlen 0.0256
-doublebw no                   -transform legacy
-nfilt 40                      -feat 1s_c_d_dd
-ncep 13                       -agc none
-lowerf 133.33334              -cmn current
-upperf 6855.4976              -varnorm no
```

Arquivo de configuração para extração de parâmetros para modelos semi-contínuos

```
-alpha 0.97                    -nfft 512
-dither yes                    -wlen 0.0256
-doublebw no                   -transform legacy
-nfilt 40                      -feat s2_4x
-ncep 13                       -agc none
-lowerf 133.33334              -cmn current
-upperf 6855.4976              -varnorm no
```

Apêndice D

Lista de sentenças da LaPSMail

abrir email
ler mensagens
abrir caixa de entrada
criar mensagem
criar nova mensagem
responder ao remetente
responder
responder a todos
enviar
salvar anexo
adicionar contato
inserir contato
inserir destinatario
anexar arquivo
salvar mensagem
salvar rascunho
próxima
próxima mensagem
anterior
mensagem anterior
organizar por ordem alfabética
organizar por data
organizar por remetente
exibir mensagens não lidas
próxima não lida
primeira mensagem
primeira não lida
última mensagem recebida
última mensagem não lida
não lidas

mover mensagem
encaminhar mensagem
encaminhar
excluir mensagem
deletar
enviar para lixeira
fechar
sair
descartar
procurar
spam
ler
pára
gravar
andrey
anderson
agnaldo
aldebaro
bruno
adalbery
ana carolina
fernanda
nelson
josué
renan
danilo
jonathas
lailson
gustavo
mariana
muller
diego
diogo
sílvia
nagib
marcos
kelly
mônica
guilherme
william
jefferson
claudomir
lucila
pedro

rodrigo
leonardo
claudio
fabiola
pelaes
rafael
suane
charles
marcel
ericson
igor
cleyton
hugo

Apêndice E

Gramática para a LaPSMail

```
#JSGF V1.0;

grammar mailbenchmark;

public <sentenca> = <comando_simples> | <comando_composto> ;

<comando_composto> = abrir <pos_abrir> | adicionar
contato | anexar arquivo | criar <pos_criar> | encaminhar
mensagem | enviar para lixeira | excluir mensagem | exibir
mensagens não lidas | inserir <pos_inserir> | ler mensagens | mensagem
anterior | mover mensagem | não lidas | organizar
<pos_organizar> | primeira <pos_primeira> | próxima <pos_proxima> | responder
<pos_responder> | salvar <pos_salvar> | última <pos_ultima>;

<pos_abrir> = caixa de entrada | email;

<pos_criar> = mensagem | nova mensagem;

<pos_inserir> = contato | destinatário;

<pos_organizar> = por data | por ordem alfabética | por remetente;

<pos_responder> = a todos | ao remetente;

<pos_salvar> = anexo | mensagem | rascunho;

<pos_ultima> = mensagem não lida | mensagem recebida;

<pos_proxima> = mensagem | não lida;

<pos_primeira> = mensagem | não lida;

<comando_simples> = ler | responder | deletar | fechar | sair | descartar |
procurar | gravar | próxima | anterior | pára ;
```

Apêndice F

Arquivo de configuração do Julius

```
#
# Sample Jconf configuration file
# for Julius library rev.4.2
#
# 1) Options can also be specified in command line option.
#    The values are default values in Julius.
# 2) For file name, relative path must be relative to THIS FILE.
# 3) Texts after '#' at each line will be ignored.  If you want to specify
#    '#', use '\#'.
# 4) Each line should be no longer than 512 bytes.
#
#

#########################################################################
#### JULIUS APPLICATION OPTION (not a part of JuliusLib)
#########################################################################
#-outfile                       # save each result in separate file
#-separatescore                 # print AM / LM scores separately
#-callbackdebug                 # print callback names at call for debug
#-charconv from to              # print with character set conversion
#-nocharconv                    # disable "-charconv"
#-module                        # start in module mode
#-record dir                    # record each inputs into dir
#-logfile file                  # redirect logs to file
#-nolog                         # disable all logs
#-help                          # print help, and exit


#########################################################################
#### GLOBAL OPTIONS
#########################################################################
####
#### Misc. Options
####
#-C jconffile                   # include jconf file at here
#-version                       # print version info to stderr, and exit
#-setting                       # print version info to stderr, and exit
```

```
#-quiet                         # output less log
#-debug                         # output more log for debug
#-check wchmm                   # for debug, enter debug shell mode
#-check trellis                 # for debug, enter debug shell mode
#-check triphone                # for debug, enter debug shell mode
#-demo                          # same as "-quiet -progout"

####
#### Audio Input
####
#-input mic                     # live microphone
#-input rawfile                 # wavefile
#-input mfcfile                 # MFCC file (HTK Parameter file)
#-input stdin                   # waveform from standard input
#-input adinnet                 # waveform via network client
#-input netaudio                # DatLink server
#-input oss                     # OSS API input (if available)
#-input alsa                    # ALSA API input (if available)
#-input esd                     # ESounD daemon input (if available)

-filelist etc/mfc_test.txt              # input file list
#-notypecheck                   # does not check parameter type of input
#-48                            # 48kHz sampling > 16kHz conv. (16kHz only)
#-NA devname                    # hostname for DatLink server
#-adport 5530                   # port number for adinnet
#-nostrip                       # do not strip zero samples
#-zmean                         # remove DC offset (use long input average)
#-nozmean                       # disable "-zmean" specified before

####
#### Speech segment detection by level and zero-cross
####
#### default: on for microphone, off for other sources
####
#-cutsilence                    # detection on
#-nocutsilence                  # detection off
#-lv 2000                       # level threshold (0-32767)
#-zc 60                         # zero-cross threshold (times in sec.)
#-headmargin 300                # head silence margin (msec)
#-tailmargin 400                # tail silence margin (msec)
#-rejectshort 0                 # reject shorter input (msec)

####
#### Input rejection by average power (EXPERIMENTAL)
####
####  This will be enabled by "--enable-power-reject" on compilation.
####  Should be used with Decoder VAD or GMM VAD.
####  Valid for real-time input only.
####
#-powerthres 9.0                # reject input by avg. energy

####
#### Gaussian Mixture Model
```

```
####
####  GMM will be used for input rejection by accumurated score, or
####  for GMM-based frontend VAD when "--enable-gmm-vad" specified.
####
####  NOTE: If you use MFCC for the GMM which is different from AM, you
####  should also set the parameters like other AM with an option "-AM_GMM".
####  If "-AM_GMM" is not used, Julius assume GMM to use the same
####  parameter as the first AM.
####
#-gmm hmmdefs                   # GMM definitions in HTK format
#-gmmnum 10                     # num of Gaussians to be computed per GMM
#-gmmreject string              # comma-separated list of GMM name to reject
#### GMM_VAD
#-gmmmargin 20                  # head margin for GMM based VAD in frames

####
#### Decoding option
####
#### Real-time processing means concurrent processing of MFCC computation
#### and 1st pass decoding.  By default, real-time processing on the
#### 1st pass is on for microphone / adinnet / netaudio input, and
#### off for others.
####
#-realtime                      # force real-time processing
#-norealtime                    # force non real-time processing

####
#### Plug-in
####
#### See plugin/00readme.txt for detail
####
#-plugindir ./plugin:/usr/local/share/julius/plugins


#################################################################
#### INSTANCE DEFINITION FOR MULTI DECODING
#################################################################
####
#### The following three argument will create a new configuration set
#### with default parameters, and switch current set to it.  Jconf
#### parameters specified after the option will be set into the current
#### set.
####
#### To do multi-model decoding, these argument should be specified at
#### the first of each model / search instances with different names.
#### Any options before the first instance definition will be IGNORED.
####
#### When no instance definition is found (as older version of Julius),
#### all the options are assigned to a default instance named "_default".
####
#### Please note that decoding with a single LM and multiple AMs is not
#### fully supported.  For example, you may want to construct the
#### jconf file as this:
```

```
####
####  -AM am_1 -AM am_2
####  -LM lm (LM spec..)
####  -SR search1 am_1 lm
####  -SR search2 am_2 lm
####
#### This type of model sharing is not supported yet, since some part
#### of LM processing depends on the assigned AM.  Instead, you can
#### get the same result by defining the same LMs for each AM, like this:
####
####  -AM am_1 -AM am_2
####  -LM lm_1 (LM spec..)
####  -LM lm_2 (same LM spec..)
####  -SR search1 am_1 lm_1
####  -SR search2 am_2 lm_2
####

## Create a new AM configuration set, and switch current to it.
## You should give a unique name.
#-AM name

## Create a new LM configuration set, and switch current to it.
## You should give a unique name.
#-LM name

## Create a new Search configuration set with AM and LM, and switch
## current to it.  AM and LM name can be either name or ID number.
#-SR name am_name_or_id lm_name_or_id

## Switch current AM to special one reserved for GMM, to specify
## analysis parameter for GMM.  Be sure not to confuse with normal AM
## configuration.
# -AM_GMM

## When using instance declarations, global options should be placed
## at top before any instance declaration, or after this option below.
## This option is only a switcher and can be used anywhere anytime.
# -GLOBAL

## This option disables the strict section checkings and back to 4.0
# -nosectioncheck

######################################################################
#### LANGUAGE MODEL (-LM)
######################################################################
####
#### Only one type of LM can be specified for a LM configuration.
####

####
#### N-gram
####
#-d binary_ngram_file           # N-gram in Julius binary format
```

```
-nlr etc/lapsam.lm                      # forward (left-to-right) N-gram
#-nrl rev_ngram                 # backward (right-to-left) N-gram
-v etc/dictionary.dic                   # word dictionary
## param.
#-silhead "<s>"                 # beginning-of-sentence (silence) word
#-siltail "</s>"                # end-of-sentence (silence) word
#-mapunk "<unk>"                # word to which unknown words should be mapped
#-iwspword                      # add a pause word to the dictionary
#-iwspentry "<UNK> [sp] sp sp"  # word that will be added by "-iwspword"
#-sepnum 150                    # num of high freq words to linearize
#-adddict dictfile              # append additional word dictionary
#-addword entry                 # append additional word entry

####
#### Grammar
####
#### "-gram", "-gramlist" can be used multiple times
#-gram gramprefix               # (comma-separated list of) grammar file prefix
#-gramlist txtfile              # text file containing grammar prefixes
#-dfa dfafile -v dictfile       # specify DFA and dictionary separately
#-nogram                        # reset all grammar list already specified

####
#### Isolated Word
####
#-w dictfile                    # word dictionary
#-wlist txtfile                 # text file containing dictionaries
#-nogram                        # reset all dictfiles already specified
## param.
#-wsil silB silE NULL           # head / tail silence models to be appended

####
#### User-defined LM
####
#-userlm                        # declare to use user LM defined in program

####
#### misc LM option
####
#-forcedict                     # skip error word entries (no stop on error)

###########################################################################
#### ACOUSTIC MODEL (-AM)  (-AM_GMM)
###########################################################################
####
#### Acoustic analysis parameters are included in this section, since
#### the AM defines the required parameter.  You can use different MFCC
#### type for each AM.
#### For GMM, the same parameter should be specified after "-AM_GMM"
####
#### When using multiple AM,  the values of "-smpPeriod", "-smpFreq",
#### "-fsize" and "-fshift" should be the same among all AM.
####
```

```
## Acoustic model
-h model/cd_tied_1/hmmdefs                    # acoustic HMM (ascii or Julius binary)
-hlist etc/tiedlist             # HMMList to map logical phone to physical
#-tmix 2                        # # of mixture to compute in a mixture PDF
#-spmodel "sp"                  # name of a short-pause silence model
#-multipath                     # force enable MULTI-PATH model handling
#-gprune {safe|heuristic|beam|none|default} # Gaussian pruning method
#-iwcd1 {max|avg|best 3}        # Inter-word triphone approximation method
#-iwsppenalty -1.0              # pause insertion penalty for "-iwsp"
#-gshmm hmmfile                 # HMM for Gaussian mixture selection
#-gsnum 24                      # Threshold number of HMM for gshmm

## Analysis
#-smpPeriod 625                 # sampling period (ns) (= 10000000 / smpFreq)
#-smpFreq 16000                 # sampling rate (Hz)
#-fsize 400                     # window size (samples)
#-fshift 160                    # frame shift (samples)
#-preemph 0.97                  # pre-emphasis coef.
#-fbank 24                      # number of filterbank channels
#-ceplif 22                     # cepstral liftering coef.
#-rawe                          # use raw energy
#-norawe                        # disable "-rawe" (this is default)
#-enormal                       # normalize log energy
#-noenormal                     # disable "-enormal" (this is default)
#-escale 1.0                    # scaling log energy for enormal
#-silfloor 50.0                 # energy silence floor in dB for enormal
#-delwin 2                      # delta window (frames)
#-accwin 2                      # acceleration window (frames)
#-hifreq -1                     # cut-off hi frequency (Hz) (-1: disable)
#-lofreq -1                     # cut-off low frequency (Hz) (-1: disable)
#-zmeanframe                    # frame-wise DC offset removal (same as HTK)
#-nozmeanframe                  # disable "-zmeanframe" (this is default)

## Cepstral mean / variance normalization
#-cmnload filename              # load initial cep. mean / variance on startup
#-cmnsave filename              # save cep. mean / variance at each input end
#-cmnupdate                     # update beginning cep. data at each input
#-cmnnoupdate                   # keep initial mean, disable "-cmnupdate"
#-cmnmapweight 100.0            # weight for MAP-CMN
#-cvn                           # enable variance normalization

## Vocal tract length normalization (VTLN)
#-vtln 1.0 300 4800             # enable VTLN (alpha, lowerfreq, upperfreq)

## Spectral subtraction (default: disabled)
#-sscalc                        # do SS, estimate noise from head sil
#-sscalclen 300                 # length of head silence for "-sscalc" (msec)
#-ssload filename               # do SS, load noise spectrum saved by "mkss"
#-ssalpha 2.0                   # alpha coef. for spectral subtraction
#-ssfloor 0.5                   # spectral floor coef.

## Others
```

```
#-htkconf confs/edaz.conf              # load analysis settings from HTK Config file

#################################################################
#### RECOGNIZER (-SR)
#################################################################
####
#### Default values for beam width and LM weights will change
#### according to compile-time setup of JuliusLib and model specification.
#### Please see the startup log for the actual values.
####

####
#### parameter (common)
####
#-inactive                      # start this process with inactive status
#-1pass                         # perform only the 1st pass, omit 2nd pass
#-no_ccd                        # switch off the phone context dependency
#-force_ccd                     # force on the phone context dependency
#-cmalpha 0.05                  # CM alpha value
#-iwsp                          # append a skippable sp at all word ends
#-transp 0.0                    # transition penalty for transparent words

####
#### parameter (1st pass)
####
#-lmp weight penalty            # LM weight and word insertion penalty (pass1)
#-penalty1 penalty              # word insertion penalty for grammar (pass1)
#-b width                       # beam width (# of nodes)
#-bs score                      # beam width (score)
#-nlimit 3                      # with enable-wpair-nlimit, set max N at nodes
#-progout                       # progressive output while decoding
#-proginterval 300              # output interval in msec for "-progout"

####
#### parameter (2nd pass)
####
#-lmp2 weight penalty           # LM weight and word insertion penalty (pass2)
#-penalty2 penalty              # word insertion penalty for grammar (pass2)
#-b2 width                      # envelope beam width of 2nd pass (#word)
#-sb 80.0                       # envelope score width at 2nd pass
#-s 500                         # hypotheses stack size on 2nd pass (#hypo)
#-m 2000                        # hypotheses overflow threshold (#hypo)
#-n n                           # num of sentences to find
#-output 1                      # num of sentences to output as result
#-lookuprange 5                 # hypo. lookup range at word expansion (#frame)
#-looktrellis                   # expand only trellis words in grammar
#-fallback1pass                 # output 1st pass result when 2nd pass fails

####
#### short-pause segmentation (and decoder-based VAD)
####
#-spsegment                     # enable sp segmentation (or decoder VAD)
#-spdur 10                      # # of frames to detect a short pause
```

```
#-pausemodels string            # comma-separated pause model names
#### for decoder-VAD
#-spmargin 40                   # backstep margin at trigger up (frame)
#-spdelay 4                     # decision delay at trigger up (frame)

####
#### lattice output
####
#-lattice                       # output result in word graph (aka -graphout)
#-graphrange 0                  # merge same words nearby, -1 to disable merge
#-graphcut 80                   # graph depth cut threshold (in depth)
#-graphboundloop 20             # max itertations for boundary adjustment loop
#-graphsearchdelay              # activate an alternate generation algorithm
#-nographsearchdelay            # disable "-graphsearchdelay"

####
#### confusion network output
####
#-confnet                       # enable confusion network output
#-noconfnet                     # disable confusion network output

####
#### multi-grammar output (for grammar and isolated word)
####
#-multigramout                  # output max hypo for each grammar
#-nomultigramout                # disable "-multigramout"

####
#### forced alignment
####
#-walign                        # enable alignment for result at word level
#-palign                        # enable alignment for result at phoneme level
#-salign                        # enable alignment for result at state level

################################################################ end of file
```

Apêndice G

Arquivos de configuração utilizados no HCopy e Sphinx_fe

Arquivo de configuração utilizado no HCopy

```
USESILDET = FALSE
ENORMALISE = TRUE
NUMCEPS = 12
CEPLIFTER = 22
NUMCHANS = 26
USEPOWER = TRUE
PREEMCOEF = 0.97
USEHAMMING = T
WINDOWSIZE = 250000.0
TARGETRATE = 100000.0
TARGETKIND = MFCC_0
ZMEANSOURCE = T
```

Arquivo de configuração utilizado no Sphinx_fe

```
-alpha 0.97
-nfilt 26
-ncep 13
-lowerf 1
-upperf 8000
-transform htk
-lifter 22
-remove_dc yes
```