UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

SIGNA: UMA APLICAÇÃO PARA ENSINO-APRENDIZAGEM DE LIBRAS

JÚLIO CÉSAR BATISTA

BLUMENAU
2015

2015/1-20

JÚLIO CÉSAR BATISTA

SIGNA: UMA APLICAÇÃO PARA ENSINO-APRENDIZAGEM DE LIBRAS

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Ciência da Computação do Centro de Ciências Exatas e Naturais da Universidade Regional de Blumenau como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Prof. Aurélio Faustino Hoppe, M.Sc. - Orientador

BLUMENAU
2015

2015/1-20

SIGNA: UMA APLICAÇÃO PARA ENSINO-APRENDIZAGEM DE LIBRAS

Por

JÚLIO CÉSAR BATISTA

Trabalho de Conclusão de Curso aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II pela banca examinadora formada por:

Presidente:	Prof. Aurélio Faustino Hoppe, M.Sc. – Orientador, FURB
Membro:	Prof. Dalton Solano dos Reis, M.Sc. – FURB
Membro:	Profa. Joyce Martins, M.Sc. – FURB

Blumenau, 06 de junho de 2015

Dedico este trabalho à minha família e meus amigos por sempre estarem ao meu lado me apoiando e me motivando.

AGRADECIMENTOS

A Deus e a Nossa Senhora Aparecida, por todas as oportunidades em minha vida.

À minha família, em especial aos meus pais, Marina Pellense e Manoel Ernesto Batista, que sempre me apoiaram e me incentivaram nos estudos.

À minha namorada, Juliana Carolina Batista, por todo o carinho, compreensão e ajuda.

Ao meu orientador, Aurélio Faustino Hoppe, pela amizade, por ter acreditado nessa ideia desde o começo, pelo acompanhamento e paciência durante o desenvolvimento desse projeto.

Aos meus colegas de faculdade e de trabalho da Ellevo Soluções e da Inventti Soluções, que sempre se mostraram dispostos a trocar ideias e ajudar durante essa jornada.

Aos meus amigos, Thiago Ruan Tesch, Weslei Arnoldo e Jackson Raul Tiedt, que sempre me apoiaram e me incentivaram em minhas escolhas.

Aos professores do curso de Ciência da Computação, pela amizade e pelo aprendizado que proporcionou a realização deste trabalho.

Ao colega, César Roberto de Souza, que dedicou o seu tempo em me auxiliar com conselhos referentes a este projeto.

If I have seen further it is by standing on the shoulders of giants.

Isaac Newton

RESUMO

Este trabalho apresenta o desenvolvimento de uma aplicação para auxílio no ensino-aprendizagem da Língua Brasileira de Sinais utilizando o dispositivo Leap Motion. O reconhecimento de sinais é feito em tempo real utilizando algoritmos de aprendizado de máquina para classificar as características extraídas das mãos do usuário utilizando o Leap Motion. Esses algoritmos utilizam uma base de dados de amostras inicial que pode ser expandida pelos usuários. O *framework* Accord.NET foi utilizado para providenciar implementações dos algoritmos de aprendizado de máquina e o reconhecimento em tempo real foi feito utilizando *WebSocket* com o *framework* ASP.NET SignalR. Foram realizados testes para verificar o desempenho e a usabilidade da aplicação. A partir dos testes de desempenho foi possível perceber que a aplicação pode ser utilizada para reconhecimento de sinais em tempo real, sendo que testes de validação cruzada apresentaram 86% de precisão no reconhecimento de sinais. A partir de testes de usabilidade foi possível perceber que a aplicação pode ser utilizada para o auxílio no ensino-aprendizagem básico da Língua Brasileira de Sinais. No entanto, a aplicação ainda apresenta limitações com sinais que apresentam características semelhantes, expressão facial e quando demonstram oclusão dos dedos para o Leap Motion.

Palavras-chave: Máquinas de vetores de suporte. Modelos ocultos de Markov. Leap Motion. Língua brasileira de sinais. Reconhecimento de sinais.

ABSTRACT

This work presents the development of an application to assist in teaching and learning the Brazilian Sign Language using the Leap Motion device. The sign recognition is done in real time using machine learning algorithms to classify features extracted from the user of hands with the help of Leap Motion. These algorithms use an initial sample database that can be expanded by the users. The Accord.NET framework was used in order to provide implementations of the machine learning algorithms and the real time recognition was done using WebSocket with the ASP.NET SignalR framework. Tests were performed to verify the performance and usability of the application. From performance tests it was possible to notice that the application can be used for sign recognition in real time, and crossed validation tests showed 86% of accuracy in sign recognition. From usability tests it was possible to observe that the application can be used to help the basic teaching and learning of the Brazilian Sign Language. However, the application has limitations with signs that exhibit similar features, facial expression and when they show occlusion of fingers for the Leap Motion.

Key-words: Support vector machines. Hidden Markov models. Leap Motion. Brazilian sign language. Sign recognition.

LISTA DE FIGURAS

Figura 1 – Exemplo de um sinal icônico e de um sinal arbitrário 18
Figura 2 – Alfabeto datilológico de LIBRAS 19
Figura 3 – Sinais dos números de um a nove em LIBRAS 20
Figura 4 – Dispositivo Leap Motion 21
Figura 5 – Área de interação do Leap Motion 22
Figura 6 – Sistema de coordenadas do Leap Motion 22
Figura 7 – Informações da mão retornadas pela API do Leap Motion 23
Figura 8 – Hierarquia de objetos da API do Leap Motion 24
Figura 9 – Classificação linear de dois conjuntos 27
Figura 10 – Mapeamento do espaço original para o espaço de características 27
Figura 11 – Método de classificação um-contra-todos 28
Figura 12 – Método de classificação um-contra-um 29
Figura 13 – Simetria da matriz de decisão 29
Figura 14 – Decisão por eliminação baseada em DDAG 30
Figura 15 – Modelo gráfico probabilístico dirigido 31
Figura 16 – Modelo gráfico de um HMM 32
Figura 17 – Exemplo de um classificador HMM 34
Figura 18 – Conjunto com 10 gestos estáticos utilizados 35
Figura 19 – Projeção de um desenho 3D em três planos 2D 36
Figura 20 – Diagrama de casos de uso 40
Figura 21 – Diagrama de pacotes da aplicação 41
Figura 22 – Diagrama de classes da estrutura de um sinal 42
Figura 23 – Diagrama de classes para manipulação de dados do Leap Motion 43
Figura 24 – Diagrama de classes do pacote `Extração de características` 44
Figura 25 – Diagrama de classes do pacote `Treinamento` 46
Figura 26 – Diagrama de classes para classificação de um sinal no servidor 47
Figura 27 – Diagrama de classes do reconhecimento de sinais no cliente 48
Figura 28 – Diagrama de classes dos algoritmos de reconhecimento de sinais no cliente 49
Figura 29 – Diagrama de classes com os estados para reconhecimento de um sinal dinâmico 50
Figura 30 – Fluxo para reconhecimento de um sinal 52
Figura 31 – Funcionamento da classe `FrameBuffer` 56

Figura 32 – Exemplo da projeção da ponta do dedo no plano gerado pela palma da mão 59
Figura 33 – Ângulos entre os dedos 60
Figura 34 – Distância da palma da mão em relação ao primeiro *frame* 64
Figura 35 – *Frames* que representam o primeiro e último *frame* 71
Figura 36 – Interface inicial da aplicação 83
Figura 37 – Interface da aplicação ao reconhecer um sinal 83
Figura 38 – Página para importar um novo sinal 84
Figura 39 – Diagrama de classes para persistência de dados 103
Figura 40 – Diagrama de classes auxiliares para gerenciamento de sinais 104
Figura 41 – Diagrama de classes úteis da camada servidor 104
Figura 42 – Diagrama de classes para inicialização dos algoritmos 105
Figura 43 – Diagrama de classes que permitem a comunicação com a camada servidor 106
Figura 44 – Diagrama de classes auxiliares da camada cliente 106
Figura 45 – Diagrama de classes para inicialização de câmera 107
Figura 46 – Diagrama de classes para inicialização de cena 107
Figura 47 – Diagrama de atividades que representa o fluxo de reconhecimento de um sinal estático 108
Figura 48 – Diagrama de estados do reconhecimento de um sinal dinâmico 108
Figura 49 – Atividades no reconhecimento de um sinal dinâmico 109
Figura 50 – Página inicial do Leap Recorder 111
Figura 51 – Botões de controle para recortar uma gravação 112

LISTA DE QUADROS

Quadro 1 – Exemplos na forma de uma tabela atributo valor 25
Quadro 2 – Equação de um hiperplano 26
Quadro 3 – Função de decisão sobre um hiperplano 26
Quadro 4 – Funções *kernel* comuns 28
Quadro 5 – Probabilidade de uma sequência de observações em um HMM 32
Quadro 6 – Matriz de transição e matriz de observação 32
Quadro 7 – Critério de máxima verossimilhança 34
Quadro 8 – Comparativo entre os trabalhos correlatos 38
Quadro 9 – UC01 Importar sinal 40
Quadro 10 – UC02 Solicitar treinamento dos algoritmos 40
Quadro 11 – UC03 Reproduzir um sinal 41
Quadro 12 – Código do método `SalvarAmostraDoSinal` 53
Quadro 13 – Código do método `Adicionar` 53
Quadro 14 – Código para carregar o próximo sinal 54
Quadro 15 – Método `ProximoSinal` da classe `Sinais` 54
Quadro 16 – Código para registrar um observador na API do Leap Motion 55
Quadro 17 – Código do método `onFrame` 56
Quadro 18 – Método `extrairParaAmostra` 57
Quadro 19 – Métodos `_extrairDadosDaMao` e `_extrairDadosDosDedos` 58
Quadro 20 – Implementação do método `CaracteristicasDoFrame` 59
Quadro 21 – Código do método `AngulosEntreDedos` 60
Quadro 22 – Código do método `DaAmostra` da classe `CaracteristicasSinalEstatico` 61
Quadro 23 – Método `DaAmostra` da classe `CaracteristicasSinalDinamico` 62
Quadro 24 – Código para calcular a distância da mão entre os frames 63
Quadro 25 – Código da classe `CaracteristicasSinalEstaticoComTipoFrame` 65
Quadro 26 – Código do método `Processar` 66
Quadro 27 – Implementação da classe `DadosSinaisEstaticos` 67
Quadro 28 – Código da classe `DadosSinaisDinamicos` 68
Quadro 29 – Implementação da classe `DadosFramesSinaisDinamicos` 69

Quadro 30 – Método GerarEntradasESaidasPrimeiroFrame 70
Quadro 31 – Método GerarEntradasESaidasUltimoFrame 70
Quadro 32 – Código do método Aprender da classe Svm 72
Quadro 33 – Método Aprender da classe Hmm 73
Quadro 34 – Método _onFrame que faz o reconhecimento de um sinal 74
Quadro 35 – Método reconhecer da classe ReconhecedorDeSinaisOnline 74
Quadro 36 – Método reconhecer da classe SinalEstatico 75
Quadro 37 – Método Reconhecer da classe GerenciadorSinais 75
Quadro 38 – Método Classificar da classe Svm 75
Quadro 39 – Método reconhecer da classe SinalDinamico 76
Quadro 40 – Código do método framesSaoIguais 77
Quadro 41 – Métodos para fazer a comparação entre a posição dos dedos 78
Quadro 42 – Implementação de *buffer* da classe SinalDinamicoReconhecendo 79
Quadro 43 – Método reconhecer da classe SinalDinamicoNaoReconheceuFrame 79
Quadro 44 – Método ReconhecerPrimeiroFrame da classe GerenciadorSinaisDinamicos 80
Quadro 45 – Método reconhecer da classe SinalDinamicoReconheceuPrimeiroFrame 81
Quadro 46 – Método ReconhecerUltimoFrame da classe GerenciadorSinaisDinamicos 81
Quadro 47 – Método reconhecer da classe SinalDinamicoReconheceuUltimoFrame 82
Quadro 48 – Método Classificar da classe Hmm 82
Quadro 49 – Resultado da aquisição de dados dos sinais com o Leap Motion 86
Quadro 50 – Perfil dos usuários 94
Quadro 51 – Avaliação das tarefas executadas 95
Quadro 52 – Avaliação de usabilidade 96
Quadro 53 – Comparação com os trabalhos correlatos 97
Quadro 54 – Formato JSON de um sinal com uma mão 110
Quadro 55 – Questionário de perfil do usuário 113
Quadro 56 – Lista de tarefas 114
Quadro 57 – Questionário de usabilidade 114

LISTA DE TABELAS

Tabela 1 – Quantidade de amostras de cada sinal utilizada para treinar os algoritmos 87
Tabela 2 – Resultado das funções *kernel* para sinais estáticos 88
Tabela 3 – Resultado do custo da SVM das funções *kernel* para sinais estáticos 88
Tabela 4 – Resultado das funções *kernel* para *frames* de sinais dinâmicos 88
Tabela 5 – Resultado do custo da SVM das funções *kernel* para *frames* de sinais dinâmicos. 89
Tabela 6 – Resultado da validação cruzada de sinais estáticos 89
Tabela 7 – Resultado da validação cruzado de sinais dinâmicos 90
Tabela 8 – Resultado da validação cruzada de *frames* de sinais dinâmicos 91
Tabela 9 – Tempo médio para inicializar os algoritmos 92
Tabela 10 – Tempo médio para classificar um sinal 92

LISTA DE ABREVIATURAS E SIGLAS

API – *Application Programming Interface*

CSS – *Cascading Style Sheets*

DDAG – *Decision Directed Acyclic Graph*

HCRF – *Hidden Conditional Random Fields*

HMM – *Hidden Markov Model*

HTML – *HyperText Markup Language*

HTTP – *Hypertext Transfer Protocol*

IDE – *Integrated Development Environment*

JSON – *Javascript Object Notation*

LIBRAS – Língua Brasileira de Sinais

SMO – *Sequential Minimal Optimization*

SVM – *Support Vector Machine*

TCP – *Transmission Control Protocol*

UML – *Unified Modeling Language*

URL – *Unified Resource Locator*

SUMÁRIO

1 INTRODUÇÃO 16
1.1 OBJETIVOS 17
1.2 ESTRUTURA 17
2 FUNDAMENTAÇÃO TEÓRICA 18
2.1 LÍNGUA BRASILEIRA DE SINAIS 18
2.2 LEAP MOTION 21
2.3 APRENDIZADO DE MÁQUINA SUPERVISIONADO 24
2.3.1 Máquinas de vetores de suporte 25
2.3.2 Modelos ocultos de Markov 31
2.4 TRABALHOS CORRELATOS 34
2.4.1 Gesture recognition library for Leap Motion 35
2.4.2 Markerless hand gesture interface based on Leap Motion controller 36
2.4.3 Reconhecimento de gestos da Língua Brasileira de Sinais através de Máquinas de Vetores de Suporte e Campos Aleatórios Condicionais Ocultos 37
2.4.4 Comparativo entre os trabalhos correlatos 38
3 DESENVOLVIMENTO 39
3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO 39
3.2 ESPECIFICAÇÃO 39
3.2.1 Casos de uso 40
3.2.2 Diagrama de pacotes 41
3.2.2.1 Pacote `Aquisição de dados` 42
3.2.2.2 Pacote `Extração de características` 43
3.2.2.3 Pacote `Treinamento` 45
3.2.2.4 Pacote `Reconhecimento` 47
3.3 IMPLEMENTAÇÃO 51
3.3.1 Técnicas e ferramentas utilizadas 51
3.3.2 Fluxo geral da aplicação 51
3.3.3 Importar sinal 52
3.3.4 Persistir dados 53
3.3.5 Buscar sinal para reconhecer 54
3.3.6 Aquisição de dados do Leap Motion 55

3.3.7 Extração de características 58
3.3.8 Treinar algoritmos 66
3.3.9 Classificação e reconhecimento 73
3.3.9.1 Reconhecimento de sinais estáticos 75
3.3.9.2 Reconhecimento de sinais dinâmicos 76
3.3.10 Operacionalidade da implementação 83
3.4 RESULTADOS E DISCUSSÕES 84
3.4.1 Experimento 01: Aquisição de dados através do Leap Motion 85
3.4.2 Experimento 02: Funções *kernel* 87
3.4.3 Experimento 03: Validação cruzada 89
3.4.4 Experimento 04: Desempenho 92
3.4.5 Experimento 05: Uso da aplicação 93
3.4.5.1 Metodologia 93
3.4.5.2 Aplicação do teste 93
3.4.5.3 Análise e interpretação dos dados coletados 94
3.4.5.3.1 Análise dos resultados da lista de tarefas 94
3.4.5.3.2 Análise quanto a usabilidade 96
3.4.6 Comparação com os trabalhos correlatos 96
4 CONCLUSÕES 98
4.1 EXTENSÕES 99
REFERÊNCIAS 100
APÊNDICE A – Diagrama de classes completo 103
APÊNDICE B – Atividades para reconhecimento de sinais estáticos e dinâmicos 108
APÊNDICE C – Formato JSON de um sinal armazenado 110
APÊNDICE D – Tutorial de uso do Leap Recorder 111
APÊNDICE E – Questionário 113

1 INTRODUÇÃO

A inteligência artificial é um assunto estudado há décadas e desde o começo as pessoas se perguntam se as máquinas são realmente capazes de pensar (SMITH et al., 2006, p. 4). Segundo Smith et al. (2006, p. 4, tradução nossa), "Ninguém pode refutar a habilidade de um computador processar lógica. Mas para muitos é incerto se uma máquina pode pensar". Durante a década de 1970 as atividades comerciais e científicas com inteligência artificial decaíram e este período foi denominado "inverno da IA" e um dos termos que nasceram neste período foi o "aprendizado de máquina" (SMITH et al., 2006, p. 17-18, tradução nossa).

O aprendizado de máquina possui três principais áreas, sendo elas: aprendizado supervisionado, aprendizado não supervisionado e aprendizado por reforço (RUSSEL; NORVIG, 2003, p. 650, tradução nossa). O aprendizado supervisionado é composto por técnicas para realizar a classificação ou análise de regressão de padrões através de exemplos de entrada e saída de uma função. Estas técnicas podem ser utilizadas para reconhecimento facial (classificação) ou até mesmo análise de valores de imóveis (análise de regressão). As técnicas de aprendizado não supervisionado são utilizadas para aprender e agrupar padrões em entradas de funções. Um uso destas técnicas seria o agrupamento de artigos de jornais por categorias. Já o aprendizado por reforço é composto pelas técnicas mais genéricas, onde ao invés de ser dito o que fazer, o algoritmo aprende através de recompensas (RUSSEL; NORVIG, 2003, p. 650, tradução nossa). Essas técnicas podem ser utilizadas na navegação de robôs, onde o robô escolhe o caminho e depois é informado se a escolha foi boa ou não.

O aprendizado de máquina supervisionado tem se mostrado interessante para a solução de vários problemas do cotidiano tais como: classificação de mensagens, avaliação de preços da bolsa de valores e mineração de dados. Desta forma, um cenário que se mostra interessante é a utilização destas técnicas com o dispositivo Leap Motion para reconhecimento de sinais da Língua Brasileira de Sinais (LIBRAS).

No Brasil, segundo o Instituto Brasileiro de Geografia e Estatística (2012, p. 30), cerca de 5,1% da população brasileira possui alguma deficiência auditiva e pode depender de uma língua de sinais. Essa porcentagem implica na necessidade de existir o ensino de LIBRAS, visto que esta também é uma língua oficial do Brasil. Entretanto, o aprendizado por crianças e adultos pode ser complicado sem uma forma de praticar e aprimorar os conhecimentos adquiridos da língua. Portanto, no ensino de LIBRAS há uma lacuna onde pode ser utilizado o estado da arte da computação para desenvolver uma aplicação para auxiliar na aprendizagem de LIBRAS.

1.1 OBJETIVOS

O objetivo deste trabalho é desenvolver uma aplicação para auxiliar no ensino-aprendizagem de LIBRAS.

Os objetivos específicos do trabalho são:

a) reconhecer sinais de LIBRAS utilizando o Leap Motion como dispositivo de entrada de dados;
b) reconhecer sinais estáticos utilizando Máquinas de Vetores de Suporte (*Support Vector Machines* - SVM);
c) reconhecer sinais dinâmicos utilizando Modelos Ocultos de Markov (*Hidden Markov Model* - HMM).

1.2 ESTRUTURA

Este trabalho está dividido em quatro capítulos. O primeiro capítulo apresenta a introdução do trabalho e os objetivos. O segundo capítulo apresenta a fundamentação teórica sobre LIBRAS, Leap Motion e aprendizado de máquina supervisionado. No terceiro capítulo é demonstrado o desenvolvimento do trabalho com requisitos, especificação, implementação, resultados e operacionalidade da aplicação. No quarto capítulo são relatadas as conclusões e também possíveis extensões.

2 FUNDAMENTAÇÃO TEÓRICA

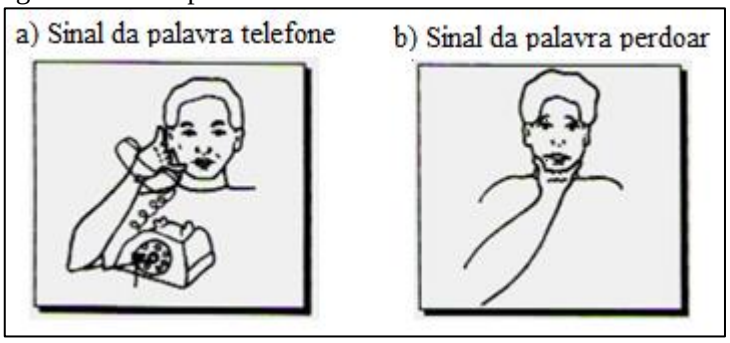
Este capítulo está organizado em quatro seções. A seção 2.1 aborda línguas de sinais e LIBRAS. Na seção 2.2 é apresentado o dispositivo Leap Motion e o seu funcionamento. A seção 2.3 apresenta os conceitos envolvendo aprendizado de máquina supervisionado seguida por SVMs na seção 2.3.1 e por HMMs na seção 2.3.2. Finalmente, na seção 2.4 são apresentados os trabalhos correlatos.

2.1 LÍNGUA BRASILEIRA DE SINAIS

O ensino de línguas de sinais teve início no século XVI com o francês Pedro Ponce de Leon que fundou uma escola de professores de surdos (CARVALHO, 2011). Mais adiante, no século XVIII o também francês Charles Michel de L'Epée criou os "sinais metódicos" em uma combinação de língua de sinais com a gramática sinalizada francesa (CARVALHO, 2011).

Diferentemente do que se pensa, as línguas de sinais não são universais e diferem entre países, ou até mesmo entre regiões de um mesmo país (ARAÚJO, 2009). Essa padronização se torna ainda mais difícil em países grandes como o Brasil, que em casos de cidades grandes é possível encontrar até certos "bairrismos" (PACHECO; ESTRUC, 2011, p. 8). A língua de sinais não é mímica, entretanto apresenta sinais icônicos e sinais arbitrários como mostra a Figura 1.

Figura 1 – Exemplo de um sinal icônico e de um sinal arbitrário



Fonte: adaptado de Pacheco e Estruc (2011, p. 14-15).

A Figura 1a apresenta um sinal icônico onde, segundo Pacheco e Estruc (2011, p. 14), os "[...] gestos fazem alusão à imagem do seu significado." A Figura 1b demonstra um sinal arbitrário que, segundo Pacheco e Estruc (2011, p. 15), "São aqueles que não mantêm nenhuma semelhança com o dado da realidade que representam."

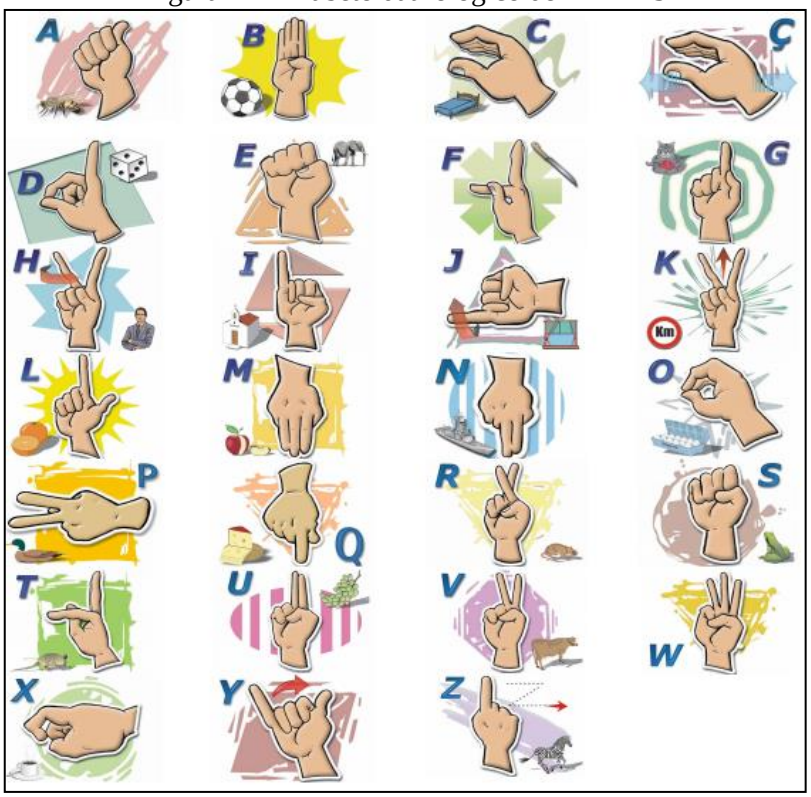
LIBRAS é um sistema linguístico de modalidade gestual-visual, baseado na Língua de Sinais Francesa e com estrutura gramatical independente da língua portuguesa brasileira. Entretanto, LIBRAS sofre influência direta da língua portuguesa brasileira por meio de

adaptações por serem línguas em contato (ALBRES, 2005, p. 1). Esta língua teve origem com o professor Ernesto Huet durante o Império de Dom Pedro II e foi reconhecida como língua oficial no Brasil pela Lei Federal nº 10.436 em 24 de abril de 2002.

LIBRAS, assim como qualquer outra língua, possui estruturas sintáticas, semânticas, morfológicas e também passa por um processo de aprendizagem (ARAÚJO, 2009). O que é conhecido como uma palavra ou item léxico em línguas oral-auditivas é denominado um sinal nas línguas de sinais (SILVA, 2011, p. 1). Os sinais são criados a partir de combinações de formas, movimentos das mãos e pontos de articulação no corpo ou no espaço onde os sinais são realizados (SILVA, 2011, p. 1). Há casos específicos onde não há um sinal que corresponde a uma palavra, como no caso dos nomes. Nesse caso as letras são sinalizadas separadamente (ARAÚJO, 2009). A soletração de palavras também é conhecida como datilologia (SILVA, 2011, p. 1).

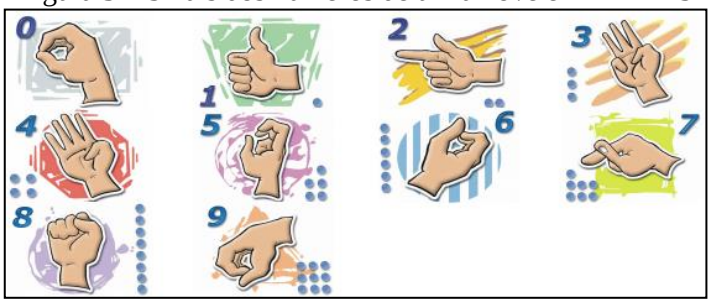
O alfabeto manual de LIBRAS (alfabeto datilológico) é composto por 27 formatos, onde cada formato corresponde a uma letra do alfabeto do português brasileiro (GESSER, 2009, p. 30). A Figura 2 apresenta o alfabeto datilológico e a Figura 3 apresenta os sinais para representar os números em LIBRAS.

Figura 2 – Alfabeto datilológico de LIBRAS



Fonte: Pacheco e Estruc (2011, p. 37).

Figura 3 – Sinais dos números de um a nove em LIBRAS



Fonte: Pacheco e Estruc (2011, p. 37).

A partir da Figura 2 e da Figura 3, é possível perceber que a maioria dos sinais numéricos e do alfabeto datilológico não possuem movimento. As exceções são os sinais das letras Ç, H, J, H, X, Y e Z que possuem movimento.

Segundo Bento (2010, p. 33), "Um sinal pode ser articulado com uma ou duas mãos. Um mesmo sinal pode ser articulado tanto com a mão direita quanto com a mão esquerda sem ocasionar mudança significativa e, portanto, não distintiva". Esta característica é possível porque os sinais são formados por unidades mínimas compostas pelos parâmetros fonológicos de configuração de mãos, ponto de articulação, movimento, orientação e expressão não-manual (BENTO, 2010, p. 36).

A configuração de mãos, segundo Bento (2010, p. 39), consiste nas "[...] formas das mãos e que podem ser da datilologia (alfabeto manual) ou demais formas feitas manualmente." Bento (2010, p. 43) também afirma que este parâmetro "É considerado um articulador primário das línguas de sinais, sendo o parâmetro mais primitivo, pois não existe sinal sem configuração de mão.".

O ponto de articulação é o local da mão configurada, podendo tocar o corpo ou estar em um espaço neutro vertical (BENTO, 2010, p. 39). De acordo com Ferreira et al. (2011, p. 29), “A execução dos sinais acontece no espaço que se situa diante do emissor, desde a linha da cintura até o alto da cabeça [...]”.

O movimento, segundo Bento (2010, p. 46), "[...] consiste no deslocamento da Configuração de Mãos, durante a realização de um determinado sinal." De acordo com Ferreira et al. (2011, p. 38), este “É um parâmetro complexo porque, durante a realização do sinal, engloba o deslocamento de uma ou ambas as mãos no espaço, abrangendo também dedos, pulso, braço e antebraço.” O movimento de um sinal ainda pode ser distinguido pela sua forma, frequência e velocidade. A forma do movimento é como as mãos e os dedos seguem suas trajetórias na execução do sinal, podendo ser extensa ou curta e movendo-se em linha reta ou curva (FERREIRA et al., 2011, p. 38). A frequência, de acordo com Ferreira et al. (2011, p. 39), é a “[...] repetição do movimento na realização do sinal, sendo que essa

repetição só acontece em alguns sinais.” Por fim, a velocidade diferencia entre sinais que requerem movimentos mais rápidos ou movimentos mais lentos e suaves (FERREIRA et al., 2011, p. 39).

Segundo Quadros e Karnopp (2004, p. 59), “Por definição, orientação é a direção para a qual a palma da mão aponta na produção do sinal”. Existem sete possíveis orientações da palma das mãos: para cima, para baixo, para o corpo, para frente, para a direita (contralateral), para a esquerda (ipsilateral) e diagonal (FERREIRA et al., 2011, p. 41).

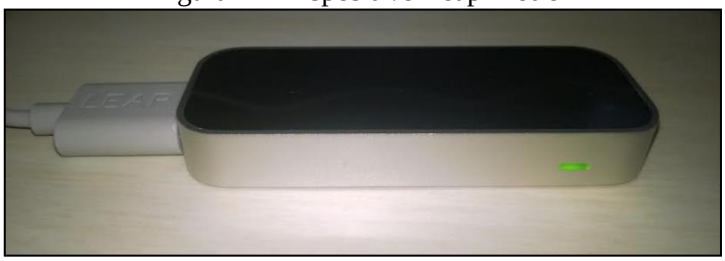
A expressão não-manual é o diferenciador que atua como complemento dos sinais manuais (BENTO, 2010, p. 39). Bento (2010, p. 52) complementa dizendo que "As expressões faciais/corporais ou não-manuais são de fundamental importância para o entendimento real do sinal, sendo que a entonação em Língua de Sinais é feita pela expressão facial."

Para Bento (2010, p. 39), "[...] as línguas de sinais apresentam como característica específica a simultaneidade na articulação dos fonemas. Assim, uma mesma Configuração de Mãos e um mesmo movimento, com locação diferente, resulta em mudança de significado, formando um par mínimo."

2.2 LEAP MOTION

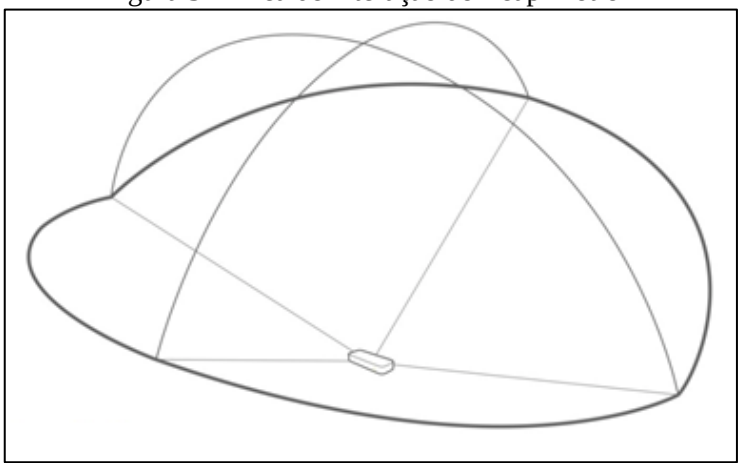
O Leap Motion (Figura 4) é um dispositivo desenvolvido pela empresa Leap Motion, Inc. e promete precisão milimétrica para detecção das mãos (WEICHERT et al., 2013).

Figura 4 – Dispositivo Leap Motion



O *hardware* do dispositivo é um tanto simples, consistindo apenas de duas câmeras estéreo e três LEDs infravermelhos, permitindo uma área de interação de aproximadamente 2,5 metros cúbicos como na Figura 5 (COLGAN, 2014).

Figura 5 – Área de interação do Leap Motion

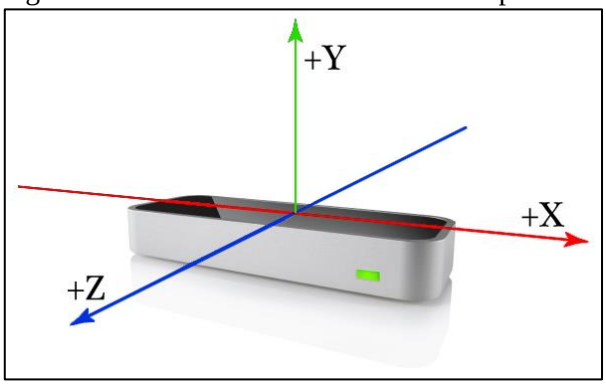


Fonte: adaptado de Colgan (2014).

A área de interação demonstrada na Figura 5 consiste em 60,960 centímetros acima do dispositivo, 60,960 centímetros ou 150º para cada lado e 60,960 centímetros ou 120º de profundidade para cada lado formando uma espécie de pirâmide invertida (COLGAN, 2014).

O Leap Motion usa um sistema de coordenadas de mão direita com origem localizada no centro acima do dispositivo, como pode ser visto na Figura 6 (LEAP MOTION INC, 2014).

Figura 6 – Sistema de coordenadas do Leap Motion



Fonte: Leap Motion Inc (2014).

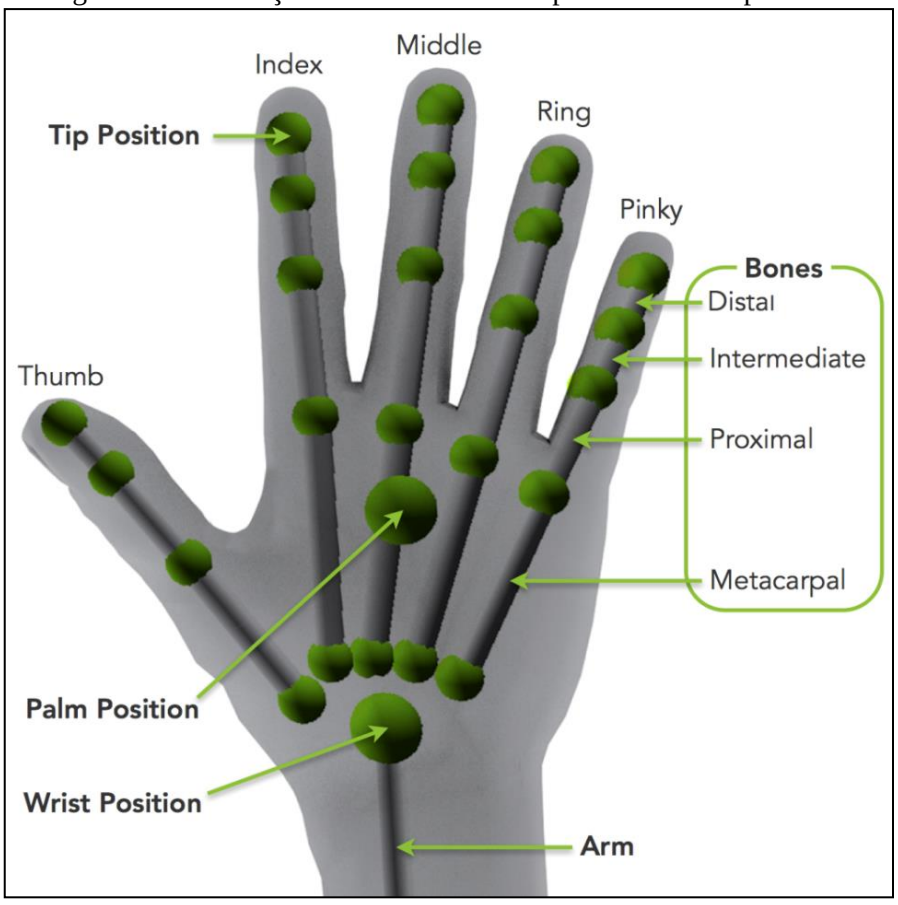
A partir da Figura 6 é possível perceber que os eixos `x` e `z` são horizontais e apresentam valores positivos e negativos, enquanto o eixo `y` é vertical e apresenta apenas valores positivos (LEAP MOTION INC, 2014).

O dispositivo lê os dados dos sensores em sua memória local e faz os ajustes necessários para então enviar os dados para o computador, onde são feitos os processamentos matemáticos (COLGAN, 2014, tradução nossa). Segundo Colgan (2014, tradução nossa), "Diferentemente do que se pensa, o Leap Motion não gera um mapa de profundidade – em vez disso ele aplica algoritmos avançados aos dados puros do sensor". Após o processamento dos dados do sensor, uma camada com algoritmos de rastreamento extrai informações como

dedos, ferramentas e infere a posição de objetos oclusos (COLGAN, 2014, tradução nossa). Também são aplicadas técnicas de filtragem para garantir coerência temporal dos dados (COLGAN, 2014, tradução nossa).

Os dados processados são enviados no formato de *frames* através de um protocolo de transporte utilizando o protocolo TCP para aplicações nativas, ou *WebSocket* para aplicações web. De acordo com Davis (2014, tradução nossa, grifo nosso), "No nível mais básico, a API do Leap Motion retorna os dados de rastreamento na forma de *frames*". Em cada *frame* é possível acessar as entidades rastreadas como mãos, dedos e ferramentas, gestos reconhecidos e também os dados puros do sensor (DAVIS, 2014, tradução nossa). Além dessas informações, a API também consegue distinguir entre mão direita e mão esquerda e informar a confiabilidade das informações retornadas (DAVIS, 2014, tradução nossa). A Figura 7 apresenta algumas informações das mãos que a API consegue retornar em cada *frame*.

Figura 7 – Informações da mão retornadas pela API do Leap Motion

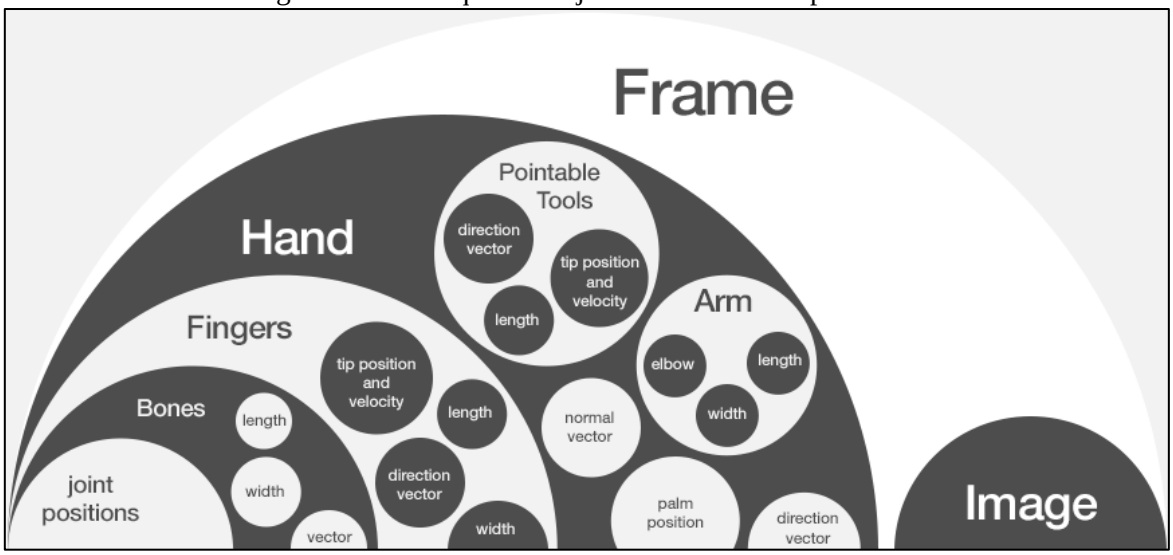


Fonte: Davis (2014).

Com base na Figura 7 fica evidente que o Leap Motion consegue entregar um esqueleto 3D de uma mão com informações dos ossos que a compõem. Além das informações do esqueleto 3D, também é possível notar que informações posicionais, como a posição da

mão, são retornadas pelo sensor. Estas informações estão organizadas na API do Leap Motion conforme mostra a Figura 8.

Figura 8 – Hierarquia de objetos da API do Leap Motion



Fonte: Davis (2014).

A Figura 8 apresenta de forma visual como as informações são retornadas pela API do Leap Motion. O ponto de partida é um *frame* que contém informações das mãos e também as imagens capturadas pelo dispositivo. A partir das mãos, é possível acessar as informações dos dedos e em seguida dos ossos que formam os dedos. Também é possível perceber quantas informações a API consegue entregar para o desenvolvedor, como por exemplo: vetores de direção para as mãos e para os dedos, vetor normal da palma das mãos, posição e velocidade da mão e dos dedos, largura e comprimento do braço e posição do cotovelo. Além das informações das mãos, a API do Leap Motion também consegue fornecer informações sobre objetos apontáveis, como canetas. Essas informações são semelhantes as informações dos dedos, por exemplo: comprimento, vetor de direção, vetor de velocidade e vetor de posição.

2.3 APRENDIZADO DE MÁQUINA SUPERVISIONADO

Desde a invenção dos computadores, sempre se imaginou se eles poderiam ser programados para aprender, se aprimorarem automaticamente com experiência (MITCHELL, 1997, p. 1). Mitchell (1997, p. 2, tradução nossa, grifo do autor) define o aprendizado de máquina da seguinte forma: "Um programa de computador é dito que **aprende** pela experiência *E* com respeito a alguma classe de tarefas *T* e medindo o desempenho *P*, se o desempenho nas tarefas *T*, medido por *P*, melhora com a experiência *E*".

O aprendizado de máquina pode ser dividido nas seguintes classes: aprendizado supervisionado, aprendizado não supervisionado e aprendizado por reforço, onde o fator que

determina a classe a ser utilizada é o tipo de *feedback* disponível para o algoritmo de aprendizado (RUSSEL; NORVIG, 2003, p. 650, tradução nossa).

Dada uma função f, as técnicas de aprendizado de máquina supervisionado tentam recuperar essa função através do aprendizado de uma função h, também conhecida como hipótese. O aprendizado desta função ocorre através da inferência sobre exemplos $(x, f(x))$, sendo x uma entrada de f e $f(x)$ o resultado de f para a entrada x (RUSSEL; NORVIG, 2003, p. 651). Segundo Russel e Norvig (2003, p. 651, tradução nossa, grifo do autor), este procedimento pode ser descrito como "Dada uma coleção de exemplos de *f*, retorne uma função *h* que aproxime *f*". Se a função f possui saídas discretas, ela é denominada uma função de classificação. Entretanto, se as saídas desta função são contínuas, então esta função é denominada uma função de regressão.

Os exemplos $(x, f(x))$ geralmente são representados utilizando uma tabela no formato atributo-valor como no Quadro 1 (SANCHES, 2003, p. 16).

Quadro 1 – Exemplos na forma de uma tabela atributo valor

	X_1	X_2	...	X_m	Y
E_1	x_{11}	x_{12}	...	x_{1m}	y_1
E_2	X_{21}	X_{22}	...	X_{2m}	Y_2
...	...	...	...	...	...
E_n	X_{n1}	X_{n2}	...	X_{nm}	Y_n

Fonte: adaptado de Sanches (2003, p. 16).

O Quadro 1 é composto por n linhas de exemplos $E = \{E_1, E_2, ..., E_n\}$ e m colunas de atributos $X = \{X_1, X_2, ..., X_m\}$, onde a combinação x_{ij} representa o j-ésimo atributo do i-ésimo exemplo (SANCHES, 2003, p. 16). Por fim o valor y_i é a classe atribuída para o exemplo E_i. Desta forma, o par de exemplos $(x, f(x))$ pode ser descrito como (E_i, y_i).

Uma boa hipótese h irá prever corretamente exemplos ainda não vistos e a dificuldade em fazer esta afirmativa é uma das dificuldades do aprendizado de máquina, pois é difícil dizer quando h é uma boa aproximação de f (RUSSEL; NORVIG, 2003, p. 651). A tentativa de recuperar f pode acabar com várias hipóteses consistentes e nesse caso é preferível a escolha da hipótese mais simples consistente com os dados.

2.3.1 Máquinas de vetores de suporte

As SVMs baseiam-se na teoria de aprendizado estatístico de Vapnik (LORENA; CARVALHO, 2007, p. 1). Essa teoria usa a ideia de um hiperplano que separa duas classes e maximiza uma margem com a maior distância possível entre o hiperplano e as instâncias em cada lado (KOTSIANTIS, 2007, p. 260, tradução nossa). Segundo Silva (2008, p. 37, grifo do autor), "[...] tal Teoria orienta a busca pela melhor solução para um dado problema de

aprendizado através da minimização não só do erro de treinamento, mas também da complexidade do modelo obtido, o que resulta em um dos principais pontos fortes das SVMs: **a alta capacidade de generalização**."

As SVMs trazem a vantagem de terem um aprendizado convexo, onde os mínimos local e global coincidem (SOUZA, 2013, p. 108). Além disso, segundo Kotsiantis (2007, p. 261, tradução nossa), o "[...] modelo de complexidade de uma SVM não é afetado pelo número de características encontradas nos dados de treinamento (o número de vetores de suporte selecionados pelo algoritmo de aprendizado de SVM é geralmente pequeno)." No entanto, uma dificuldade ao trabalhar com SVMs é que todos os dados de treinamento devem estar presentes antes do aprendizado começar (SOUZA, 2013, p. 108).

As SVMs lineares utilizam como limite de decisão um hiperplano separando duas regiões no espaço (TAHIM, 2010, p. 1). Dessa forma, os dados de um conjunto de treinamento `X` com `n` dados são classificados como `+1` ou `-1` (LORENA; CARVALHO, 2007, p. 53). O Quadro 2 mostra a equação de um hiperplano para separar um conjunto `X` linearmente em duas regiões: `w • x + b > 0` e `w • x + b < 0`.

Quadro 2 – Equação de um hiperplano

$$f(x) = w \cdot x + b = 0$$

Fonte: Lorena e Carvalho (2007, p. 53).

De acordo com Lorena e Carvalho (2007, p. 53), "[...] `w • x` é o produto escalar entre os vetores `w` e `x`, `w ∈ X` é o vetor normal ao hiperplano descrito [...]". Dada a equação do hiperplano, é possível expressar uma função de decisão como a do Quadro 3.

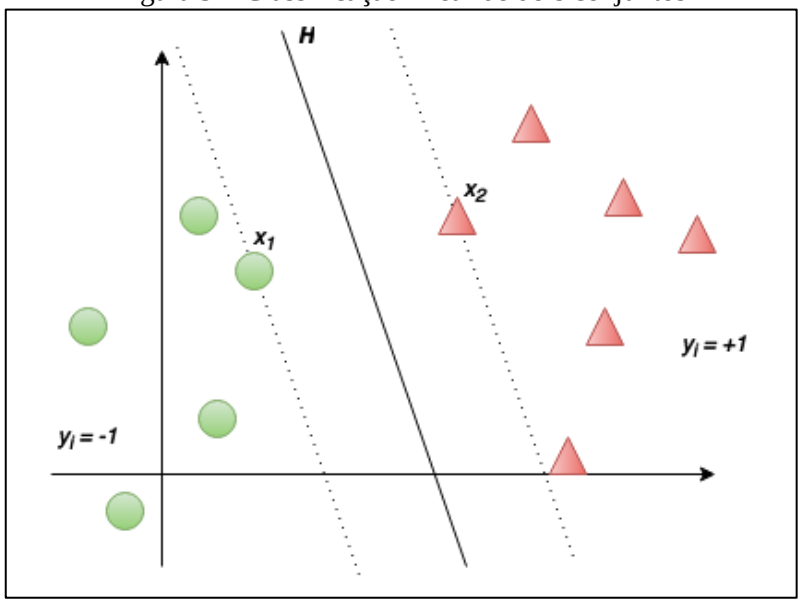
Quadro 3 – Função de decisão sobre um hiperplano

$$g(x) = sgn\big(f(x)\big) = \begin{cases} +1 \; se \; w \cdot \mathrm{x} + \mathrm{b} > 0 \\ -1 \; se \; w \cdot \mathrm{x} + \mathrm{b} < 0 \end{cases}$$

Fonte: Lorena e Carvalho (2007, p. 53).

A função de decisão do Quadro 3 tem apenas dois resultados possíveis, os rótulos `+1` ou `-1`. Portanto, ela classifica vetor `x` como um destes rótulos de acordo com o resultado da equação `w • x + b`. Dada a equação do hiperplano e a função de decisão, é possível representar o modelo de classificação linear através de um gráfico como na Figura 9.

Figura 9 – Classificação linear de dois conjuntos

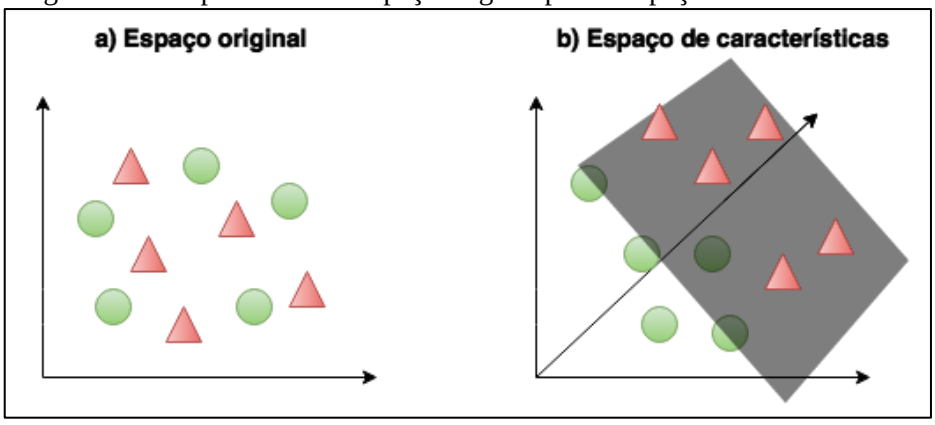


Fonte: adaptado de Tahim (2010, p. 3).

No exemplo da Figura 9 é possível notar que o hiperplano H maximiza a margem entre os dois conjuntos. Os pontos x_1 e x_2 são os pontos de vetores de suporte e a solução é representada como uma combinação linear apenas desses pontos, os outros pontos são ignorados (KOTSIANTIS, 2007, p. 261, tradução nossa). Devido a escolha destes poucos vetores de suporte, segundo Kotsiantis (2007, p. 261, tradução nossa), as "[...] SVMs são bem adaptadas para lidar com tarefas de aprendizado onde o número de características é grande com respeito ao número de instâncias de treinamento."

Segundo Bonesso (2013, p. 32), "[...] se os dados de treinamento não forem linearmente separáveis, o hiperplano obtido pelo classificador pode ter baixo poder de generalização, mesmo que o hiperplano seja determinado de maneira ótima." Para resolver esse problema é possível aplicar o truque do *kernel*, que mapeia o espaço original para um espaço com mais dimensões chamado de espaço de características como na Figura 10 (BONESSO, 2013, p. 32).

Figura 10 – Mapeamento do espaço original para o espaço de características



A Figura 10 demonstra um exemplo de uma função *kernel*, que é o mapeamento de vetores no espaço original 2D (Figura 10a) para o espaço de características 3D (Figura 10b). Algumas escolhas comuns de funções *kernel* são apresentadas no Quadro 4.

Quadro 4 – Funções *kernel* comuns

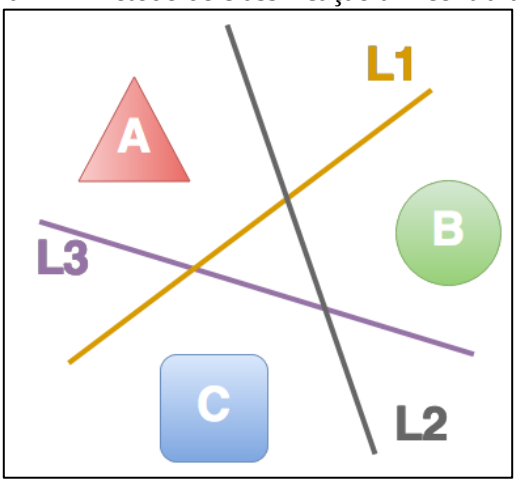
Função	Nome	Parâmetros
$k(x,z) = x \cdot z + c$	Linear	
$k(x,z) = (x \cdot z + c)^d$	Polinomial	Grau polinomial `d`, Constante de homogeneidade `c`.
$k(x,z) = \exp\{-\frac{1}{2\sigma^2} \parallel x - z \parallel^2\}$	Gaussiano	Parâmetro de largura σ^2

Fonte: adaptado de Souza (2013, p. 117).

As SVMs são baseadas em problemas binários de classificação (SOUZA, 2013, p. 108). Essa abordagem pode ser estendida para problemas de múltiplas classes através dos métodos um-contra-todos e um-contra-um (FLACH, 2012, p. 83).

No método um-contra-todos são treinados `k` classificadores binários, onde o primeiro separa a classe C_1 das classes C_2, C_3, ... C_n, o segundo separa a classe C_2 das classes C_1, C_3, ..., C_n, e assim por diante (FLACH, 2012, p. 83, tradução nossa). No entanto, um problema deste método seria a necessidade de treinar `k` classificadores no mesmo conjunto de dados de treinamento aumentando linearmente o tempo de aprendizado e decisão (SOUZA, 2013, p. 120). A Figura 11 apresenta um exemplo do método um-contra-todos.

Figura 11 – Método de classificação um-contra-todos



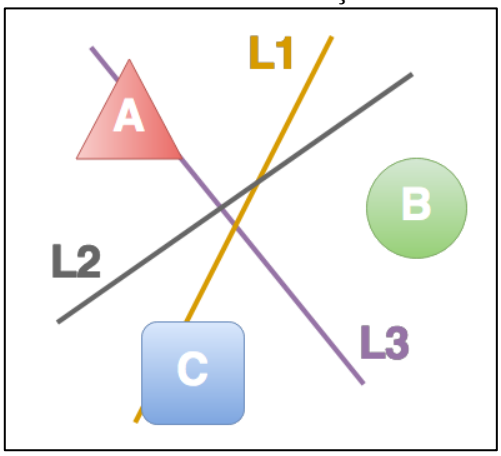
Fonte: adaptado de Souza (2013, p. 120).

A partir da Figura 11 é possível perceber que o classificador treinado para a classe `A` separa a classe `A` das demais classes através da linha `L1`. O mesmo ocorre para a classe `B` e para a classe `C`, que separam as devidas classes das demais através das linhas `L2` e `L3`, respectivamente.

O método um-contra-um decompõe o problema em todos os possíveis subproblemas de classificação par-a-par (SOUZA, 2013, p. 121). Aproveitando a simetria da matriz de

decisão, é possível descartar decisões redundantes e desnecessárias e considerar apenas `k(k - 1)/2` classificadores binários (SOUZA, 2013, p. 121). Esta solução pode parecer mais custosa inicialmente, no entanto os `k(k - 1)/2` classificadores são significativamente menores que os `k` classificadores do método um-contra-todos, além de terem um custo de aprendizado reduzido devido a incorporação de dados de apenas duas classes em cada classificador (SOUZA, 2013, p. 121). A Figura 12 demonstra um exemplo do método um-contra-um e a Figura 13 apresenta um exemplo da matriz de decisão.

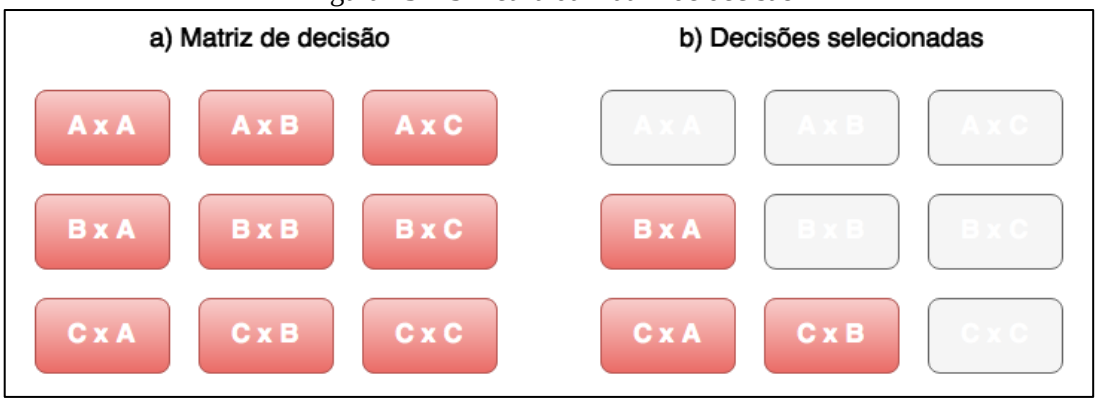
Figura 12 – Método de classificação um-contra-um



Fonte: adaptado de Souza (2013, p. 120).

O exemplo da Figura 12 demonstra como as classes são separadas em problemas binários. Por exemplo, a classificação entre a classe `A` e a classe `C` ocorre pela linha `L2`, assim como a classificação entre as classes `A` e `B` ocorre pela linha `L1` e por fim a classificação entre a classe `B` e `C` ocorre pela linha `L3`.

Figura 13 – Simetria da matriz de decisão



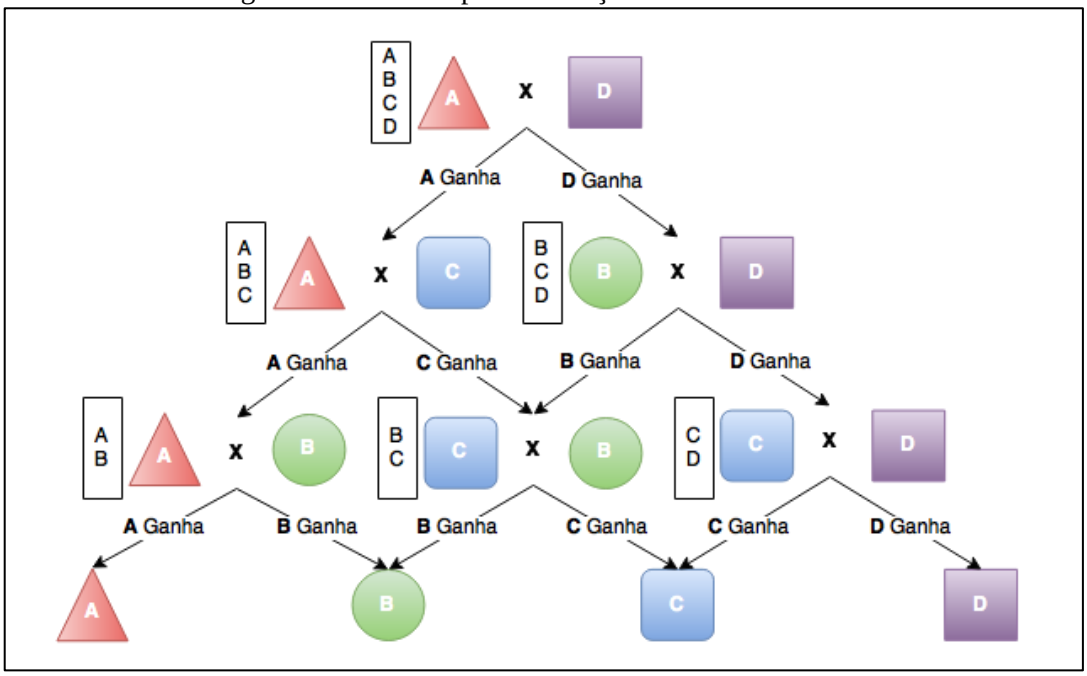
Fonte: adaptado de Souza (2013, p. 121).

A matriz de decisão apresentada na Figura 13a possui decisões redundantes, como, por exemplo, a decisão `A x C` que é semelhante a decisão `C x A`. A partir da Figura 13b é possível perceber que apenas algumas decisões são necessárias, podendo eliminar as decisões redundantes.

Uma abordagem para combinar os classificadores do método um-contra-um é a decisão por votação. Nesta abordagem todos os classificadores são questionados paralelamente e incrementam um painel de votação. No final a classe com mais votos é selecionada para a amostra apresentada (SOUZA, 2013, p. 122). Esta abordagem ainda pode apresentar problemas, visto que pode existir um número alto de subproblemas a serem considerados (SOUZA, 2013, p. 122).

Outra abordagem é uma estratégia de eliminação baseada em Grafos de Decisão Acíclicos Dirigidos (*Decision Directed Acyclic Graphs* - DDAG) onde uma das classes é eliminada a cada processo de classificação, como mostrado na Figura 14 (SOUZA, 2013, p. 122). Esta abordagem consegue escalar linearmente chegando a decisão final da classe em `k - 1` passos, que acaba se tornando interessante para aplicações em tempo real (SOUZA, 2013, p. 122).

Figura 14 – Decisão por eliminação baseada em DDAG



Fonte: adaptado de Souza (2013, p. 123).

A árvore de decisão apresentada na Figura 14 demonstra como as classes podem ser eliminadas em cada processo de classificação. Por exemplo, se a decisão `A x D` classificar para a classe `A`, a classe `D` não é mais considerada nos demais processos de classificação.

De acordo com Souza (2013, p. 118) "[...] o problema de aprendizado em SVMs se resume a solução de um problema de otimização quadrática [...]". Devido a essa característica, qualquer otimizador quadrático pode ser utilizado para a resolução do aprendizado (SOUZA, 2013, p. 118). No entanto, métodos específicos como o Otimização Sequencial Mínima (*Sequential Minimal Optimization* - SMO) são capazes de explorar

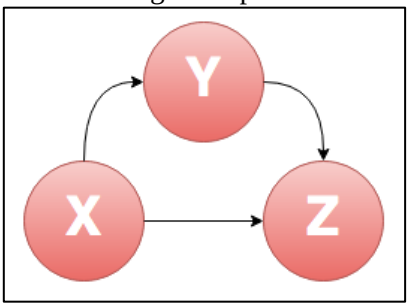
características específicas de SVMs e diminuir o custo computacional (SOUZA, 2013, p. 118). O SMO reduz o problema de otimização quadrática em pequenos problemas de otimização quadrática que são resolvidos analiticamente (PLATT, 1998, p. 1). O SMO consegue escalar em tempo linear ou quadrático em relação ao conjunto de treinamento por evitar otimizações quadráticas e computação de matrizes (PLATT, 1998, p. 1).

2.3.2 Modelos ocultos de Markov

Modelos que incorporam estruturas ocultas já provaram ser vantajosos para modelar sequências de gestos (WANG et al., 2006, p. 1). Um HMM é um modelo probabilístico que permite eventos observados e eventos ocultos (JURAFSKY; MARTIN, 2008, p. 211).

Um dos principais modelos probabilísticos são os modelos geradores que modelam a probabilidade $P(X,Y)$ (FLACH, 2012, p. 262). Os modelos geradores podem ser visualizados como modelos gráficos probabilísticos. Segundo Jordan (2010, p. 1, tradução nossa), "Um modelo gráfico é uma família de distribuições de probabilidades em termos de um grafo dirigido ou não dirigido". Nesse grafo, cada nó representa uma variável aleatória e as arestas, que ligam os nós, representam o relacionamento probabilístico entre as variáveis (BISHOP, 2006, p. 360). A Figura 15 demonstra um exemplo de um modelo gráfico probabilístico dirigido onde é possível perceber que a variável Z está condicionada nas variáveis X e Y, e a variável Y está condicionada na variável X.

Figura 15 – Modelo gráfico probabilístico dirigido



Fonte: adaptado de Bishop (2006, p. 361).

Um HMM modela a probabilidade conjunta $P(X,Y)$ de um conjunto de observações X e uma sequência de estados ocultos Y (SOUZA, 2013, p. 129). Este modelo apresenta a propriedade de Markov, onde um estado depende apenas do estado anterior e da probabilidade de transição entre esses estados (NOWIKI et al., 2014, p. 33). O Quadro 5 apresenta a probabilidade de ocorrência de observações dada uma sequência de estados ocultos modelada por um HMM (SOUZA, 2014).

Quadro 5 – Probabilidade de uma sequência de observações em um HMM

$$P(X,Y) = \prod_{t=1}^{T} \underbrace{P(Y_t|Y_{t-1})}_{A} \underbrace{P(X_t|Y_t)}_{B}$$

Fonte: adaptado de Souza (2014).

A partir do Quadro 5 é possível notar que um HMM pode ser visualizado como um modelo gráfico dirigido dadas as probabilidades condicionais $P(Y_t|Y_{t-1})$ e $P(X_t|Y_t)$. A probabilidade $P(Y_t|Y_{t-1})$ é probabilidade de estar no estado Y_t dado que o estado anterior era o estado Y_{t-1} (SOUZA, 2014). A probabilidade $P(X_t|Y_t)$ é a probabilidade de observar X_t dado o estado Y_t (SOUZA, 2014). Para computar essas probabilidades são usadas as matrizes A e B (SOUZA, 2014). A matriz A é a matriz de transição de estados que representa a transição do estado Y_{t-1} para o estado Y_t (JURAFSKY; MARTIN, 2008, p. 211). A matriz B é a matriz de probabilidades de observação que representa a observação de X_t pelo estado Y_t (JURAFSKY; MARTIN, 2008, p. 211). O Quadro 6 demonstra o exemplo da matriz de transições e da matriz de observações para o HMM da Figura 16.

Quadro 6 – Matriz de transição e matriz de observação

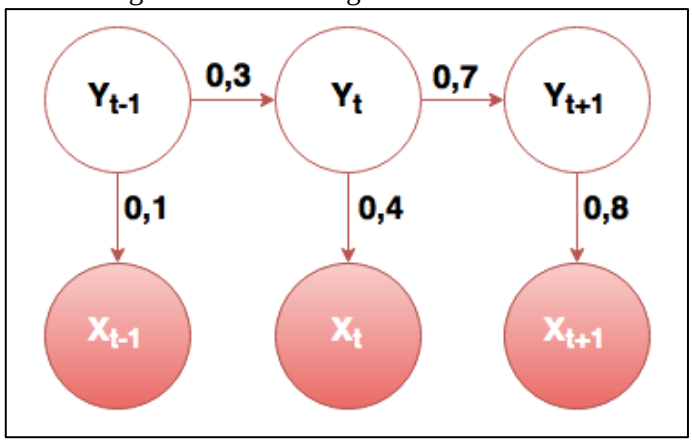
Matriz de transições

	Y_{t-1}	Y_t	Y_{t+1}
Y_{t-1}	0	0,3	0
Y_t	0	0	0,7
Y_{t+1}	0	0	0

Matriz de observações

	X_{t-1}	X_t	X_{t+1}
Y_{t-1}	0,1	0	0
Y_t	0	0,4	0
Y_{t+1}	0	0	0,8

Figura 16 – Modelo gráfico de um HMM



Fonte: adaptado de Souza (2013, p. 132).

A partir do Quadro 6 e da Figura 16 é possível perceber que as transições que não existem entre os estados ocultos são representadas com o valor 0. Por exemplo, não existe a transição entre os estados Y_{t-1} e Y_{t+1} e, portanto, na matriz de transições o valor é 0. A mesma observação pode ser feita para a matriz de observações, pois as observações que não existem também são representadas com o valor 0. Por exemplo, o estado Y_t não emite a observação X_{t+1} e, portanto, o valor na matriz de observações é 0.

A partir da Figura 16 também é possível identificar a topologia de transição de estados de um HMM. Nesse exemplo a topologia utilizada é esquerda-para-direita, também conhecida como Bakis (JURAFSKY; MARTIN, 2008, p. 212). Esta topologia pode ser utilizada para analisar sequências de acordo com o tempo porque não permite voltar para estados anteriores, como ir do estado Y_t para o estado Y_{t-1} (SOUZA, 2014).

O modelo de um HMM pode ser definido pela tupla $\lambda = (n, A, B, \pi)$ (SOUZA, 2014). O parâmetro n é um inteiro que representa o número total de estados do HMM (SOUZA, 2014). Os parâmetros A e B representam, respectivamente, as matrizes de transição e observação. Finalmente, o parâmetro π é um vetor que representa a probabilidade inicial de cada estado (SOUZA, 2014).

O aprendizado de um HMM consiste em aprender as matrizes A e B dada uma sequência de observações X e um conjunto de estados possíveis Y (JURAFSKY; MARTIN, 2008, p. 221). De acordo com Jurafsky e Martin (2008, p. 221, tradução nossa, grifo do autor) "O algoritmo padrão para treinamento de um HMM é o algoritmo **forward-backward**, ou **Baum-Welch** [...]". Este é um algoritmo de aprendizado não supervisionado porque não requer que os estados ocultos Y estejam associados às observações X durante o treinamento (SOUZA, 2014).

O algoritmo Baum-Welch inicia com probabilidades de transição e observação e deriva melhores probabilidades iterativamente (JURAFSKY; MARTIN, 2008, p. 222). De acordo com Nowiki et al. (2014, p. 35, tradução nossa) esse procedimento é repetido "[...] até que o nível desejado de convergência é alcançado." Os novos valores das matrizes de transição e observação são calculados utilizando as probabilidades *forward* e *backward* (NOWIKI et al., 2014, p. 34-35). *Forward* representa a probabilidade de terminar no estado Y_i depois das primeiras k observações utilizando $P(Y_i | X_{1:k})$ para todos os estados Y (NOWIKI et al., 2014, p. 35). *Backward* representa a probabilidade de observar as observações $X_{k+1:t}$ a partir do estado Y_i utilizando $P(X_{k+1:t} | Y_i)$.

Os HMMs são capazes de devolver a probabilidade $P(X|\lambda)$ indicando que o modelo λ tenha gerado a sequência de observações X (SOUZA, 2013, p. 127). Dessa forma, é possível criar um modelo λ_i para cada uma das c classes do problema de classificação e treiná-los para reconhecer cada uma dessas classes separadamente (SOUZA, 2013, p. 129). Ao criar um modelo λ_i para cada classe $\omega_i \in \Omega$ com $i \leq c$, é possível computar a probabilidade de cada modelo para uma nova sequência X (SOUZA, 2010). De acordo com Souza (2010, tradução nossa), "Como cada modelo é especializado em uma determinada classe, o modelo com maior

probabilidade pode ser utilizado para determinar a classe mais provável para a nova sequência [...]“. O critério de máxima verossimilhança pode ser utilizado para obter a classe ω_i com a maior probabilidade para a nova sequência X, conforme o Quadro 7 (SOUZA, 2013, p. 129).

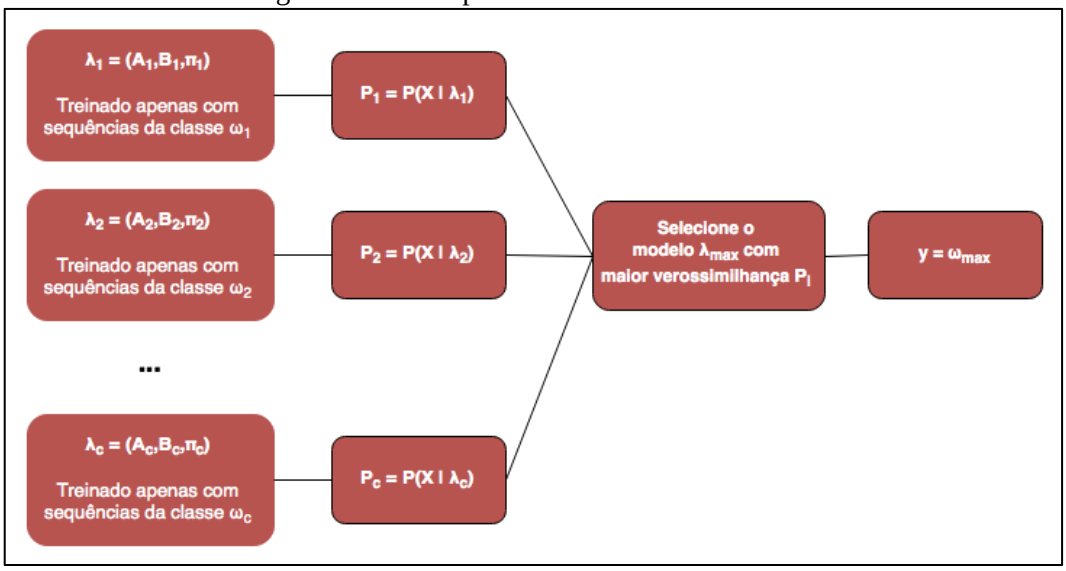
Quadro 7 – Critério de máxima verossimilhança

$$\hat{y} = \underset{\omega_i \epsilon \Omega}{\operatorname{argmax}} P(\omega_i|X)$$

Fonte: adaptado de Souza (2013, p. 129).

A partir do Quadro 7 é possível perceber que a classe ω_i selecionada para a sequência X será a classe que maximiza a probabilidade $P(\omega_i|X)$. A Figura 17 ilustra esse conceito.

Figura 17 – Exemplo de um classificador HMM



Fonte: adaptado de Souza (2013, p. 128).

A partir da Figura 17 é possível perceber que são criados c modelos λ_i e que cada modelo é treinado apenas com sequências de uma determinada classe ω_i. Com esses modelos treinados é possível classificar qual classe ω_i gera a maior probabilidade para uma nova sequência X.

2.4 TRABALHOS CORRELATOS

Esta seção tem como objetivo apresentar alguns trabalhos relacionados ao reconhecimento de gestos utilizando extração de características e algoritmos de classificação. Dentre os trabalhos selecionados estão: o trabalho de Souza (2013) com o objetivo do reconhecimento de sinais de LIBRAS utilizando o sensor Microsoft Kinect, e os trabalhos de Avola et al. (2014) e Nowiki et al. (2014) que tem o objetivo de reconhecer gestos utilizando o Leap Motion.

2.4.1 Gesture recognition library for Leap Motion

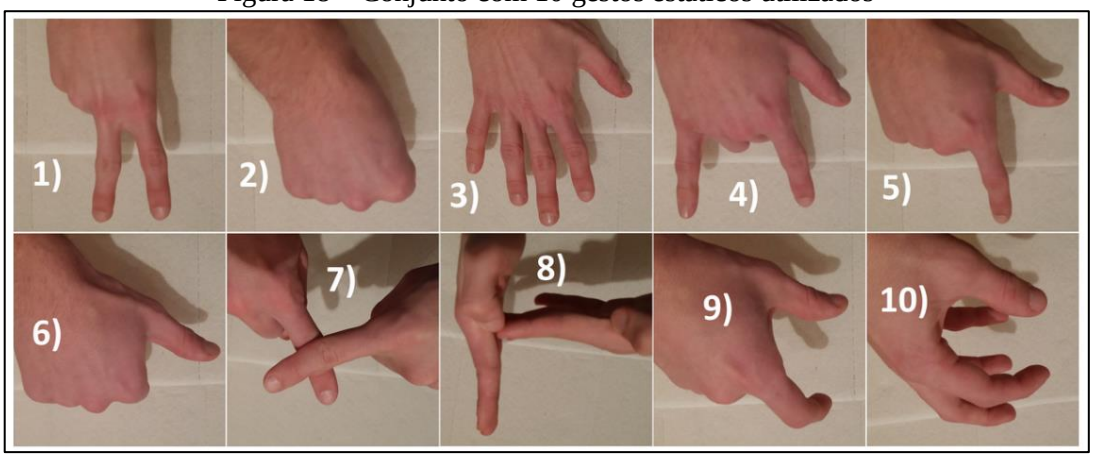
Nowiki et al. (2014) apresenta o desenvolvimento de uma biblioteca para reconhecimento de gestos estáticos e dinâmicos para o dispositivo Leap Motion. Foram utilizadas SVMs para o reconhecimento de gestos estáticos e HMMs para reconhecimento de gestos dinâmicos.

Segundo Nowiki el al. (2014, p. 19, tradução nossa), "As máquinas de vetores de suporte foram escolhidas porque existe uma fundamentação matemática sólida apoiando a simples ideia de maximização da margem entre classes". Como implementação de SVM foi utilizada a biblioteca libSVM, entretanto, devido a problemas de desempenho durante o treinamento para classificação dos gestos, também foram feitos testes com a biblioteca libLinear. A troca da biblioteca melhorou o desempenho do treinamento, que passou de aproximadamente 12 horas, para 5 segundos com 5000 exemplos. Entretanto, a troca de biblioteca implicou em uma perda de aproximadamente 5% a 17% na precisão do reconhecimento.

Os modelos ocultos de Markov foram implementados de forma que cada estado sempre tem uma transição para ele mesmo e para o próximo estado. Segundo Nowiki el al. (2014, p. 35, tradução nossa), "as transições para si mesmo foram utilizadas para modelar diferentes velocidades dos gestos e permite conseguir um sistema mais robusto."

A biblioteca atingiu 99% de precisão no reconhecimento de cinco gestos estáticos e 85% de precisão para o reconhecimento de 10 gestos estáticos. Na tarefa de reconhecimento de gestos dinâmicos, a precisão foi de 85% para um conjunto com seis gestos dinâmicos. A Figura 18 demonstra o conjunto de 10 gestos estáticos utilizado.

Figura 18 – Conjunto com 10 gestos estáticos utilizados



Fonte: Nowiki et al. (2014, p. 22).

Como pode ser observado na Figura 18, o conjunto de gestos estáticos utilizado é diversificado. É importante notar que o segundo e o sexto gesto são semelhantes, onde apenas

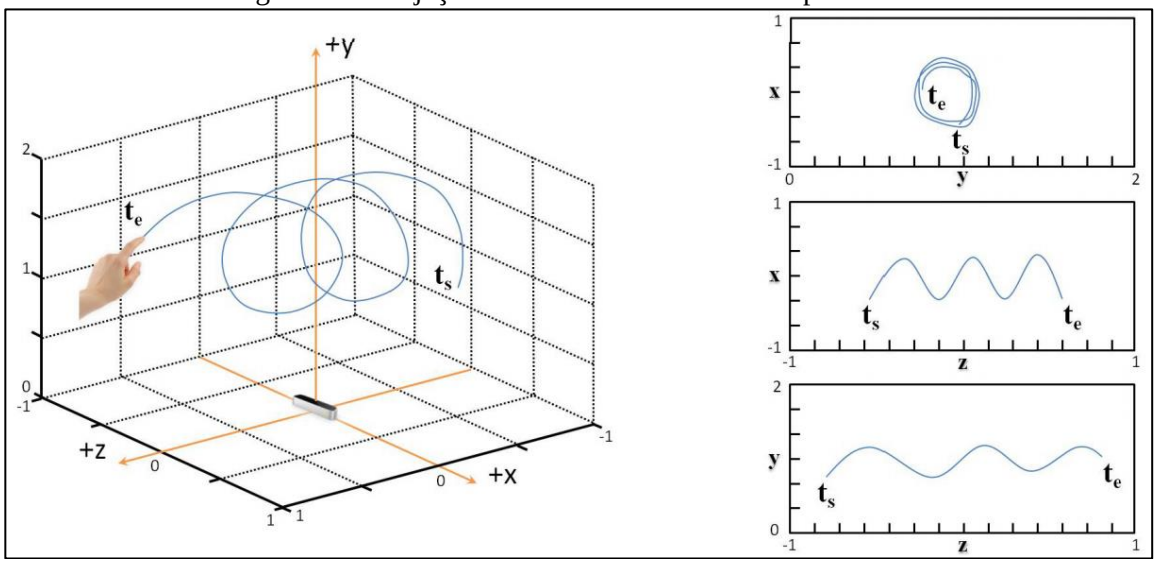
a posição do dedão é diferente. Também é possível observar que foram utilizados gestos com duas mãos, como o sétimo e o oitavo gesto. Outro ponto sobre o sétimo gesto é que ele gera um oclusão do ponto de vista do Leap Motion que pode dificultar a classificação. Finalmente, também é possível citar que o nono e o décimo gesto possuem configurações um pouco mais complexas porque os dedos estão curvados.

2.4.2 Markerless hand gesture interface based on Leap Motion controller

Avola et al. (2014) apresenta um *framework* em desenvolvimento para o reconhecimento de gestos utilizando o Leap Motion. O Leap Motion foi utilizado por satisfazer os requisitos técnicos como precisão e rastreamento da mão em tempo real.

O *framework* possui uma camada chamada *Data Pre-Processing Layer* que captura os *frames* enviados pelo Leap Motion e verifica se o *frame* possui ao menos uma mão ou objeto apontável. Posteriormente as informações espaciais 3D e temporais da palma da mão e dos dedos são projetadas nos planos `(x, y)`, `(x, z)`, `(y, z)` como pode ser visto na Figura 19.

Figura 19 – Projeção de um desenho 3D em três planos 2D



Fonte: adaptado de Avola et al. (2014, p. 264).

A Figura 19 demonstra um desenho no espaço 3D e suas respectivas representações nos planos 2D `(x, y)`, `(x, z)`, `(y, z)`.

Além da camada de pré-processamento, o *framework* possui uma camada chamada *Feature Extraction and Recognition Layer*. Esta camada faz o reconhecimento de um gesto utilizando informações como envoltória convexa, maior triângulo, maior quadrilátero, retângulo delimitador, razão dos ângulos e perímetro do esboço.

De acordo com Avola et al. (2014, p. 265, tradução nossa), “[...] a ideia de transformar informações 3D espaciais e temporais em um conjunto de arestas 2D é simples e efetivo. O

uso de um algoritmo para reconhecimento de desenho a mão livre para classificar as arestas funciona corretamente”.

2.4.3 Reconhecimento de gestos da Língua Brasileira de Sinais através de Máquinas de Vetores de Suporte e Campos Aleatórios Condicionais Ocultos

Souza (2013) desenvolveu uma ferramenta para reconhecimento de sinais estáticos, dinâmicos e soletração de LIBRAS. Para isto, foram utilizadas SVMs para reconhecimento de sinais estáticos e Campos Aleatórios Ocultos Condicionais (*Hidden Conditional Random Fields* - HCRF) para reconhecimento de sinais dinâmicos. A ferramenta foi desenvolvida em duas camadas. A primeira camada extrai informações como: configuração das mãos, face e posição das mãos em relação à face. A segunda camada é responsável pela contextualização e classificação do sinal.

Para treinamento do reconhecimento de sinais estáticos foram utilizadas 300 amostras de 46 poses fundamentais de LIBRAS, totalizando 13.800 amostras. Para treinamento do reconhecimento de sinais dinâmicos foram utilizados 939 quadros de 13 palavras do vocabulário de LIBRAS, totalizando 139.154 quadros. Segundo Souza (2013, p. 155), "As sequências foram divididas em 10 conjuntos mutuamente exclusivos em antecipação ao uso de validação cruzada".

Os experimentos para reconhecimento de sinais estáticos utilizando SVMs utilizaram as funções *kernel* gaussianas, quadráticas e lineares. O desempenho dos classificadores, segundo Souza (2013, p. 163), foi medido "em termos do coeficiente Kappa (k) de Cohen, o número total de vetores de suporte necessários para a estratégia de votação e o número médio de avaliações de vetores de suporte encontrados pelo caminho de votação do DDAG". O classificador escolhido como tendo o melhor desempenho foi o classificador DDAG composto por SVMs lineares. Esse classificador foi escolhido devido ao aprendizado convexo e por possuir tempo de avaliação constante, visto que a avaliação não depende do número de vetores de suporte em cada máquina.

Para o reconhecimento de sinais dinâmicos foram feitos experimentos com HMMs e HCRFs em conjunto de redes neurais artificiais e SVMs. Neste caso foram utilizadas funções de natureza discreta e contínua para criar e aprender os modelos propostos. A combinação escolhida para reconhecimento de sinais dinâmicos foi de SVMs em DDAG e HCRFs. Segundo Souza (2013, p. 180), “Quando alimentados com informações adequadas fornecidas através de uma SVM quadrática, os modelos discriminativos baseados em HCRFs apresentam até 19% de ganho quando comparados a seus primos gerativos baseados em HMMs”.

2.4.4 Comparativo entre os trabalhos correlatos

O Quadro 8 apresenta um comparativo entre os trabalhos correlatos evidenciando algumas características propostas neste trabalho.

Quadro 8 – Comparativo entre os trabalhos correlatos

Características / trabalhos correlatos	Nowiki et al. (2014)	Avola et al. (2014)	Souza (2013)
Leap Motion	Sim	Sim	Não
Reconhecimento de gestos dinâmicos	Através de HMM	Através de um algoritmo de reconhecimento de desenho a mão livre	Através de HCRF
Reconhecimento de gestos estáticos	Através de SVM	Não informado	Através de SVM
Reconhecimento de sinais de LIBRAS	Não	Não	Sim

A partir do Quadro 8 é possível perceber que há interesse no uso do Leap Motion para o reconhecimento de gestos, tanto dinâmicos quanto estáticos. No entanto, o uso não foi focado no reconhecimento de sinais de LIBRAS. Outro ponto a ser observado é o uso de SVMs como classificadores de sinais e gestos estáticos, que foram utilizados nos trabalhos de Souza (2013) e Nowiki et al. (2014). Também é possível perceber o uso de modelos probabilísticos como HMMs e HCRFs para reconhecer sinais dinâmicos. Além dos modelos probabilísticos, também foi utilizado um algoritmo de classificação de desenho livre no trabalho de Avola et al. (2014). Este algoritmo foi utilizado porque um gesto dinâmico pode ser considerado um desenho, no entanto no trabalho não há referências quanto ao uso desta técnica para o reconhecimento de gestos estáticos.

3 DESENVOLVIMENTO

Neste capítulo é apresentado o desenvolvimento da aplicação. A seção 3.1 apresenta os requisitos da aplicação desenvolvida. Na seção 3.2 está a especificação contendo os diagramas da parte principal da aplicação. Na seção 3.3 está descrita a implementação das principais partes da aplicação, assim como as ferramentas utilizadas e a operacionalidade da aplicação. Por fim, na seção 3.4 são apresentados os testes realizados e os resultados obtidos.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

A aplicação de ensino-aprendizagem proposta tem como requisitos:

a) utilizar o dispositivo Leap Motion com a biblioteca LeapJS para entrada de dados através de um navegador web (Requisito Não-Funcional - RNF);
b) exibir um modelo 3D da mão do usuário e do sinal que deve ser reproduzido em uma interface gráfica com o Leap Motion (Requisito Funcional - RF);
c) utilizar a biblioteca Three.js com o *plugin* LeapJS RiggedHand para exibir os modelos 3D das mãos (RNF);
d) reconhecer sinais estáticos e dinâmicos de LIBRAS utilizando os algoritmos SVM e HMM (RF);
e) utilizar a linguagem C# e o *framework* Accord.NET que contém a implementação dos algoritmos SVM e HMM (RNF);
f) permitir que seja possível treinar os algoritmos de reconhecimento de sinais (RF);
g) prover *feedback* ao usuário informando que o sinal foi reconhecido corretamente (RF);
h) desenvolver a aplicação em uma arquitetura cliente-servidor (RNF);
i) utilizar o *framework* ASP.NET WebAPI para envio de dados do navegador para o servidor (RNF);
j) utilizar o *framework* ASP.NET SignalR para comunicação do navegador com o servidor em tempo real através de *WebSockets* (RNF).

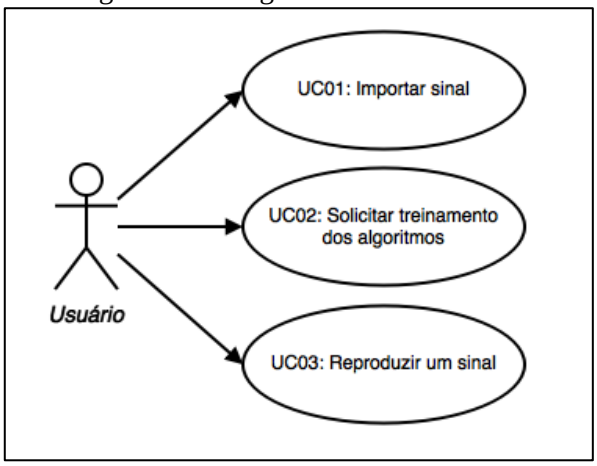
3.2 ESPECIFICAÇÃO

A especificação da aplicação foi desenvolvida de acordo com os diagramas da *Unified Modeling Language* (UML) utilizando a ferramenta draw.io.

3.2.1 Casos de uso

Nesta seção estão descritos os casos de uso referentes à Figura 20. A aplicação possui três cenários e apenas um ator, o Usuário.

Figura 20 – Diagrama de casos de uso



No caso de uso `UC01: Importar sinal` o usuário pode adicionar novos sinais para serem reconhecidos utilizando a aplicação. O caso de uso `UC02: Solicitar treinamento dos algoritmos` é onde o usuário inicia o treinamento dos algoritmos com os sinais adicionados. O caso de uso `UC03: Reproduzir um sinal` é o principal caso de uso da aplicação, onde o usuário reproduz o sinal e a aplicação faz o reconhecimento. O detalhamento dos casos de uso pode ser visualizado no Quadro 9, Quadro 10 e Quadro 11.

Quadro 9 – UC01 Importar sinal

Número	01
Caso de Uso	Importar sinal
Ator	Usuário
Pré-condições	1. Ter um sinal gravado através do Leap Recorder; 2. Estar com o Leap Motion conectado.
Cenário Principal	1. O Usuário seleciona a aba Importar sinal; 2. O Usuário informa os dados do sinal; 3. O Usuário verifica se o sinal importado está correto pela pré-visualização; 4. O Usuário salva os dados do sinal; 5. O Usuário é notificado que o sinal foi importado.

Quadro 10 – UC02 Solicitar treinamento dos algoritmos

Número	02
Caso de Uso	Solicitar treinamento dos algoritmos
Ator	Usuário
Pré-condições	1. Ter importado ao menos um sinal.
Cenário Principal	1. O Usuário clica em treinar; 2. O Usuário é notificado que o treinamento foi finalizado.

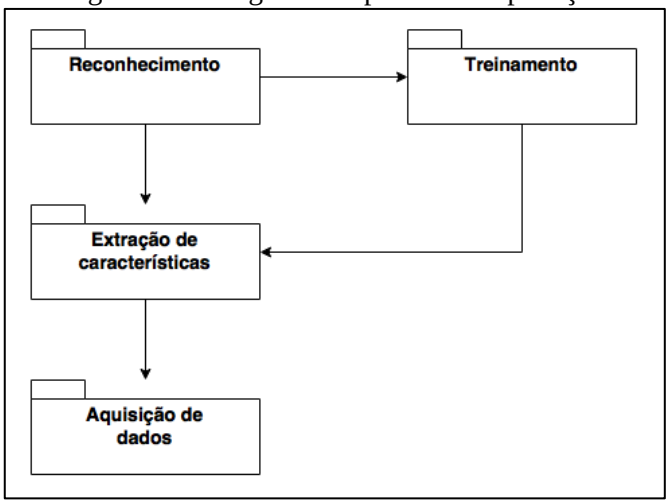
Quadro 11 – UC03 Reproduzir um sinal

Número	03
Caso de Uso	Reproduzir um sinal
Ator	Usuário
Pré-condições	1. Estar com o Leap Motion conectado; 2. Os algoritmos de reconhecimento de sinais estarem treinados.
Cenário Principal	1. O Usuário visualiza o sinal que deve ser reproduzido; 2. O Usuário reproduz o sinal; 3. O Usuário é notificado que o sinal reproduzido foi reconhecido.
Fluxo Alternativo	1. No passo 2, o sinal reproduzido pode não ser classificado corretamente, sendo que o fluxo continua no passo 2.

3.2.2 Diagrama de pacotes

Para facilitar o entendimento da aplicação optou-se pelo diagrama de pacotes da Figura 21. Esse diagrama agrupou as classes em pacotes que definem os principais passos envolvidos no reconhecimento de um sinal.

Figura 21 – Diagrama de pacotes da aplicação

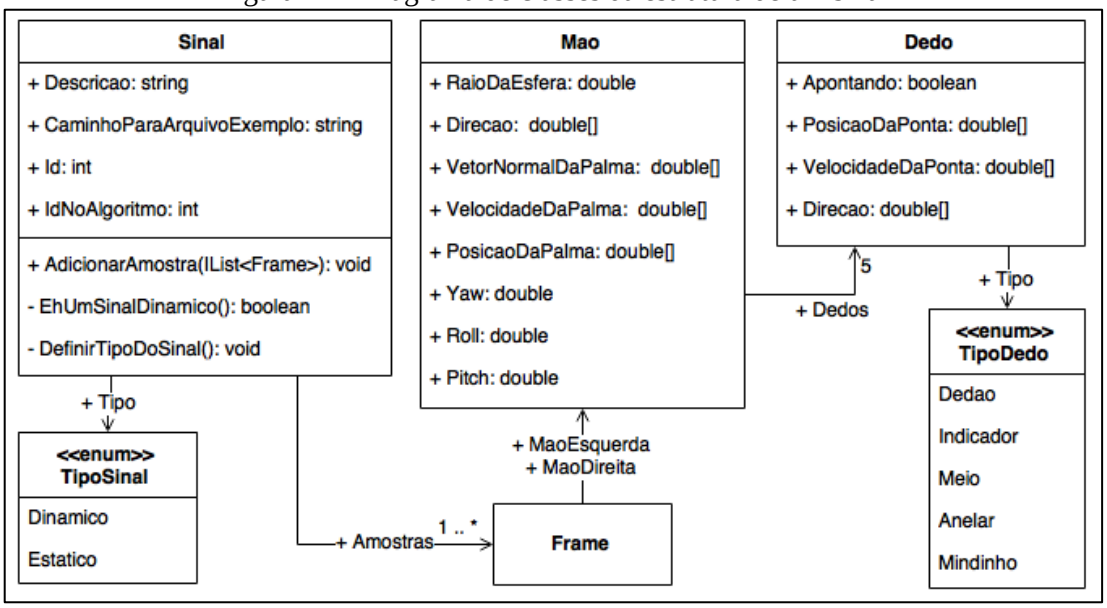


Nas próximas seções são detalhadas as principais características dos pacotes da Figura 21. A seção 3.2.2.1 apresenta os detalhes do pacote `Aquisição de dados` que contém a estrutura utilizada para representar um sinal e a aquisição de dados do Leap Motion. A seção 3.2.2.2 descreve o pacote `Extração de características`, onde são extraídas as características utilizadas para treinar os algoritmos e classificar os sinais. A seção 3.2.2.3 descreve o pacote `Treinamento`, que contém as estruturas para preparar as características para treinar os algoritmos. Por fim, a seção 3.2.2.4 apresenta o pacote `Reconhecimento`, onde estão os algoritmos de classificação e as estruturas envolvidas no reconhecimento de sinais. Os demais diagramas de classes podem ser vistos no apêndice A.

3.2.2.1 Pacote `Aquisição de dados`

O pacote de aquisição de dados define a representação de um sinal na aplicação, assim como as informações do Leap Motion são adaptadas para essa representação. A Figura 22 apresenta as classes que definem a estrutura de um sinal e a Figura 23 demonstra as classes que transformam as informações do Leap Motion para essa estrutura.

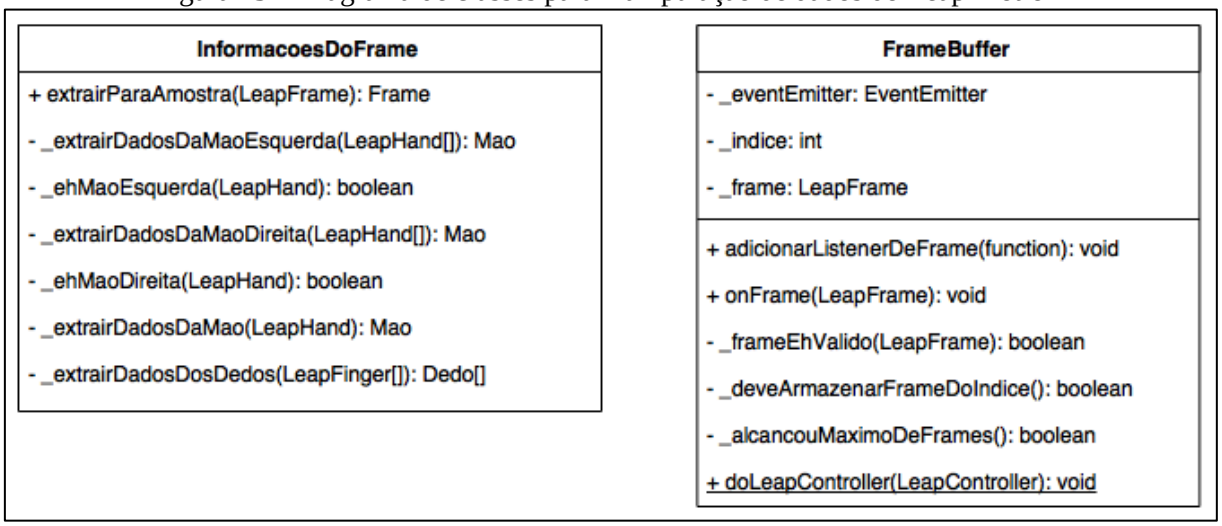
Figura 22 – Diagrama de classes da estrutura de um sinal



A classe `Sinal` (Figura 22) é responsável por representar um sinal dentro da aplicação. Essa classe é usada para prover informações sobre um sinal, como por exemplo: a descrição (`Descricao`), o identificador (`Id`), o identificador no algoritmo de classificação (`IdNoAlgoritmo`) e também as amostras do sinal (propriedade `Amostras`). Uma amostra de um sinal é um vetor de *frames*, dessa forma, uma amostra de sinal estático contém apenas um *frame* e uma amostra de um sinal dinâmico pode conter muitos *frames*.

As classes `Frame`, `Mao` e `Dedo` (Figura 22) funcionam como estruturas de dados, elas armazenam informações que serão manipuladas posteriormente por outras classes. A classe `Frame` contém as informações necessárias de um *frame*, as mãos. A classe `Mao` contém as informações de cada mão em um *frame* e a classe `Dedo` contém as informações de cada dedo em uma mão.

Figura 23 – Diagrama de classes para manipulação de dados do Leap Motion



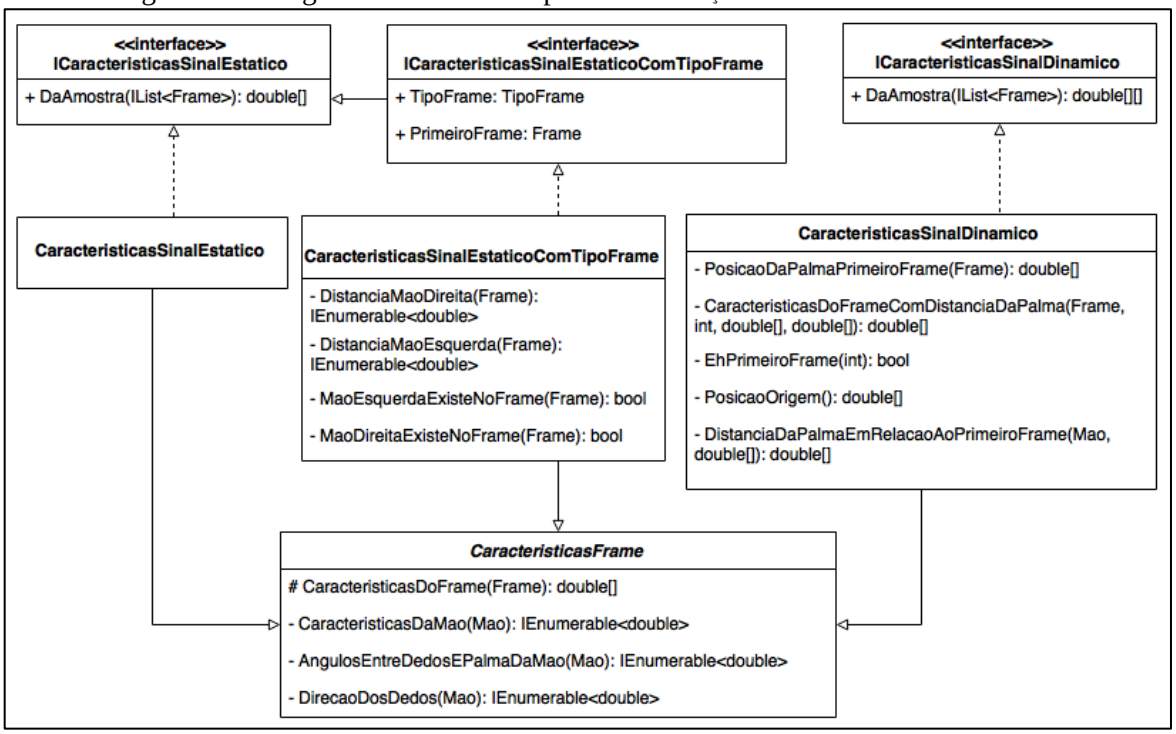
A classe `FrameBuffer` (Figura 23) é responsável por se conectar com a API do Leap Motion e capturar os *frames* que são enviados pelo dispositivo. Essa classe também funciona como um acumulador de *frames*, filtrando os *frames* que são enviados pelo dispositivo. O método `onFrame` recebe os *frames* do dispositivo e propaga para o resto da aplicação quando for necessário. Para verificar se é necessário enviar um *frame* para a aplicação são necessários os métodos `_frameEhValido`, `_deveArmazenarFrameDoIndice` e `_alcancouMaximoDeFrames`. A propagação dos *frames* para a aplicação ocorre através do padrão de projeto `Observer`. O padrão é implementado pela biblioteca EventEmitter e utilizado com o atributo `_eventEmitter`. O método `adicionarListenerDeFrame` permite que um observador se registre para receber as notificações de um novo *frame*.

A classe `InformacoesDoFrame` (Figura 23) é responsável por adaptar um *frame* do Leap Motion em um *frame* com a estrutura da classe `Frame`. Essa transformação é feita pelo método `extrairParaAmostra`. Os métodos `_extrairDadosDaMaoDireita` e `_extrairDadosDaMaoEsquerda` adaptam as informações do Leap Motion das respectivas mãos para as respectivas propriedades da classe `Mao`. Os métodos `_ehMaoDireita` e `_ehMaoEsquerda` verificam se uma mão é uma mão direita ou esquerda respectivamente. Finalmente, o método `_extrairDadosDosDedos` é responsável por adaptar as informações dos dedos de acordo com a classe `Dedo`.

3.2.2.2 Pacote `Extração de características`

O pacote de extração de características é responsável por extrair características das amostras dos sinais em um vetor para os algoritmos de classificação. As classes que fazem parte desse pacote podem ser vistas na Figura 24.

Figura 24 – Diagrama de classes do pacote `Extração de características`



A partir das interfaces `ICaracteristicasSinalEstatico` e `ICaracteristicasSinalDinamico` (Figura 24) é possível extrair as características de sinais. Os respectivos métodos `DaAmostra` conseguem extrair as características dos sinais estáticos e dinâmicos. As duas interfaces são necessárias devido aos diferentes tipos de retorno de cada tipo de sinal. O método `DaAmostra` da interface `ICaracteristicasSinalEstatico` retorna apenas um vetor que representa as características de um sinal estático. O método `DaAmostra` da interface `ICaracteristicasSinalDinamico` retorna uma matriz que representa as características de cada *frame* de um sinal dinâmico.

A interface `ICaracteristicasSinalEstaticoComTipoFrame` (Figura 24) é necessária para extrair características de cada *frame* de um sinal dinâmico. Esta interface permite extrair as características de um *frame* com o método `DaAmostra` e informações como: tipo do *frame* com a propriedade `TipoFrame` e a distância do *frame* em relação ao primeiro *frame* com a propriedade `PrimeiroFrame`.

A classe abstrata `CaracteristicasFrame` (Figura 24) é a classe que consegue extrair as características de um único *frame* através do método `CaracteristicasDoFrame`. As características de cada *frame* são extraídas com auxílio dos métodos `CaracteristicasDaMao`, `AngulosEntreDedosEPalmaDaMao` e `DirecaoDosDedos`. Como a classe `CarateristicasFrame` (Figura 24) é abstrata é necessário prover uma implementação que estende essa classe e possa ser instanciada. Nesse caso, as classes

`CaracteristicasSinalEstatico`, `CaracteristicasSinalEstaticoComTipoFrame` e `CaracteristicasSinalDinamico` fornecem as devidas implementações. Essas classes também implementam as devidas interfaces para prover as características de acordo com cada método `DaAmostra`.

A classe `CaracteristicasSinalEstatico` (Figura 24) é responsável por implementar a extração de características de sinais estáticos. Como um sinal estático pode ser visto como apenas um *frame*, esta classe apenas delega as operações para a classe base `CaracteristicasFrame`.

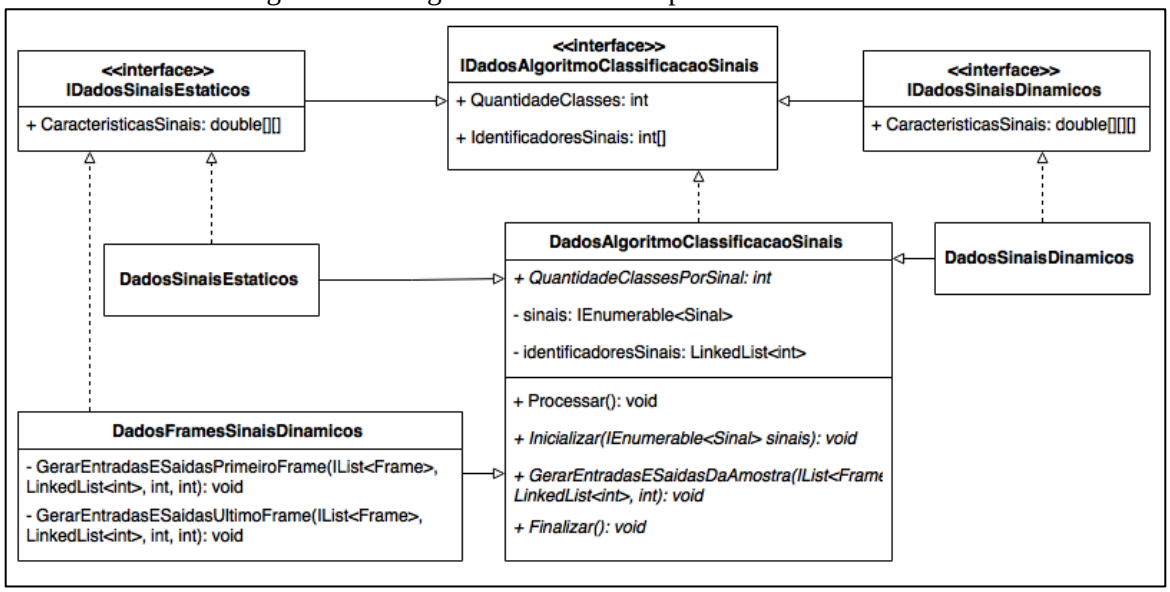
A classe `CaracteristicasSinalEstaticoComTipoFrame` (Figura 24) implementa a extração de características dos *frames* de sinais dinâmicos. Essa classe extrai as características padrões de um *frame* através da classe base `CaracteristicasFrame` e provê as características tipo do *frame* e distância em relação ao primeiro *frame*. Para isso, são necessários os métodos `DistanciaMaoDireita` e `DistanciaMaoEsquerda` que calculam a distância das respectivas mãos em relação ao primeiro *frame*. Também são necessários os métodos `MaoEsquerdaExisteNoFrame` e `MaoDireitaExisteNoFrame` que indicam se as respectivas mãos existem no *frame* para que seja possível calcular as distâncias.

Por fim, a classe `CaracteristicasSinalDinamico` (Figura 24) é responsável por implementar a extração de características de sinais dinâmicos. Essa classe extrai as características de cada *frame* que compõe um sinal dinâmico utilizando a implementação da classe base `CaracteristicasFrame`. Além das características padrões de cada *frame*, essa classe também adiciona a distância da palma da mão de cada *frame* em relação ao primeiro *frame*. Para adicionar essa característica são necessários os métodos `PosicaoDaPalmaPrimeiroFrame`, `CaracteristicasDoFrameComDistanciaDaPalma`, `EhPrimeiroFrame`, `PosicaoOrigem`, `DistanciaDaPalmaEmRelacaoAoPrimeiroFrame`.

3.2.2.3 Pacote `Treinamento`

O pacote de treinamento é responsável por reunir as características dos sinais e agrupá-las no formato que o *framework* Accord.NET espera. O diagrama de classes desse pacote pode ser visto na Figura 25.

Figura 25 – Diagrama de classes do pacote `Treinamento`



As interfaces `IDadosSinaisEstaticos` e `IDadosSinaisDinamicos` (Figura 25) fornecem as informações para treinar os algoritmos de classificação com as características extraídas dos sinais. Essas interfaces permitem acessar as características dos sinais, a quantidade de sinais diferentes e o identificador de cada sinal através das propriedades `CaracteristicasSinais`, `QuantidadeClasses` e `IdentificadoresSinais`, respectivamente.

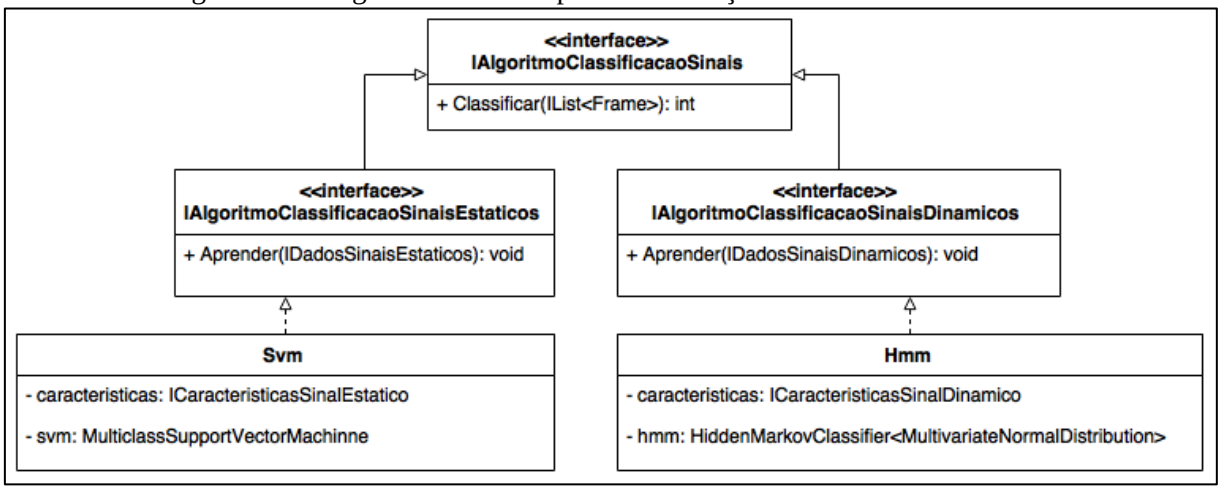
O padrão de projeto `Template Method` foi utilizado na estrutura de classes responsáveis por gerar os dados de treinamento. A classe abstrata `DadosAlgoritmoClassificacaoSinais` (Figura 25) é a classe fundamental da estrutura. Esta classe é responsável pelo processamento padrão para gerar os dados de treinamento, sendo que as operações específicas são delegadas para as subclasses que implementam os métodos abstratos `Inicializar`, `GerarEntradasESaidasDaAmostra` e `Finalizar`. As subclasses também devem informar quantos sinais são representados por uma amostra através da propriedade `QuantidadeClassesPorSinal`.

As classes `DadosSinaisEstaticos` e `DadosSinaisDinamicos` (Figura 25) apenas implementam os métodos necessários pelo `Template Method`. Elas não precisam adicionar mais nenhum comportamento para gerar os dados de sinais estáticos e dinâmicos. A classe `DadosFramesSinaisDinamicos` gera dados de treinamento para cada *frame* de um sinal dinâmico. Essa classe também implementa o `Template Method`, no entanto, ela requer os métodos `GerarEntradasESaidasPrimeiroFrame` e `GerarEntradasESaidasUltimoFrame` para gerar os dados que representam o primeiro *frame* e os dados que representam o último *frame* de um sinal dinâmico.

3.2.2.4 Pacote `Reconhecimento`

O pacote de reconhecimento é responsável pelos algoritmos de classificação e pela estrutura que envolve o reconhecimento de sinais. A Figura 26 apresenta o diagrama de classes da classificação de sinais no servidor.

Figura 26 – Diagrama de classes para classificação de um sinal no servidor



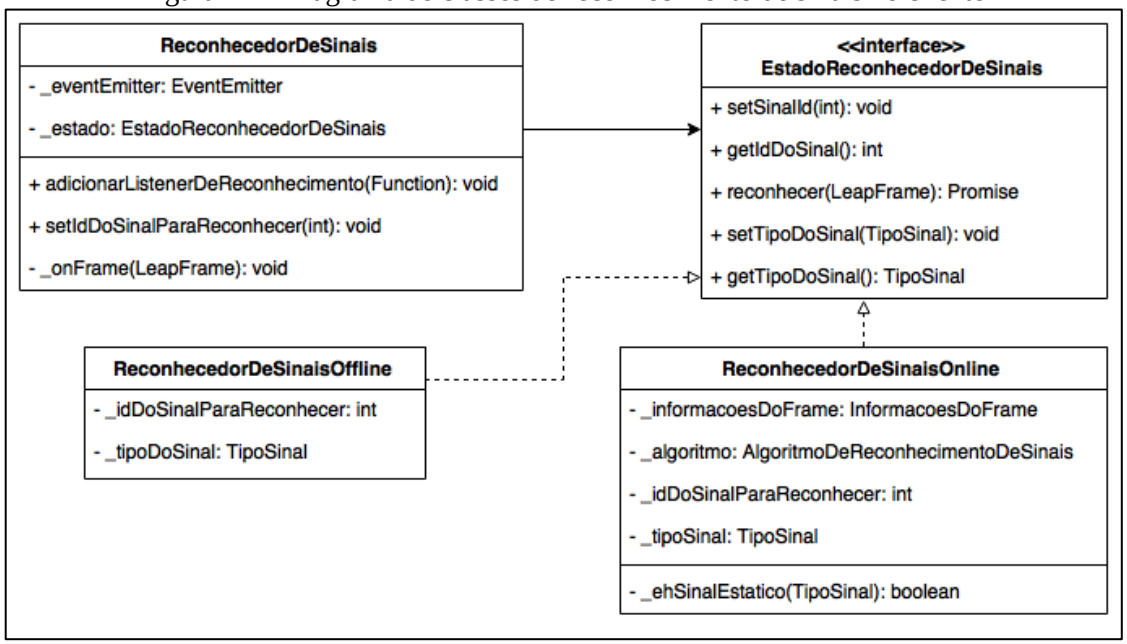
A classificação de um sinal é feita através da interface `IAlgoritmoClassificacaoSinais` (Figura 26) com o método `Classificar`. Este método requer uma amostra que será classificada e retorna o identificador do sinal fornecido pela interface `IDadosAlgoritmoClassificacaoSinais` do pacote `Treinamento`. Além da interface para classificação, os algoritmos também implementam as interfaces `IAlgoritmoClassificacaoSinaisEstaticos` e `IAlgoritmoClassificacaoSinaisDinamicos` que permitem que os algoritmos sejam treinados para classificar sinais através do método `Aprender`.

A classe `Svm` (Figura 26) é responsável pela classificação de sinais estáticos e também para classificação dos *frames* de um sinal dinâmico utilizando SVMs. Essa classe apenas implementa os métodos requisitados pela interface `IAlgoritmoClassificacaoSinaisEstaticos`.

A classe `Hmm` (Figura 26) é responsável pela classificação de sinais dinâmicos utilizando HMMs. Essa classe apenas implementa os métodos requisitados pela interface `IAlgoritmoClassificacaoSinaisDinamicos`.

Além da estrutura de classificação, o pacote `Reconhecimento` também contém as classes para efetuar o reconhecimento de sinais. A Figura 27 demonstra as classes para efetuar o reconhecimento de um sinal.

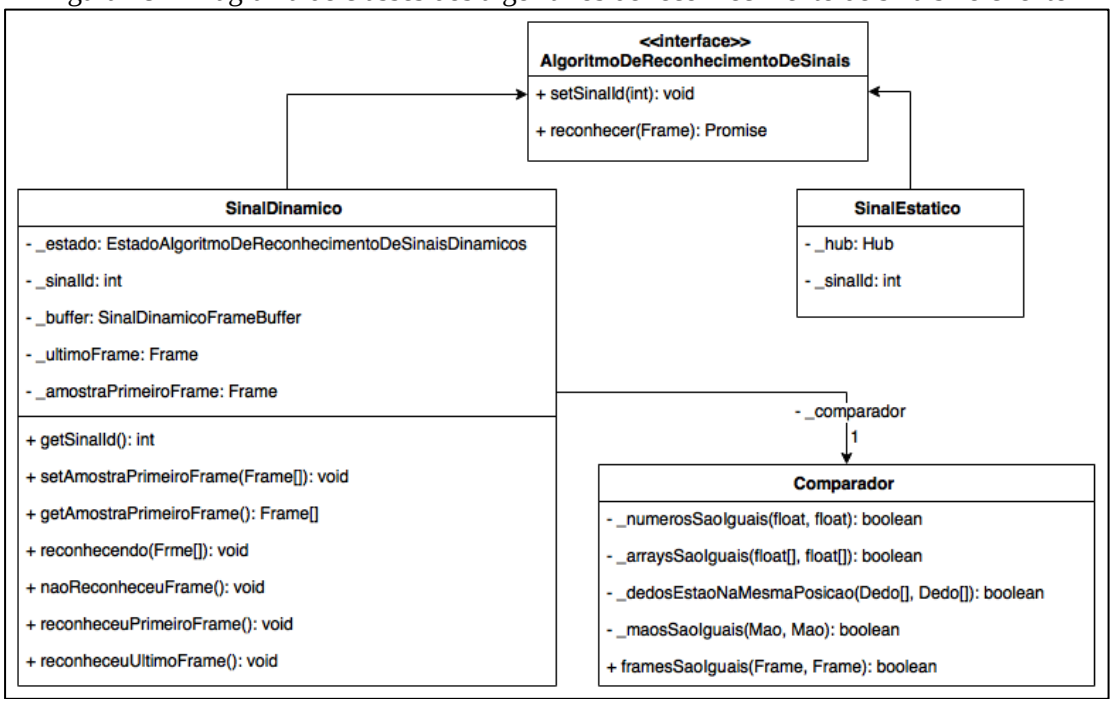
Figura 27 – Diagrama de classes do reconhecimento de sinais no cliente



O reconhecimento de um sinal no cliente ocorre a partir da classe `ReconhecedorDeSinais` (Figura 27). Essa classe recebe o identificador do sinal que deve ser reconhecido através do método `setIdDoSinalParaReconhecer` e quando o sinal é reconhecido ela notifica os observadores através do padrão de projetos `Observer`, implementado pela biblioteca EventEmitter com o atributo `_eventEmitter`. Essa classe não é responsável pelo reconhecimento de um sinal, ela delega o reconhecimento para os estados internos através do padrão de projetos `State` com as classes `ReconhecedorDeSinaisOffline` e `ReconhecedorDeSinaisOnline`.

O padrão `State` foi implementado porque o Leap Motion pode começar a enviar *frames* enquanto o cliente ainda não se comunicou com o servidor e nesse caso o estado do `ReconhecedorDeSinais` é `ReconhecedorDeSinaisOffline` (Figura 27). Este estado retorna os valores padrões para as operações da interface `EstadoReconhecedorDeSinais`. Quando a comunicação com o servidor for estabelecida, o estado do `ReconhecedorDeSinais` passa a ser `ReconhecedorDeSinaisOnline`. O `ReconhecedorDeSinaisOnline` identifica qual é o tipo do sinal através do atributo `_tipoDoSinal` e do método `_ehSinalEstatico` e delega o reconhecimento do sinal para o devido algoritmo do atributo `_algoritmo`. Essa classe ainda adapta as informações dos *frames* do Leap Motion para a classe `Frame` utilizando a classe `InformacoesDoFrame` pelo atributo `_informacoesDoFrame`. A Figura 28 apresenta o diagrama de classe dos algoritmos de reconhecimento de sinais que implementam a interface `AlgoritmoDeReconhecimentoDeSinais`.

Figura 28 – Diagrama de classes dos algoritmos de reconhecimento de sinais no cliente



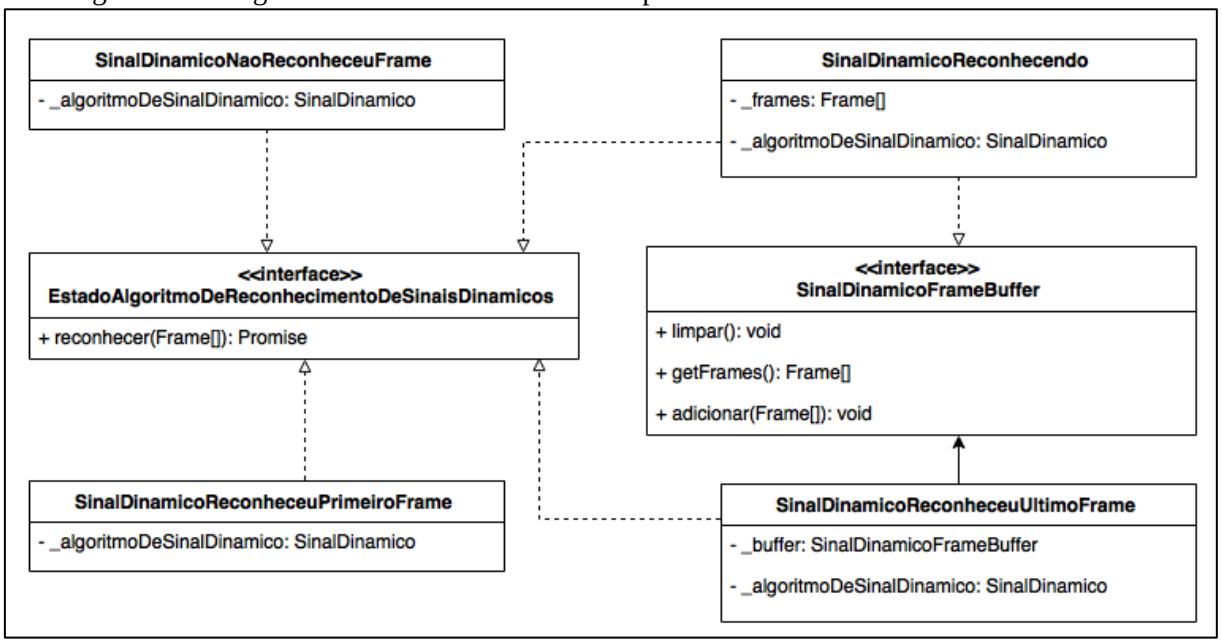
A classe `SinalEstatico` (Figura 28) é responsável pelo reconhecimento de sinais estáticos. Para isso, ela armazena o identificador do sinal que deve ser reconhecido no atributo `_sinalId` e faz o reconhecimento deste sinal através do método `reconhecer`. O atributo `_hub` é implementado pela biblioteca SignalR e permite a chamada de um método no servidor através de *WebSocket*.

A classe `SinalDinamico` (Figura 28) é responsável pelo reconhecimento de sinais dinâmicos e possui uma estrutura um pouco mais complexa. O reconhecimento de um sinal dinâmico envolve alguns estados e cuidados a mais com os *frames*. Os estados envolvidos no reconhecimento de um sinal dinâmico são novamente implementados através do padrão `State` pela interface `EstadoAlgoritmoReconhecimentoDeSinaisDinamicos`. Além desses estados, a classe `SinalDinamico` ainda guarda o último *frame* que foi utilizado por qualquer um dos estados no atributo `_ultimoFrame` para que ele seja comparado com o *frame* atual pela classe `Comparador`. A classe `Comparador` é então responsável por verificar se houve algum movimento entre os *frames*, permitindo que o reconhecimento seja efetuado.

A classe `SinalDinamico` também guarda a amostra do primeiro *frame* reconhecido no atributo `_amostraPrimeiroFrame` para que ele seja utilizado durante o reconhecimento do último *frame*. A classe tem um *buffer* que guarda os *frames* que são enviados pelo Leap Motion enquanto se está tentando reconhecer um *frame*, para que estes *frames* intermediários possam ser enviados para o reconhecimento do sinal completo. Finalmente, a classe expõe os métodos `reconhecendo`, `naoReconheceuFrame`, `reconheceuPrimeiroFrame` e

`reconheceuUltimoFrame` que manipulam o estado interno do reconhecimento de um sinal dinâmico. Cada um desses métodos muda o estado para a classe com o mesmo nome do diagrama da Figura 29.

Figura 29 – Diagrama de classes com os estados para reconhecimento de um sinal dinâmico



As classes `SinalDinamicoNaoReconheceuFrame`, `SinalDinamicoReconhecendo`, `SinalDinamicoReconheceuPrimeiroFrame` e `SinalDinamicoReconheceuUltimoFrame` (Figura 29) representam os quatro estados internos da classe `SinalDinamico`. A classe `SinalDinamicoNaoReconheceuFrame` é o estado inicial porque o primeiro *frame* do sinal dinâmico ainda não foi reconhecido, portanto esta classe se comunica com o servidor para tentar reconhecer o primeiro *frame* de um sinal dinâmico. A classe `SinalDinamicoReconheceuPrimeiroFrame` é o estado atribuído depois que o primeiro *frame* foi reconhecido, portanto essa classe tenta reconhecer o último *frame* de um sinal dinâmico dado o primeiro *frame* reconhecido. Por fim, a classe `SinalDinamicoReconheceuUltimoFrame` é o estado atribuído depois que o último *frame* foi reconhecido e esta classe se comunica com o servidor para tentar reconhecer o sinal dinâmico como um todo dado o primeiro *frame*, o último *frame* e os *frames* armazenados pelo *buffer* no atributo `_buffer`. A classe `SinalDinamicoReconhecendo` é um estado que não se comunica com o servidor, mas é um estado intermediário enquanto um *frame* está sendo reconhecido e outro *frame* chega para ser reconhecido. Portanto, os *frames* que chegam neste estado intermediário são armazenados no *buffer* pelo atributo `_frames` e serão utilizados posteriormente para reconhecer um sinal dinâmico como um todo.

3.3 IMPLEMENTAÇÃO

Nas próximas seções são apresentadas as técnicas e ferramentas utilizadas para o desenvolvimento da aplicação. Também são apresentadas as principais rotinas envolvendo o reconhecimento de sinais e a operacionalidade da implementação.

3.3.1 Técnicas e ferramentas utilizadas

A aplicação foi desenvolvida em uma arquitetura cliente-servidor. A camada cliente é responsável pela interface com o usuário e captura de dados através do dispositivo Leap Motion e a camada servidor é responsável pelo armazenamento de informações e classificação dos sinais.

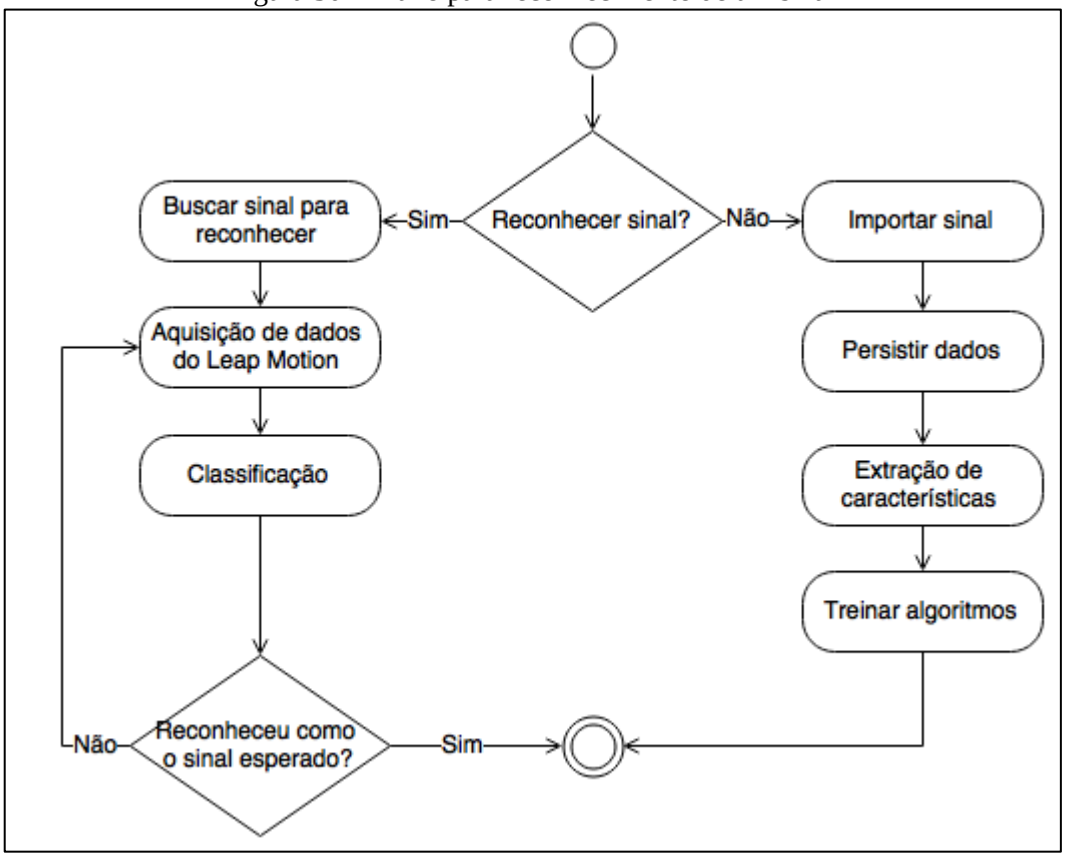
A camada cliente foi desenvolvida utilizando as linguagens Javascript, CSS e HTML com auxílio do editor Atom, do gerenciador de dependência Bower e do executor de tarefas Gulp. Para o desenvolvimento desta camada foram utilizadas as seguintes bibliotecas: EventEmitter, jQuery, LeapJS, Leap Plugins, Leap Rigged Hand, Leap Playback, Pure CSS, SignalR, Three.js e Three.js Controls.

A camada servidor foi desenvolvida utilizando a linguagem C# com auxílio da IDE Visual Studio 2013, da extensão ReSharper e da extensão NuGet para gerenciamento de dependências. Para o desenvolvimento dessa camada foram utilizadas as seguintes bibliotecas: Accord.NET, ASP.NET WebAPI, ASP.NET SignalR, FluentAssertions, JSON.NET, Microsoft Unity, Microsoft UnitTestFramework e Moq.

3.3.2 Fluxo geral da aplicação

Para um melhor entendimento da estrutura da aplicação foi elaborado o diagrama de atividades da Figura 30 representando os dois principais fluxos da aplicação.

Figura 30 – Fluxo para reconhecimento de um sinal



A partir da Figura 30 é possível perceber que a aplicação pode seguir dois fluxos, um para reconhecer sinais e outro para adicionar sinais. O reconhecimento de sinais estáticos e dinâmicos utiliza a mesma estrutura geral, mas diferem no passo Classificação. Os detalhes para esse passo podem ser vistos no apêndice B.

Nas próximas seções são apresentados os passos individuais do diagrama da Figura 30. A seção 3.3.3 descreve a importação de um sinal. A seção 3.3.4 demonstra a persistência dos dados. A seção 3.3.5 demonstra como é selecionado um sinal para ser reconhecido. Na seção 3.3.6 é demonstrado como são adquiridos os dados do Leap Motion. Na seção 3.3.7 é demonstrada a extração de características de um sinal e a seção 3.3.8 apresenta o treinamento dos algoritmos. Finalmente, na seção 3.3.9 é demonstrada a classificação e o reconhecimento de sinais.

3.3.3 Importar sinal

Como sinais estáticos e dinâmicos compartilham a mesma estrutura representada pela classe `Sinal`, o procedimento para adicionar novos sinais ou amostras é o mesmo. Este procedimento é definido pelo método `SalvarAmostraDoSinal` da classe `GerenciadorSinais`, como pode ser visto no Quadro 12.

Quadro 12 – Código do método `SalvarAmostraDoSinal`

```
01 public void SalvarAmostraDoSinal(string descricaoDoSinal, string
conteudoDoArquivoDeExemplo, IList<Frame> amostra)
02 {
03      var nomeDoArquivo =
CriarArquivoDeExemploSeNaoExistir(descricaoDoSinal,
conteudoDoArquivoDeExemplo);
04      Adicionar(new Sinal
05      {
06          Descricao = descricaoDoSinal,
07          CaminhoParaArquivoDeExemplo = nomeDoArquivo,
08          Amostras = new List<IList<Frame>> { amostra }
09      });
10 }
```

O método `SalvarAmostraDoSinal` (Quadro 12) cria o arquivo de exemplo do sinal que será exibido para o usuário através do método `CriarArquivoDeExemploSeNaoExistir` (linha 03) e adiciona o sinal à coleção de sinais (linha 04). O método `Adicionar` (Quadro 13) tem a responsabilidade de decidir se a amostra que está sendo adicionada é um novo sinal ou uma nova amostra de um sinal já existente.

Quadro 13 – Código do método `Adicionar`

```
01 private void Adicionar(Sinal sinal)
02 {
03      var sinalNoRepositorio =
repositorio.BuscarPorDescricao(sinal.Descricao);
04      if (sinalNoRepositorio == null)
05          repositorio.Adicionar(sinal);
06      else
07          sinalNoRepositorio.AdicionarAmostra(sinal.Amostras[0]);
08
09      repositorio.SalvarAlteracoes();
10 }
```

O método `Adicionar` (Quadro 13) procura no repositório um sinal com a mesma descrição do sinal que está sendo adicionado (linha 03). Se este sinal não existir, ele é adicionado ao repositório (linha 05), caso contrário a amostra do sinal é adicionada às amostras do sinal que já existe (linha 07). Por fim, o método persiste as alterações feitas no repositório através do método `SalvarAlteracoes` (linha 09).

3.3.4 Persistir dados

Para persistir os dados dos sinais, optou-se por utilizar um arquivo JSON. Os dados desse arquivo são a representação no formato JSON da classe `Sinal` da Figura 22. A biblioteca JSON.NET recebe um objeto e retorna a representação do mesmo em uma `string` com o conteúdo JSON como pode ser visto no Quadro 54 do apêndice C.

3.3.5 Buscar sinal para reconhecer

A sequência de sinais para reconhecimento é a sequência de cadastro, para isso a camada cliente deve se conectar com à camada servidor para saber qual é o próximo sinal. O Quadro 14 demonstra como a camada cliente pede para o servidor qual é o próximo sinal para ser reconhecido.

Quadro 14 – Código para carregar o próximo sinal

```
01 _carregarProximoSinal: function() {
02     Signa.Hubs
03         .iniciar()
04         .done(function() {
05             var id = this._informacoesDoSinal ?
this._informacoesDoSinal.Id : -1;
06             Signa.Hubs
07                 .sinais()
08                 .proximoSinal(id)
09                 .done(this._onNovoSinal.bind(this));
10         }.bind(this));
11 }
```

O Quadro 14 demonstra o código necessário para buscar o próximo sinal. Inicialmente, na linha 03 é iniciada a comunicação com o servidor e quando ela estiver estabelecida é executada a função definida na linha 04. Antes de pedir o próximo sinal para o servidor, é preciso obter o identificador do sinal anterior e, se ele não existir deve-se assumir o valor `-1` (linha 05). Finalmente, é possível pedir para a classe `Sinais` no servidor o próximo sinal através do método `ProximoSinal` (linhas 07 e 08) e quando o sinal for selecionado o método `_onNovoSinal` (linha 09) será executado. O Quadro 15 demonstra o método `ProximoSinal` da classe `Sinais`.

Quadro 15 – Método `ProximoSinal` da classe `Sinais`

```
01 public ProximoSinalResponseModel ProximoSinal(int
indiceDoSinalAnterior)
02 {
03     var indice = indiceDoSinalAnterior + 1;
04     if (indice == repositorio.Quantidade)
05         indice = 0;
06
07     var sinal = repositorio.BuscarPorIndice(indice);
08     return new ProximoSinalResponseModel
09     {
10         Id = sinal.Id,
11         IdReconhecimento = sinal.IdNoAlgoritmo,
12         Descricao = sinal.Descricao,
13         CaminhoParaArquivoDeExemplo =
sinal.CaminhoParaArquivoDeExemplo,
14         Tipo = sinal.Tipo
15     };
16 }
```

Inicialmente, o método `ProximoSinal` (Quadro 15) verifica se o índice do próximo sinal não é maior que a quantidade de sinais existente (linha 04) e reinicia em zero se

necessário (linha 05). Com o índice correto, é possível buscar o sinal pelo índice no repositório (linha 07). Com o sinal obtido pela consulta no repositório, é possível retornar as informações para a camada cliente (linha 08).

3.3.6 Aquisição de dados do Leap Motion

A API do Leap Motion disponibiliza os dados no formato de *frames* através do padrão de projetos `Observer`, onde é necessário registrar um observador que será notificado a cada novo *frame*. O Quadro 16 apresenta como esse observador é registrado pela classe `FrameBuffer` que é responsável por propagar os *frames* da API do Leap Motion para a aplicação.

Quadro 16 – Código para registrar um observador na API do Leap Motion

```
01 FrameBuffer.doLeapController = function(leapController) {
02     var frameBuffer = new FrameBuffer();
03     leapController.on('frame', frameBuffer.onFrame.bind(frameBuffer));
04     return frameBuffer;
05 };
06 // CÓDIGO...
07 adicionarListenerDeFrame: function(callback) {
08     this._eventEmitter.addListener(ID_EVENTO_FRAME, callback);
09 }
```

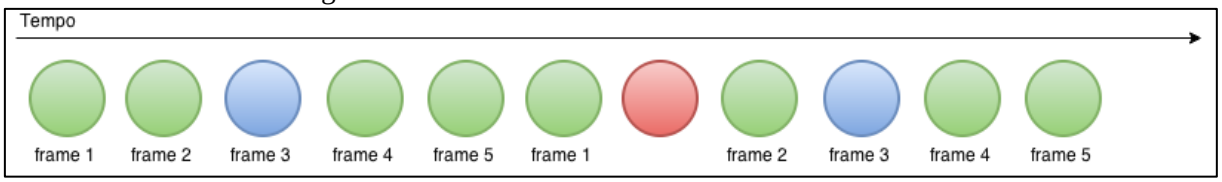
A partir do Quadro 16, na linha 03 é possível verificar como um observador é registrado na API do Leap Motion. O observador é o método `onFrame` da classe `FrameBuffer` e o evento disparado pela API do Leap Motion se chama *frame*. Também na linha 08 é possível verificar como um observador é registrado na classe `FrameBuffer` para ser notificado de um novo *frame*. O método `onFrame` é responsável por filtrar os *frames* propagá-los para a aplicação como demonstra o Quadro 17.

Quadro 17 – Código do método `onFrame`

```
01 var ID_EVENTO_FRAME = 'frame';
02 var INDICE_DO_FRAME_QUE_DEVE_SER_ARMAZENADO = 3;
03 var QUANTIDADE_DE_FRAMES_DO_BUFFER = 5;
04
05 onFrame: function(frame) {
06     if (this._frameEhValido(frame)) {
07         this._indice++;
08         if (this._deveArmazenarFrameDoIndice()) {
09             this._frame = frame;
10         } else if (this._alcancouMaximoDeFrames()) {
11             this._eventEmitter.trigger(ID_EVENTO_FRAME, [this._frame]);
12             this._indice = 0;
13         }
14     }
15 },
16 _frameEhValido: function(frame) {
17     return frame.hands.length > 0;
18 },
19 _deveArmazenarFrameDoIndice: function() {
20     return this._indice === INDICE_DO_FRAME_QUE_DEVE_SER_ARMAZENADO;
21 },
22 _alcancouMaximoDeFrames: function() {
23     return this._indice === QUANTIDADE_DE_FRAMES_DO_BUFFER;
24 }
```

O método `onFrame` na linha 05 do Quadro 17 funciona como um filtro para a aplicação. De todos os *frames* enviados pelo Leap Motion ele propaga para aplicação apenas um *frame* a cada cinco *frames* válidos como pode ser visto na Figura 31. Essa implementação foi escolhida devido a precisão do Leap Motion onde muitos *frames* seriam processados sem necessidade. Um *frame* válido é um *frame* que contém ao menos uma mão como pode ser visto no método `_frameEhValido` na linha 16. Caso o *frame* seja válido, então o método `onFrame` verifica se ele deve ser armazenado (linha 08) ou então se o *buffer* está cheio e o *frame* armazenado deve ser enviado para a aplicação (linha 10). Na linha 11 é possível perceber como um evento é disparado através da biblioteca EventEmitter para que os observadores recebam a notificação do *frame*.

Figura 31 – Funcionamento da classe `FrameBuffer`



A partir da Figura 31 é possível perceber o funcionamento da classe `FrameBuffer` ao longo do tempo. Os círculos verdes representam os *frames* válidos (que possuem ao menos uma mão) e os círculos vermelhos representam os *frames* inválidos (descartados). Os círculos azuis são os *frames* que realmente chegam na aplicação e que serão tratados para seja feito o reconhecimento de um sinal com eles.

Internamente a aplicação trabalha com a estrutura de *frames* definida na Figura 22. Para que seja possível trabalhar com essa estrutura é necessário adaptar os dados da API do Leap Motion para as devidas classes. Como pode ser visto no Quadro 18, o método `extrairParaAmostra` da classe `InformacoesDoFrame` faz essa adaptação.

Quadro 18 – Método `extrairParaAmostra`

```
01 extrairParaAmostra: function(frame) {
02     return {
03         MaoEsquerda: this._extrairDadosDaMaoEsquerda(frame.hands),
04         MaoDireita: this._extrairDadosDaMaoDireita(frame.hands)
05     };
06 },
07 _extrairDadosDaMaoEsquerda: function(maos) {
08     if (this._ehMaoEsquerda(maos[0]))
09         return this._extrairDadosDaMao(maos[0]);
10
11     if (this._ehMaoEsquerda(maos[1]))
12         return this._extrairDadosDaMao(maos[1]);
13
14     return null;
15 },
16 _ehMaoEsquerda: function(hand) {
17     return hand && hand.type.toUpperCase() === 'LEFT';
18 },
19 // o método _ehMaoDireita é semelhante ao método _ehMaoEsquerda
20 // o método _extrairDadosDaMaoDireita é semelhante ao método
_extrairDadosDaMaoEsquerda
```

O método `extrairParaAmostra` (Quadro 18) retorna um objeto com as propriedades `MaoEsquerda` e `MaoDireita` que representam as respectivas mãos. Os dados da mão esquerda são adaptados pelo método `_extrairDadosDaMaoEsquerda` que verifica se uma das duas mãos do *frame* é a mão esquerda (linhas 08 e 11) e adapta os dados da devida mão (linhas 09 e 12). Caso nenhuma delas seja a mão esquerda, então o valor `null` é retornado (linha 14). Para verificar se uma mão é a mão esquerda, basta verificar o tipo da mão pelo atributo `type` como demonstrado na linha 17. Os dados da mão direita são adaptados pelo método `_extrairDadosDaMaoDireita` que tem o funcionamento semelhante ao método `_extrairDadosDaMaoEsquerda`. A transformação dos dados é igual para ambas as mãos e pode ser vista no Quadro 19.

Quadro 19 – Métodos `_extrairDadosDaMao` e `_extrairDadosDosDedos`

```
01 _extrairDadosDaMao: function(leapHand) {
02     return {
03         VetorNormalDaPalma: leapHand.palmNormal,
04         PosicaoDaPalma: leapHand.stabilizedPalmPosition,
05         VelocidadeDaPalma: leapHand.palmVelocity,
06         Direcao: leapHand.direction,
07         Dedos: this._extrairDadosDosDedos(leapHand.fingers),
08         RaioDaEsfera: leapHand.sphereRadius,
09         Pitch: leapHand.pitch(),
10         Roll: leapHand.roll(),
11         Yaw: leapHand.yaw()
12     };
13 },
14 _extrairDadosDosDedos: function(leapFingers) {
15     var dedos = new Array(leapFingers.length);
16
17     for (var i = 0; i < dedos.length; i++) {
18         dedos[i] = {
19             Tipo: leapFingers[i].type,
20             Direcao: leapFingers[i].direction,
21             PosicaoDaPonta: leapFingers[i].stabilizedTipPosition,
22             VelocidadeDaPonta: leapFingers[i].tipVelocity,
23             Apontando: leapFingers[i].extended
24         };
25     }
26
27     return dedos;
28 }
```

O método `_extrairDadosDaMao` (Quadro 19) adapta as informações de uma mão da API do Leap Motion para a classe `Mao` (Figura 22). Em geral, esse método faz a associação dos atributos da API do Leap Motion para a classe `Mao`. Apenas o atributo `PosicaoDaPalma` (linha 04) tem um valor diferenciado que é a posição estabilizada da palma. A posição estabilizada é a posição com correções de velocidade que por sua vez possui mais precisão quanto à posição real da mão no espaço 3D. Cada mão possui um conjunto de dedos e esses são adaptados pelo método `_extrairDadosDosDedos` (linha 14) que retorna um vetor com os dedos das mãos. Este método segue a mesma ideia do método `_extrairDadosDaMao` fazendo uma associação dos atributos da API do Leap Motion com a classe `Dedo`.

3.3.7 Extração de características

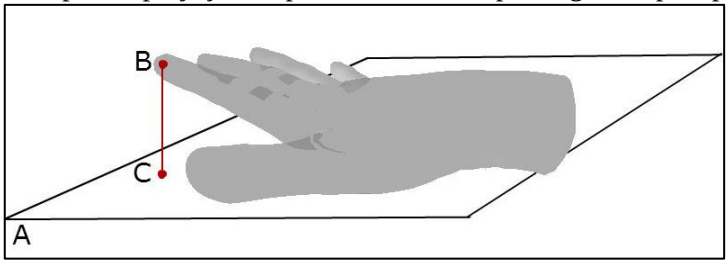
A extração de características consiste em transformar os dados da estrutura da classe `Frame` e um vetor que possa ser utilizado para classificar um sinal e também para treinar os algoritmos de classificação. As características padrões de um *frame* são extraídas pelo método `CaracteristicasDoFrame` da classe `CaracteristicasFrame` demonstrado no Quadro 20.

Quadro 20 – Implementação do método `CaracteristicasDoFrame`

```
01 protected double[] CaracteristicasDoFrame(Frame frame)
02 {
03     return CaracteristicasDaMao(frame.MaoEsquerda)
04         .Concat(CaracteristicasDaMao(frame.MaoDireita))
05         .ToArray();
06 }
07 private IEnumerable<double> CaracteristicasDaMao(Mao mao)
08 {
09     return mao.VetorNormalDaPalma.Normalizado()
10          .Concat(mao.Direcao.Normalizado())
11          .Concat(AngulosEntreDedos(mao))
12          .Concat(DirecaoDosDedos(mao));
13 }
14 private IEnumerable<double> DirecaoDosDedos(Mao mao)
15 {
16     return mao.Dedos
17         .Select(d => d.Direcao.Normalizado())
18         .ToArray()
19         .Concatenar();
20 }
```

As características de um *frame* são representadas por um vetor com as características de cada mão como demonstrado nas linhas 03, 04 e 05 do Quadro 20. O método `Concat` concatena os vetores e por fim o método `ToArray` converte a sequência em um vetor. As características de uma mão são compostas pelo vetor normal da palma da mão (linha 09), pela direção da mão (linha 10), o ângulo entre os dedos (linha 11) e também a direção dos dedos (linha 12). A direção dos dedos é obtida iterando sobre os dedos da mão e aplicando uma projeção transformando a coleção de dedos em uma coleção das direções normalizadas de cada dedo (linhas 16 e 17). Por fim, o método `Concatenar` une todas as coleções em um único vetor. Ainda é possível perceber que todos os valores são normalizados através da normalização de vetores implementada pelo método `Normalizado`. O ângulo entre os dedos é calculado a partir do plano gerado pelo vetor normal da palma da mão e da posição da ponta dos dedos (Figura 32).

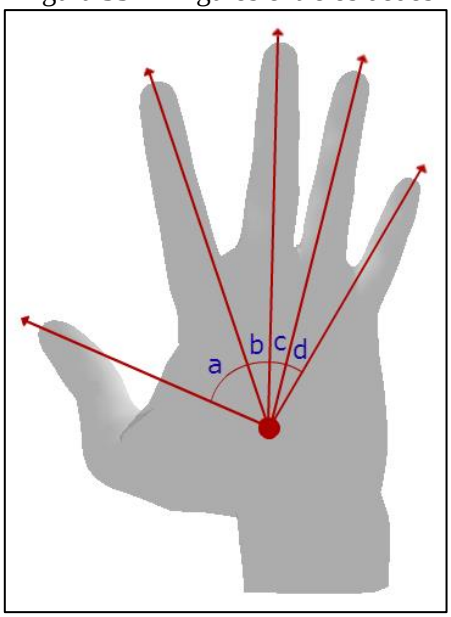
Figura 32 – Exemplo da projeção da ponta do dedo no plano gerado pela palma da mão



A Figura 32 demonstra um exemplo da projeção da ponta de um dedo em um plano. Supondo que o plano `A` tenha sido gerado pelo vetor normal da palma da mão, e o ponto `B` seja a posição da ponta do dedo no espaço, então o ponto `C` é a projeção do ponto `B` no plano `A`. O

ponto `c` será utilizado para fazer o cálculo do ângulo entre os dedos. A Figura 33 demonstra um exemplo de como o cálculo é feito e o Quadro 21 apresenta o código implementado.

Figura 33 – Ângulos entre os dedos



A Figura 33 apresenta os quatro ângulos utilizados como características. Os ângulos são calculados entre os dedos, partindo do dedão para o mindinho. O cálculo dos ângulos é feito utilizando os vetores formados entre a posição da ponta dos dedos e a posição da palma da mão.

Quadro 21 – Código do método `AngulosEntreDedos`

```
01 private IEnumerable<double> AngulosEntreDedos(Mao mao)
02 {
03     var angulos = new double[mao.Dedos.Length - 1];
04
05     for (int i = 0; i < angulos.Length; i++)
06     {
07         var dedoAtual = mao.Dedos[i];
08         var proximoDedo = mao.Dedos[i + 1];
09         var posicaoDedoAtual =
PosicaoDoDedoProjetadaNoPlanoDaPalma(dedoAtual, mao);
10         var posicaoProximoDedo =
PosicaoDoDedoProjetadaNoPlanoDaPalma(proximoDedo, mao);
11         angulos[i] = posicaoDedoAtual.AnguloAte(posicaoProximoDedo);
12     }
13
14     return angulos.Normalizado();
15 }
16 private double[] PosicaoDoDedoProjetadaNoPlanoDaPalma(Dedo dedo, Mao
mao)
17 {
18     return dedo.PosicaoDaPonta
19         .ProjetadoNoPlano(mao.VetorNormalDaPalma)
20
.Subtrair(mao.PosicaoDaPalma.ProjetadoNoPlano(mao.VetorNormalDaPalma));
21 }
```

O método `AngulosEntreDedos` (Quadro 21) é responsável por fazer o cálculo do ângulo entre a ponta dos dedos projetadas no plano gerado pelo vetor normal da palma da mão. O método itera sobre todos os dedos da mão e faz o cálculo com base no dedo atual da iteração e com o próximo dedo (linhas 07 e 08). Nas linhas 09 e 10 é possível perceber o cálculo da posição dos dedos em relação ao plano gerado e na linha 11 é feito o cálculo do ângulo utilizando esses valores. Por fim, a linha 14 retorna o valor dos ângulos normalizados. O método `PosicaoDoDedoProjetadaNoPlanoDaPalma` (linha 16) retorna o vetor da posição da ponta do dedo em relação a palma da mão, sendo que o vetor é projetado no plano gerado pela palma da mão.

A extração de características de um sinal estático é feita pela classe `CaracteristicasSinalEstatico` através do método `DaAmostra`. Este método recebe uma amostra e devolve o vetor de características dessa amostra como demonstrado no Quadro 22.

Quadro 22 – Código do método `DaAmostra` da classe `CaracteristicasSinalEstatico`

```
01 public double[] DaAmostra(IList<Frame> amostra)
02 {
03       var primeiroFrame = amostra[0];
04       return CaracteristicasDoFrame(primeiroFrame);
05 }
```

Como um sinal estático é composto por apenas um *frame*, o método `DaAmostra` retorna as características do primeiro *frame* de uma amostra como pode ser visto nas linhas 03 e 04 do Quadro 22.

As características de um sinal dinâmico são extraídas pela classe `CaracteristicasSinalDinamico`. O Quadro 23 demonstra o código do método `DaAmostra` dessa classe.

Quadro 23 – Método `DaAmostra` da classe `CaracteristicasSinalDinamico`

```
01 public double[][] DaAmostra(IList<Frame> amostra)
02 {
03     var caracteristicasDosFrames = new double[amostra.Count][];
04     var posicaoMaoDireitaPrimeiroFrame =
PosicaoDaPalmaPrimeiroFrame(amostra[0].MaoDireita);
05     var posicaoMaoEsquerdaPrimeiroFrame =
PosicaoDaPalmaPrimeiroFrame(amostra[0].MaoEsquerda);
06
07     for (int i = 0; i < amostra.Count; i++)
08         caracteristicasDosFrames[i] =
CaracteristicasDoFrameComDistanciaDaPalma(amostra[i], i,
posicaoMaoDireitaPrimeiroFrame, posicaoMaoEsquerdaPrimeiroFrame);
09
10     return caracteristicasDosFrames;
11 }
12
13 private double[] PosicaoDaPalmaPrimeiroFrame(Mao mao)
14 {
15     if (mao == null)
16         return PosicaoOrigem();
17
18     return mao.PosicaoDaPalma;
19 }
```

Como pode ser observado no Quadro 23, as características de um sinal dinâmico são um agregado das características de todos os *frames* que fazem parte do sinal. Inicialmente o método `DaAmostra` descobre a posição das mãos esquerda e direita no primeiro *frame* da amostra (linhas 04 e 05) utilizando o método `PosicaoDaPalmaPrimeiroFrame`. O método `PosicaoDaPalmaPrimeiroFrame` retorna o vetor `[0,0,0]` através do método `PosicaoOrigem` se a mão não existir, senão retorna a posição da palma da mão no *frame* (linhas 16 e 18).

A partir do Quadro 23, ainda é possível observar que o método `CaracteristicasDoFrameComDistanciaDaPalma` é responsável por extrair as características de cada *frame* da amostra de um sinal dinâmico. O Quadro 24 apresenta o código deste método.

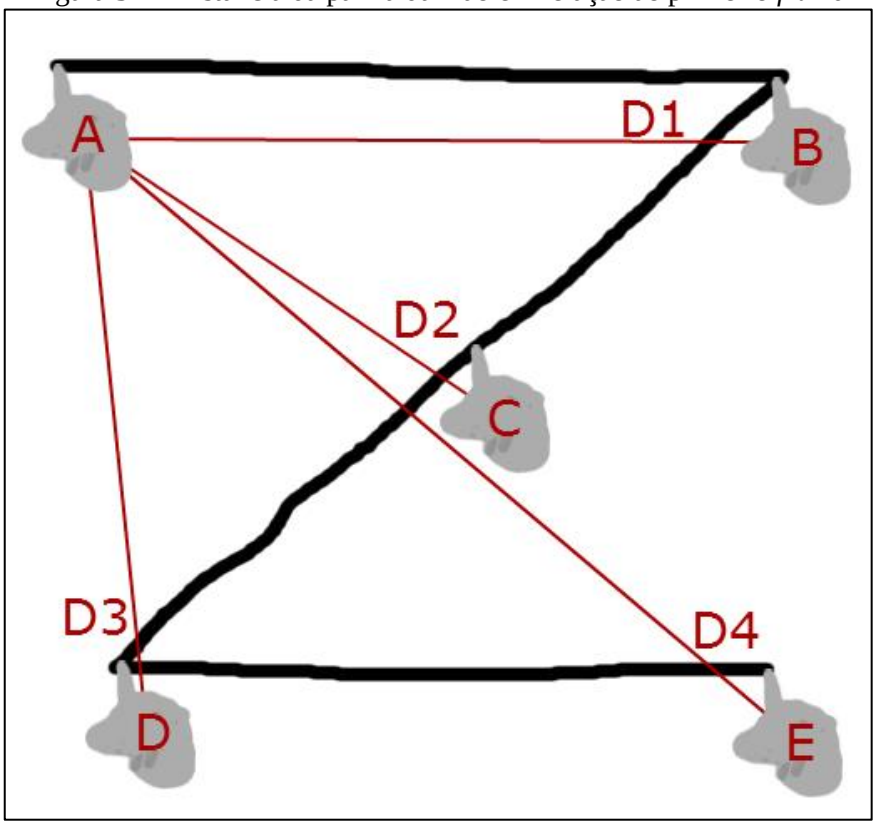
Quadro 24 – Código para calcular a distância da mão entre os frames

```
01 private double[] CaracteristicasDoFrameComDistanciaDaPalma(Frame frame,
int indice, double[] posicaoDaMaoDireitaPrimeiroFrame, double[]
posicaoDaMaoEsquerdaPrimeiroFrame)
02 {
03     var caracteristicas = CaracteristicasDoFrame(frame);
04     if (EhPrimeiroFrame(indice))
05     {
06         caracteristicas = caracteristicas
07             .Concat(PosicaoOrigem())
08             .Concat(PosicaoOrigem())
09             .ToArray();
10     }
11     else
12     {
13         caracteristicas = caracteristicas
14
.Concat(DistanciaDaMaoEmRelacaoAoPrimeiroFrame(frame.MaoDireita,
posicaoDaMaoDireitaPrimeiroFrame))
15
.Concat(DistanciaDaMaoEmRelacaoAoPrimeiroFrame(frame.MaoEsquerda,
posicaoDaMaoEsquerdaPrimeiroFrame))
16             .ToArray();
17     }
18     return caracteristicas;
19 }
20
21 private double[] DistanciaDaMaoEmRelacaoAoPrimeiroFrame(Mao mao,
double[] posicaoDaPalmaDaMaoNoPrimeiroFrame)
22 {
23     if (mao == null)
24         return PosicaoOrigem();
25
26     return mao.PosicaoDaPalma
27         .Subtrair(posicaoDaPalmaDaMaoNoPrimeiroFrame)
28         .Normalizado();
29 }
```

Como pode ser observado no Quadro 24, as características de um sinal dinâmico não são compostas apenas pelas características dos *frames*. Além das características comuns de um *frame*, também é adicionado um vetor que informa a distância da palma da mão em relação ao primeiro *frame* (linhas 14 e 15). Na linha 03 do Quadro 24, são extraídas as características padrões de um *frame* e, caso este seja o primeiro *frame* da amostra, então a distância das mãos é um vetor `[0, 0, 0]` (linha 05), senão é feito o cálculo da distância das mãos (linha 13).

O método `DistanciaDaMaoEmRelacaoAoPrimeiroFrame`, na linha 21 do Quadro 24, faz o cálculo da distância da palma da mão em relação ao primeiro *frame*. Se a mão não existir, então a distância é um vetor `[0,0,0]` (linha 24). Se a mão existir no *frame*, então é feita a diferença entre as posições (linha 27) e retornado o valor normalizado (linha 28). A Figura 34 demonstra a ideia do método.

Figura 34 – Distância da palma da mão em relação ao primeiro *frame*



A Figura 34 demonstra o sinal da letra Z feito em cinco *frames* onde é demonstrado como a distância entre um determinado *frame* é calculada em relação ao primeiro *frame*. Por exemplo, a linha `D1` que liga os *frames* `A` e `B` representa a distância da palma da mão entre esses *frames*. Também é possível perceber que o *frame* `A` não possui a distância calculada por ser a posição de origem.

Além das características de um sinal dinâmico como um todo, também é necessário extrair as características dos *frames* que fazem parte de um sinal dinâmico. A extração dessas características é responsabilidade da classe `CaracteristicasSinalEstaticoComTipoFrame`. O Quadro 25 apresenta o código dessa classe.

Quadro 25 – Código da classe `CaracteristicasSinalEstaticoComTipoFrame`

```
01 public double[] DaAmostra(IList<Frame> amostra)
02 {
03     double[] tipo = {(double)TipoFrame};
04 
05     return tipo
06         .Concat(DistanciaMaoDireita(amostra[0]))
07         .Concat(DistanciaMaoEsquerda(amostra[0]))
08         .Concat(caracteristicas.DaAmostra(amostra))
09         .ToArray();
10 }
11 private IEnumerable<double> DistanciaMaoDireita(Frame frame)
12 {
13     if (MaoDireitaExisteNoFrame(frame) &&
MaoDireitaExisteNoFrame(PrimeiroFrame))
14     {
15         return PrimeiroFrame.MaoDireita.PosicaoDaPalma
16             .Subtrair(frame.MaoDireita.PosicaoDaPalma)
17             .Normalizado();
18     }
19 
20     return new[] {0.0, 0.0, 0.0};
21 }
22 private bool MaoDireitaExisteNoFrame(Frame frame)
23 {
24     return frame != null && frame.MaoDireita != null;
25 }
26 private IEnumerable<double> DistanciaMaoEsquerda(Frame frame)
27 {
28     if (MaoEsquerdaExisteNoFrame(frame) &&
MaoEsquerdaExisteNoFrame(PrimeiroFrame))
29     {
30         return PrimeiroFrame.MaoEsquerda.PosicaoDaPalma
31             .Subtrair(frame.MaoEsquerda.PosicaoDaPalma)
32             .Normalizado();
33     }
34 
35     return new[] {0.0, 0.0, 0.0};
36 }
37 private bool MaoEsquerdaExisteNoFrame(Frame frame)
38 {
39     return frame != null && frame.MaoEsquerda != null;
40 }
```

Na linha 05 do Quadro 25 é possível perceber que as características de um *frame* iniciam com o tipo do *frame* que pode ser inicial (`-1`) ou final (`+1`). Além do tipo do *frame*, as características também são compostas pela distância da mão direita em relação ao primeiro *frame* (linha 06) e pela distância da mão esquerda em relação ao primeiro *frame* (linha 07). Por fim, são adicionadas as características padrões de um *frame* (linha 08).

A distância das mãos em relação ao primeiro *frame* é calculada somente se a devida mão existir tanto no primeiro *frame* quanto no *frame* atual (linhas 13 e 28). Caso as mãos não existam em um dos, *frames* um vetor que representa a posição zero é retornado (linhas 20 e 35). O cálculo da distância das mãos é o mesmo, onde é utilizada a posição da palma da mão

no primeiro *frame* subtraindo da posição da mão no *frame* atual (linhas 15 e 16). O valor da distância ainda é normalizado como demonstrado na linha 17.

3.3.8 Treinar algoritmos

O treinamento consiste em reunir as características de todos os sinais em uma estrutura que pode ser utilizada para treinar SVMs e HMMs implementados pelo *framework* Accord.NET. A rotina para reunir as características é implementada através do padrão de projetos `Template Method` com o método `Processar` da classe `DadosAlgoritmoClassificacaoSinais`. Esta classe executa a rotina padrão e delega para as subclasses a execução de rotinas específicas referentes aos dados de sinais dinâmicos, sinais estáticos e *frames* de sinais dinâmicos. O código implementado por esse método pode ser visualizado no Quadro 26.

Quadro 26 – Código do método `Processar`

```
01 public void Processar()
02 {
03     var identificadorDoSinal = 0;
04     QuantidadeClasses = 0;
05     identificadoresSinais = new LinkedList<int>();
06 
07     Inicializar(sinais);
08 
09     foreach (var sinal in sinais)
10     {
11         foreach (var amostra in sinal.Amostras)
12             GerarEntradasESaidasDaAmostra(amostra,
identificadoresSinais, identificadorDoSinal);
13 
14         sinal.IdNoAlgoritmo = identificadorDoSinal;
15         identificadorDoSinal++;
16         QuantidadeClasses += QuantidadeClassesPorSinal;
17     }
18 
19     IdentificadoresSinais = identificadoresSinais.ToArray();
20     Finalizar();
21 }
```

O método `Processar` (Quadro 26) executa a inicialização das subclasses através do método `Inicializar` na linha 07. Posteriormente o método itera sobre a coleção sinais (linha 09) e sobre todas as amostras de cada sinal (linha 11). Para cada amostra, o método delega para as subclasses gerarem as entradas e saídas para os devidos algoritmos (linha 12). Para cada sinal, o método atribui o identificador do sinal no algoritmo ao sinal (linha 14) para que o sinal possa ser reconhecido corretamente e também adiciona a quantidade de classes que um sinal gera para os algoritmos (linha 16). Uma classe para um algoritmo de classificação é um valor de saída para a tarefa de classificar uma amostra. Dessa forma, todo sinal deve gerar ao

menos uma classe. Por fim, o método finaliza a execução através do método `Finalizar` (linha 20) para que as subclasses façam ajustes aos valores gerados.

A classe `DadosSinaisEstaticos` é responsável por gerar os dados de treinamento para o algoritmo de classificação de sinais estáticos e também implementar os métodos abstratos definidos pela classe `DadosAlgoritmoClassificacaoSinais`. O código dessa classe pode ser visto no Quadro 27.

Quadro 27 – Implementação da classe `DadosSinaisEstaticos`

```
01 public override int QuantidadeClassesPorSinal
02 {
03     get { return 1; }
04 }
05 protected override void Inicializar(IEnumerable<Sinal> sinais)
06 {
07     caracteristicasSinais = new LinkedList<double[]>();
08     caracteristicas = new CaracteristicasSinalEstatico();
09 }
10 protected override void GerarEntradasESaidasDaAmostra(IList<Frame>
amostra, LinkedList<int> identificadoresSinais, int identificadorSinal)
11 {
12     var caracteristicasDaAmostra = caracteristicas.DaAmostra(amostra);
13     caracteristicasSinais.AddLast(caracteristicasDaAmostra);
14     identificadoresSinais.AddLast(identificadorSinal);
15 }
16 protected override void Finalizar()
17 {
18     CaracteristicasSinais = caracteristicasSinais.ToArray();
19 }
```

A partir da linha 03 do Quadro 27 é possível perceber que um sinal estático gera apenas uma classe para o algoritmo de classificação. Na linha 05 é demonstrado o método `Inicializar` que inicializa a lista de características dos sinais estáticos (linha 07) e também o gerador de características de sinais estáticos (linha 08). O método `GerarEntradasESaidasDaAmostra` extrai as características da amostra (linha 12), adiciona na lista de características (linha 13) e também adiciona o identificador na lista de identificadores dos sinais. Por fim, o método `Finalizar` transforma a lista de características em um vetor contendo as características de todas as amostras de todos os sinais (linha 18).

A classe `DadosSinaisDinamicos` é responsável por gerar os dados de treinamento para o algoritmo de classificação de sinais dinâmicos e também implementar os métodos abstratos definidos pela classe `DadosAlgoritmoClassificacaoSinais`. O código dessa classe pode ser visto no Quadro 28.

Quadro 28 – Código da classe `DadosSinaisDinamicos`

```
01 public override int QuantidadeClassesPorSinal
02 {
03     get { return 1; }
04 }
05 protected override void Inicializar(IEnumerable<Sinal> sinais)
06 {
07     caracteristicasSinais = new LinkedList<double[][]>();
08     caracteristicas = new CaracteristicasSinalDinamico();
09 }
10 protected override void GerarEntradasESaidasDaAmostra(IList<Frame>
amostra, LinkedList<int> identificadoresSinais, int identificadorSinal)
11 {
12     var caracteristicasDaAmostra = caracteristicas.DaAmostra(amostra);
13     caracteristicasSinais.AddLast(caracteristicasDaAmostra);
14     identificadoresSinais.AddLast(identificadorSinal);
15 }
16 protected override void Finalizar()
17 {
18     CaracteristicasSinais = caracteristicasSinais.ToArray();
19 }
```

A classe `DadosSinaisDinamicos` (Quadro 28) segue a mesma ideia da classe `DadosSinaisEstaticos` (Quadro 27) onde a única mudança é que esta classe utiliza a classe `CaracteristicasSinalDinamico` como pode ser visto na linha 08 do Quadro 28.

Além dos dados de treinamento para os sinais estáticos e para os sinais dinâmicos, também é necessário ter dados para reconhecer individualmente cada *frame* de um sinal dinâmico. Para isso a classe `DadosFramesSinaisDinamicos` implementa os métodos abstratos definidos pela classe `DadosAlgoritmoClassificacaoSinais`. A implementação dessa classe pode ser visualizada no Quadro 29.

Quadro 29 – Implementação da classe `DadosFramesSinaisDinamicos`

```
01 public override int QuantidadeClassesPorSinal
02 {
03     get { return 2; }
04 }
05 protected override void Inicializar(IEnumerable<Sinal> sinais)
06 {
07     quantidadeDeSinais = sinais.Count();
08     caracteristicasSinais = new LinkedList<double[]>();
09     var geradorDeCaracteristicas = new CaracteristicasSinalEstatico();
10     caracteristicasComTipoFrame = new
CaracteristicasSinalEstaticoComTipoFrame(geradorDeCaracteristicas);
11 }
12 protected override void GerarEntradasESaidasDaAmostra(IList<Frame>
amostra, LinkedList<int> identificadoresSinais, int identificadorSinal)
13 {
14     var quantidadeFramesQueRepresentamPrimeiroFrame = amostra.Count/2;
15     caracteristicasComTipoFrame.PrimeiroFrame = amostra.First();
16
17     GerarEntradasESaidasPrimeiroFrame(amostra, identificadoresSinais,
identificadorSinal, quantidadeFramesQueRepresentamPrimeiroFrame);
18     GerarEntradasESaidasUltimoFrame(amostra, identificadoresSinais,
identificadorSinal, quantidadeFramesQueRepresentamPrimeiroFrame);
19 }
20 protected override void Finalizar()
21 {
22     CaracteristicasSinais = caracteristicasSinais.ToArray();
23 }
```

A implementação da classe `DadosFramesSinaisDinamicos` (Quadro 29) difere um pouco das classes `DadosSinaisEstaticos` e `DadosSinaisDinamicos`. Inicialmente, como demonstra a linha 03, um sinal gera duas classes, uma que representa o primeiro *frame* e uma que representa o último *frame*. A inicialização desta classe guarda a quantidade de sinais (linha 07) que será utilizada posteriormente para gerar os identificadores. Além disso, a inicialização também cria uma instância de `CaracteristicasSinalEstaticoComTipoFrame` para extrair as características dos *frames*. A finalização desta classe segue a mesma ideia das classes `DadosSinaisEstaticos` e `DadosSinaisDinamicos` onde a lista é apenas convertida em um vetor (linha 22). A principal diferença entre esta classe e as classes `DadosSinaisEstaticos` e `DadosSinaisDinamicos` está na hora de gerar as entradas e saídas. Inicialmente descobre-se quantos *frames* irão representar o primeiro *frame* (linha 14) e depois é repassado o primeiro *frame* da amostra para a classe que gera as características (linha 15). Por fim, são geradas as entradas e saídas que representam o primeiro *frame* (linha 17) e as entradas e saídas que representam o último *frame* (linha 18).

As entradas e saídas que representam o primeiro *frame* são geradas pelo método `GerarEntradasESaidasPrimeiroFrame` demonstrado no Quadro 30 e as entradas e saídas que representam o último *frame* são geradas pelo método `GerarEntradasESaidasUltimoFrame` demonstrado no Quadro 31.

Quadro 30 – Método `GerarEntradasESaidasPrimeiroFrame`

```
01 private void GerarEntradasESaidasPrimeiroFrame(IList<Frame> amostra,
LinkedList<int> saidas, int indice, int
quantidadeFramesQueRepresentamPrimeiroFrame)
02 {
03      var amostraDoFrame = new Frame[1];
04      caracteristicasComTipoFrame.TipoFrame = TipoFrame.Primeiro;
05      for (int i = 0; i < quantidadeFramesQueRepresentamPrimeiroFrame;
i++)
06      {
07          amostraDoFrame[0] = amostra[i];
08
caracteristicasSinais.AddLast(caracteristicasComTipoFrame.DaAmostra(amostra
DoFrame));
09          saidas.AddLast(indice);
10      }
11 }
```

A partir do Quadro 30 é possível observar que as características extraídas são associadas como primeiro *frame* (linha 04). Também é possível perceber que os primeiros *frames* da amostra são associados como sendo o primeiro *frame* (linha 05). A partir desse ponto, todos os primeiros *frames* da amostra geram características que irão representar o primeiro *frame* (linha 08) e também é adicionado o identificador do sinal para cada característica (linha 09).

Quadro 31 – Método `GerarEntradasESaidasUltimoFrame`

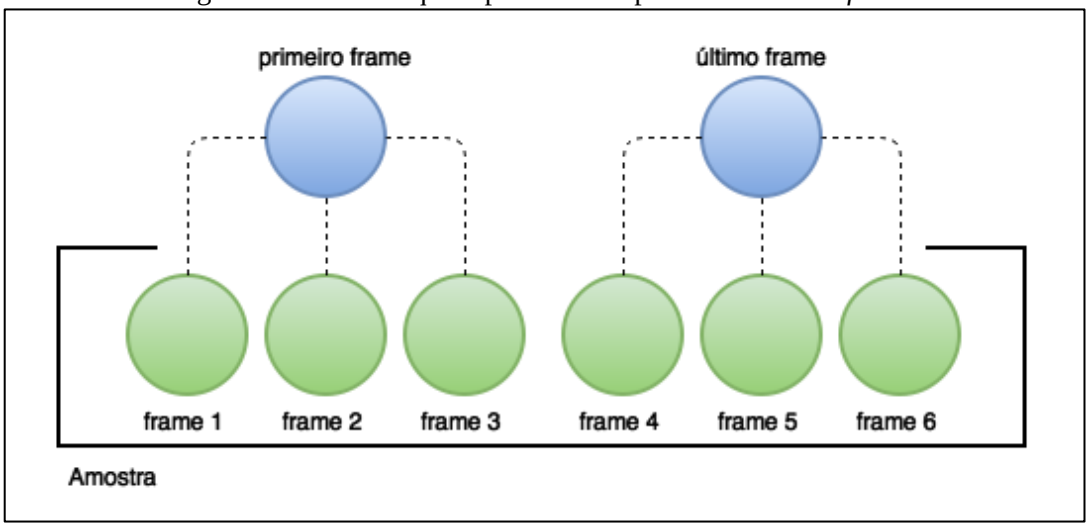
```
01 private void GerarEntradasESaidasUltimoFrame(IList<Frame> amostra,
LinkedList<int> saidas, int indice, int
quantidadeFramesQueRepresentamPrimeiroFrame)
02 {
03      var amostraDoFrame = new Frame[1];
04      caracteristicasComTipoFrame.TipoFrame = TipoFrame.Ultimo;
05      for (int i = quantidadeFramesQueRepresentamPrimeiroFrame; i <
amostra.Count; i++)
06      {
07          amostraDoFrame[0] = amostra[i];
08
caracteristicasSinais.AddLast(caracteristicasComTipoFrame.DaAmostra(amostra
DoFrame));
09          saidas.AddLast(indice + quantidadeDeSinais);
10      }
11 }
```

A partir do Quadro 31 é possível observar que as características extraídas são associadas como último *frame* (linha 04). Também é possível perceber que os últimos *frames* da amostra são associados como sendo o último *frame* (linha 05). A partir desse ponto, todos os últimos *frames* da amostra geram características que irão representar o último *frame* (linha 08) e também é adicionado o identificador do sinal para cada característica (linha 09). Como o objetivo é diferenciar entre o primeiro e o último *frame*, estes devem ter identificadores diferentes. Portanto, o identificador do último *frame* é o identificador do sinal somado com a

quantidade de sinais. Por exemplo, se o identificador do sinal for três e existirem nove sinais, o identificador do último *frame* será doze.

A Figura 35 apresenta visualmente a ideia de que uma parte dos *frames* representa o primeiro *frame* e a outra parte dos *frames* representa o último *frame*.

Figura 35 – *Frames* que representam o primeiro e último *frame*



A Figura 35 apresenta o exemplo de uma amostra de um sinal dinâmico com seis *frames*. O total de *frames* é dividido ao meio e a primeira metade destes *frames* representa o primeiro *frame* e a segunda metade representa o último *frame*. Portanto, os *frames* `frame 1`, `frame 2` e `frame 3` representam o primeiro *frame* e os *frames* `frame 4`, `frame 5` e `frame 6` representam o último *frame*.

O treinamento do algoritmo SVM ocorre através do método `Aprender` disponibilizado pela interface `IAlgoritmoClassificacaoSinaisEstaticos`. O código deste método pode ser visualizado no Quadro 32.

Quadro 32 – Código do método `Aprender` da classe `Svm`

```
01 public void Aprender(IDadosSinaisEstaticos dados)
02 {
03     var kernel = new Polynomial(degree: 3, constant: 1);
04     svm = new
MulticlassSupportVectorMachine(QuantidadeIndeterminadaDeCaracteristicas,
kernel, dados.QuantidadeClasses);
05
06     var teacher = new MulticlassSupportVectorLearning(svm,
dados.CaracteristicasSinais, dados.IdentificadoresSinais)
07     {
08         Algorithm = (machine, classInputs, classOutputs, j, k) =>
09             new SequentialMinimalOptimization(machine, classInputs,
classOutputs)
10             {
11                 Complexity = 1
12             }
13     };
14
15     teacher.Run();
16 }
```

A SVM treinada utiliza uma função *kernel* polinomial (linha 03) e o método de decisão um-contra-um através da instância de `MulticlassSupportVectorMachine` na linha 04. A instância de `MulticlassSupportVectorMachine` ainda requer a quantidade de características contida em cada vetor, neste caso a quantidade é zero, que representa uma quantidade indeterminada de características. Além da quantidade de características, também é necessário informar a quantidade de classes diferentes. O treinamento da SVM ocorre através da classe `MulticlassSupportVectorLearning` (linha 06) que precisa da instância da SVM, das características de todos os sinais e também do identificador de cada característica. Essa classe ainda precisa do algoritmo que será utilizado para treinamento, neste caso, o algoritmo utilizado é o `SequentialMinimalOptimization` como pode ser visto na linha 09. Por fim, é executado o algoritmo para treinamento na linha 15. A propriedade `Complexity` da classe `SequentialMinimalOptimization` indica o parâmetro custo (`C`) das SVMs. Quanto maior o valor de `C` mais preciso é o modelo que pode levar as SVMs a não generalizar corretamente.

O treinamento de um HMM ocorre através do método `Aprender` da classe `Hmm`. O código desse método pode ser visto no Quadro 33.

Quadro 33 – Método `Aprender` da classe `Hmm`

```
01 private const int QuantidadeEstados = 5;
02 public void Aprender(IDadosSinaisDinamicos dados)
03 {
04     var quantidadeCaracteristicas =
dados.CaracteristicasSinais[0][0].Length;
05     hmm = new HiddenMarkovClassifier<MultivariateNormalDistribution>(
06         classes: dados.QuantidadeClasses,
07         topology: new Forward(QuantidadeEstados),
08         initial: new
MultivariateNormalDistribution(quantidadeCaracteristicas)
09     );
10
11     var teacher = new
HiddenMarkovClassifierLearning<MultivariateNormalDistribution>(hmm,
12         modelIndex => new
BaumWelchLearning<MultivariateNormalDistribution>(hmm.Models[modelIndex])
13         {
14             Tolerance = 0.001,
15             Iterations = 100,
16             FittingOptions = new NormalOptions { Regularization = 1e-5}
17         });
18
19     teacher.Run(dados.CaracteristicasSinais,
dados.IdentificadoresSinais);
20 }
```

O método `Aprender` (Quadro 33) inicia um classificador de HMM utilizando distribuições normais multivariadas para as características (linha 05). Para iniciar o classificador é necessário informar a quantidade de classes (linha 06), a topologia do HMM, que nesse caso é Bakis (linha 07) e as probabilidades iniciais (linha 08). A inicialização das probabilidades leva em consideração a quantidade de características utilizadas para treinamento (linhas 04 e 08). Na linha 11 é inicializado o algoritmo que fará o treinamento do classificador de HMM. O algoritmo de treinamento utiliza internamente o algoritmo Baum-Welch para treinar os HMMs. Para iniciar o algoritmo Baum-Welch é necessário informar a tolerância para convergência (linha 14), o número máximo de iterações (linha 15) e valores de adaptação do *framework* Accord.NET (linha 16). Finalmente, na linha 19 os HMMs são treinados utilizando as características e os identificadores dos sinais.

3.3.9 Classificação e reconhecimento

O reconhecimento de sinais parte da camada cliente pelo método `_onFrame` na classe `ReconhecedorDeSinais`. O Quadro 34 demonstra esse método.

Quadro 34 – Método `_onFrame` que faz o reconhecimento de um sinal

```
01 _onFrame: function(frame) {
02   this._estado
03       .reconhecer(frame)
04       .then(function(sinalFoiReconhecido) {
05           if (sinalFoiReconhecido) {
06               this._eventEmitter.trigger(EVENTO_RECONHECEU_SINAL);
07           }
08       }.bind(this));
09 }
```

O método `_onFrame` (Quadro 34) delega para o estado interno da classe o reconhecimento do sinal (linhas 02 e 03) e trata o retorno do reconhecimento. Se o reconhecimento do sinal ocorrer com sucesso então é disparado um evento indicando que um sinal foi reconhecido (linha 06).

A classe pode estar no estado `ReconhecedorDeSinaisOffline` ou `ReconhecedorDeSinaisOnline`. O estado `ReconhecedorDeSinaisOffline` é o estado padrão enquanto não foi estabelecida a conexão com o servidor. Quando a conexão com o servidor é estabelecida, o estado muda para `ReconhecedorDeSinaisOnline` e este estado é responsável pelo reconhecimento de sinais. O Quadro 35 demonstra o código do método `reconhecer` da classe `ReconhecedorDeSinaisOnline`.

Quadro 35 – Método `reconhecer` da classe `ReconhecedorDeSinaisOnline`

```
01 reconhecer: function(frame) {
02     if (this._idDoSinalParaReconhecer === -1)
03         return Promise.resolve(false);
04
05     var dados = this._informacoesDoFrame.extrairParaAmostra(frame);
06     return this._algoritmo
07         .reconhecer(dados)
08         .then(function(sinalFoiReconhecido) {
09             if (sinalFoiReconhecido) {
10                 this._idDoSinalParaReconhecer = -1;
11             }
12             return sinalFoiReconhecido;
13         }.bind(this));
14 }
```

O método `reconhecer` (Quadro 35) verifica se há um sinal para ser reconhecido (linha 02), caso não exista então ele retorna `false` (linha 03) indicando que o sinal não foi reconhecido com sucesso. Caso exista um sinal para ser reconhecido, o *frame* do Leap Motion é adaptado para uma instância de `Frame` (linha 05). Com os dados preparados, o reconhecimento do sinal é delegado para o devido algoritmo, que pode ser `SinalEstatico` ou `SinalDinamico` (linhas 06 e 07). Se o sinal foi reconhecido com sucesso o estado remove o identificador do sinal que deveria ser reconhecido (linha 10).

3.3.9.1 Reconhecimento de sinais estáticos

O reconhecimento de um sinal estático ocorre de acordo com a Figura 47 do apêndice B. O método `reconhecer` da classe `SinalEstatico` demonstrado no Quadro 36 é responsável pelo reconhecimento de um sinal estático.

Quadro 36 – Método `reconhecer` da classe `SinalEstatico`

```
01 reconhecer: function(frame) {
02       var amostra = [frame];
03       return this._hub.reconhecer(this._sinalId, amostra);
04 }
```

O método `reconhecer` (Quadro 36) transforma o *frame* em uma amostra (linha 02) e envia para o servidor reconhecer o sinal através do protocolo TCP com o *framework* ASP.NET SignalR (linha 03).

No servidor o reconhecimento de um sinal estático ocorre na classe `GerenciadorSinais` porque a rotina de reconhecimento é idêntica para sinais estáticos e dinâmicos. O Quadro 37 apresenta o método `Reconhecer`.

Quadro 37 – Método `Reconhecer` da classe `GerenciadorSinais`

```
01 public bool Reconhecer(int idSinal, IList<Frame> amostra)
02 {
03       return idSinal ==
algoritmoClassificacaoSinais.Classificar(amostra);
04 }
```

O método `Reconhecer` (Quadro 36) recebe o identificador do sinal e a amostra que deve ser reconhecida. Ele então executa a classificação do sinal através do método `Classificar` da interface `IAlgoritmoClassificacaoSinais` e compara com o identificador do sinal esperado (linha 03).

A implementação do método `Classificar` para sinais estáticos está na classe `Svm` e o código pode ser visualizado no Quadro 38.

Quadro 38 – Método `Classificar` da classe `Svm`

```
01 public int Classificar(IList<Frame> frame)
02 {
03       if (svm == null)
04       {
05           throw new InvalidOperationException("É necessário treinar o
algoritmo antes de reconhecer");
06       }
07       var caracteristicasDoSinal = caracteristicas.DaAmostra(frame);
08       return svm.Compute(caracteristicasDoSinal,
MulticlassComputeMethod.Elimination);
09 }
```

O método `Classificar` (Quadro 38) inicialmente verifica se o algoritmo já foi treinado (linha 03) e caso ainda não tenha sido treinado, lança uma exceção (linha 05). Se o algoritmo já estiver treinado, o método extrai as características da amostra (linha 07) e envia

para o algoritmo SVM implementado pelo *framework* Accord.NET classificar a amostra utilizando o método eliminação por DDAG (linha 08).

3.3.9.2 Reconhecimento de sinais dinâmicos

O reconhecimento de um sinal dinâmico ocorre de acordo com os diagramas da Figura 48 e Figura 49 do apêndice B. O método `reconhecer` da classe `SinalDinamico`, demonstrado no Quadro 39, é responsável pelo reconhecimento de um sinal dinâmico.

Quadro 39 – Método `reconhecer` da classe `SinalDinamico`

```
01 reconhecer: function(frame) {
02     if (this._comparador.framesSaoIguais(frame, this._ultimoFrame)) {
03         return Promise.resolve(false);
04     }
05
06     var estado = this._estado,
07         amostra = [frame];
08
09     this._ultimoFrame = frame;
10     this.reconhecendo(amostra);
11
12     return estado.reconhecer(amostra);
13 }
```

O método `reconhecer` (Quadro 39) verifica se o *frame* que deve ser reconhecido é similar ao anterior (linha 02) e, caso o *frame* seja similar, o método retorna `false` (linha 03). Essa abordagem foi utilizada para que exista ao menos algum movimento entre os *frames* que são enviados para reconhecimento. Se houve movimento entre os *frames*, então o *frame* é transformado em uma amostra (linha 07), armazenado (linha 09), o estado é alterado para reconhecendo (linha 10) e a amostra é enviada para reconhecimento com o estado anterior (linha 12).

A classe `Comparador` é responsável por verificar se dois *frames* são semelhantes. A comparação é necessária para que seja feito ao menos algum movimento para reconhecer um sinal dinâmico e também para que não sejam classificados *frames* muito semelhantes. O método `framesSaoIguais` é quem faz essa verificação e o seu código pode ser visto no Quadro 40.

Quadro 40 – Código do método `framesSaoIguais`

```
01 framesSaoIguais: function(frameA, frameB) {
02     if (!frameA || !frameB) {
03         return false;
04     }
05     return this._maosSaoIguais(frameA.MaoEsquerda, frameB.MaoEsquerda)
06         && this._maosSaoIguais(frameA.MaoDireita, frameB.MaoDireita);
07 },
08 _maosSaoIguais: function(maoA, maoB) {
09     if (maoA === maoB) {
10         return true;
11     } else if (!maoA || !maoB) {
12         return false;
13     }
14     var dedosMaoA = maoA.Dedos;
15     var dedosMaoB = maoB.Dedos;
16     return this._dedosEstaoNaMesmaPosicao(dedosMaoA, dedosMaoB);
17 }
```

O método `framesSaoIguais` (Quadro 40) verifica se os dois *frames* que devem ser comparados existem (linha 02).Se ao menos um deles não existir, os *frames* são considerados diferentes. Para que os *frames* sejam semelhantes, são comparados os valores das duas mãos dos dois *frames* (linhas 05 e 06). A comparação de semelhança entre as mãos é feita pelo método `_maosSaoIguais` (linha 08) que verifica se elas são iguais (linha 09) ou então se ao menos uma delas não existe (linha 11). Caso as duas mãos existam e não sejam iguais, então são comparadas as posições dos dedos de ambas as mãos com o método `_dedosEstaoNaMesmaPosicao` (linha 16). O método `_dedosEstaoNaMesmaPosicao` é demonstrado no Quadro 41.

Quadro 41 – Métodos para fazer a comparação entre a posição dos dedos

```
01 _dedosEstaoNaMesmaPosicao: function(dedosMaoA, dedosMaoB) {
02     for (var i = 0; i < dedosMaoA.length; i++) {
03         var posicaoDedoMaoA = dedosMaoA[i].PosicaoDaPonta;
04         var posicaoDedoMaoB = dedosMaoB[i].PosicaoDaPonta;
05 
06         if (!arraysSaoIguais(posicaoDedoMaoA, posicaoDedoMaoB)) {
07             return false;
08         }
09     }
10     return true;
11 }
12 function arraysSaoIguais(arrayA, arrayB) {
13     for (var i = 0; i < arrayA.length; i++) {
14         if (!numerosSaoIguais(arrayA[i], arrayB[i])) {
15             return false;
16         }
17     }
18     return true;
19 }
20 var LIMITE_DIFERENCA_ENTRE_NUMEROS = 10;
21 function numerosSaoIguais(numeroA, numeroB) {
22     var diferenca = Math.abs(numeroA - numeroB);
23     return diferenca < LIMITE_DIFERENCA_ENTRE_NUMEROS;
24 }
```

O método `_dedosEstaoNaMesmaPosicao` (Quadro 41) verifica se a posição dos dedos é semelhante entre as mãos dos *frames* (linha 02). Como a posição de um dedo é representada por um vetor, é necessário fazer a comparação do vetor inteiro (linha 06). Se ao menos um dos dedos estiver em uma posição diferente, o método indica que os dedos estão em posições diferentes (linha 07), senão o método indica que eles estão em posições semelhantes (linha 10). A comparação dos valores de um vetor é feita no método `arraysSaoIguais` (linha 12) que indica se todos os valores do vetor são semelhantes (linha 18) ou não (linha 15). Para isso, esse método verifica se os valores do vetor são semelhantes considerando uma margem e essa comparação é feita no método `numerosSaoIguais` (linha 21). Este método obtém o valor não negativo da diferença de dois números (linha 22) e compara com um valor limite para indicar se houve ou não movimento (linha 23).

A classe `SinalDinamico` possui quatro possíveis estados para o reconhecimento de um sinal dinâmico que são implementados pelas classes: `SinalDinamicoNaoReconheceuFrame`, `SinalDinamicoReconhecendo`, `SinalDinamicoReconheceuPrimeiroFrame` e `SinalDinamicoReconheceuUltimoFrame`.

O estado `SinalDinamicoReconhecendo` é um estado intermediário entre os outros três estados do reconhecimento. Ele é utilizado enquanto os demais estados estão efetuando o reconhecimento de um sinal dinâmico e funciona como uma espécie de *buffer* que armazena

os *frames* que passaram enquanto outro *frame* estava sendo reconhecido. O código do método `reconhecer` pode ser visto no Quadro 42.

Quadro 42 – Implementação de *buffer* da classe `SinalDinamicoReconhecendo`

```
01 reconhecer: function(amostra) {
02     this.adicionar(amostra);
03     return Promise.resolve(false);
04 },
05
06 adicionar: function(amostra) {
07     this._frames.push(amostra[0]);
08     if (this._frames.length === QUANTIDADE_MAXIMA) {
09         this._algoritmoDeSinalDinamico.naoReconheceuFrame();
10         this._frames = [];
11     }
12 },
```

O método `reconhecer` (Quadro 42) adiciona a amostra no *buffer* interno (linha 02) e retorna `false` (linha 03) porque o sinal não foi reconhecido. O método `adicionar` armazena o primeiro *frame* da amostra no vetor de *frames* (linha 07) e verifica se o vetor ficou cheio (linha 08). Se o vetor ficou cheio, então `SinalDinamico` passa para o estado `SinalDinamicoNaoReconheceuFrame` (linha 09) e o vetor é esvaziado (linha 10). A mudança de estado ocorre porque dentro da quantidade máxima de *frames* que podiam ser armazenados, o último *frame* não foi reconhecido e, portanto, tem que reconhecer o primeiro *frame* novamente.

A classe `SinalDinamicoNaoReconheceuFrame` é o estado inicial de `SinalDinamico`, e esse estado é responsável pelo reconhecimento do primeiro *frame* de um sinal dinâmico. O código do método `reconhecer` dessa classe pode ser visto no Quadro 43.

Quadro 43 – Método `reconhecer` da classe `SinalDinamicoNaoReconheceuFrame`

```
01 reconhecer: function(amostraPrimeiroFrame) {
02     var algoritmoDeSinalDinamico = this._algoritmoDeSinalDinamico;
03     var idSinal = algoritmoDeSinalDinamico.getSinalId();
04
05     return Signa.Hubs.sinaisDinamicos()
06         .reconhecerPrimeiroFrame(idSinal, amostraPrimeiroFrame)
07         .then(function(reconheceuAmostraComoPrimeiroFrameDoSinal) {
08             if (reconheceuAmostraComoPrimeiroFrameDoSinal) {
09
algoritmoDeSinalDinamico.setAmostraPrimeiroFrame(amostraPrimeiroFrame);
10                 algoritmoDeSinalDinamico.reconheceuPrimeiroFrame();
11                 return false;
12             }
13             algoritmoDeSinalDinamico.naoReconheceuFrame();
14             return false;
15         });
16 }
```

O método `reconhecer` (Quadro 43) se comunica com o servidor utilizando o *framework* ASP.NET SignalR para fazer o reconhecimento do primeiro *frame* de um sinal

dinâmico (linhas 05 e 06). Se o reconhecimento do primeiro *frame* ocorreu com sucesso, então é armazenado o primeiro *frame* do sinal dinâmico (linha 09) e o estado de `SinalDinamico` é alterado para `SinalDinamicoReconheceuPrimeiroFrame` (linha 10). Se não reconheceu o primeiro *frame*, o estado se mantém como `SinalDinamicoNaoReconheceuFrame` (linha 13). O retorno do reconhecimento é sempre `false` (linhas 11 e 14) porque o sinal ainda não foi reconhecido, apenas o primeiro *frame* foi reconhecido.

O reconhecimento do primeiro *frame* de um sinal dinâmico ocorre na classe `GerenciadorSinaisDinamicos` com o método `ReconhecerPrimeiroFrame`. O código deste método pode ser visto no Quadro 44.

Quadro 44 – Método `ReconhecerPrimeiroFrame` da classe `GerenciadorSinaisDinamicos`

```
01 public bool ReconhecerPrimeiroFrame(int idSinal, IList<Frame> amostra)
02 {
03   caracteristicas.PrimeiroFrame = null;
04   caracteristicas.TipoFrame = TipoFrame.Primeiro;
05   return idSinal ==
algoritmoClassificacaoSinaisEstaticos.Classificar(amostra);
06 }
```

O método `ReconhecerPrimeiroFrame` (Quadro 44) tem um comportamento similar ao método `Reconhecer` de um sinal estático visto no Quadro 37. Esse método adiciona os valores necessários para a extração de características, como por exemplo: informa que não existe o primeiro *frame* (linha 03) e que o tipo do *frame* que está sendo reconhecido é o primeiro (linha 04). O reconhecimento ocorre da mesma forma, visto que ambos utilizam SVMs para classificação.

O próximo estado no reconhecimento de um sinal dinâmico é o estado `SinalDinamicoReconheceuPrimeiroFrame`. Este estado é responsável pelo reconhecimento do último *frame* de um sinal dinâmico. O código do método `reconhecer` pode ser visto no Quadro 45.

Quadro 45 – Método `reconhecer` da classe `SinalDinamicoReconheceuPrimeiroFrame`

```
01 reconhecer: function(amostraUltimoFrame) {
02     var algoritmoDeSinalDinamico = this._algoritmoDeSinalDinamico;
03     var amostraPrimeiroFrame =
algoritmoDeSinalDinamico.getAmostraPrimeiroFrame();
04     var idSinal = algoritmoDeSinalDinamico.getSinalId();
05
06     return Signa.Hubs.sinaisDinamicos()
07         .reconhecerUltimoFrame(idSinal, amostraPrimeiroFrame,
amostraUltimoFrame)
08         .then(function(reconheceuAmostraComoUltimoFrameDoSinal) {
09             if (reconheceuAmostraComoUltimoFrameDoSinal) {
10                 algoritmoDeSinalDinamico.reconheceuUltimoFrame();
11                 return false;
12             }
13             algoritmoDeSinalDinamico.reconheceuPrimeiroFrame();
14             return false;
15         });
16 }
```

O método `reconhecer` (Quadro 45) inicialmente busca a amostra do primeiro *frame* que foi reconhecido (linha 03) e envia esta amostra para efetuar o reconhecimento do último *frame* no servidor (linhas 06 e 07). Se o último *frame* for reconhecido, o estado de `SinalDinamico` passa a ser `SinalDinamicoReconheceuUltimoFrame` (linha 10), caso contrário o estado se mantém como `SinalDinamicoReconheceuPrimeiroFrame`. Da mesma forma que no estado `SinalDinamicoNaoReconheceuPrimeiroFrame` o retorno do reconhecimento é sempre `false` (linhas 11 e 14) porque o sinal ainda não foi reconhecido, apenas o primeiro e o último *frame*.

No servidor o reconhecimento do último *frame* de um sinal dinâmico também ocorre na classe `GerenciadorSinaisDinamicos`. O método utilizado para o reconhecimento é `ReconhecerUltimoFrame`, que pode ser visto no Quadro 46.

Quadro 46 – Método `ReconhecerUltimoFrame` da classe `GerenciadorSinaisDinamicos`

```
01 public bool ReconhecerUltimoFrame(int idSinal, IList<Frame>
amostraPrimeiroFrame, IList<Frame> amostraUltimoFrame)
02 {
03   caracteristicas.PrimeiroFrame = amostraPrimeiroFrame[0];
04   caracteristicas.TipoFrame = TipoFrame.Ultimo;
05   var idSinalUltimoFrame = idSinal + repositorio.Count(s => s.Tipo ==
TipoSinal.Dinamico);
06   return idSinalUltimoFrame ==
algoritmoClassificacaoSinaisEstaticos.Classificar(amostraUltimoFrame);
07 }
```

O método `ReconhecerUltimoFrame` (Quadro 46) funciona da mesma forma que o método `ReconhecerPrimeiroFrame`, descrito no Quadro 45. A diferença é que esse método utiliza o primeiro *frame* já reconhecido na extração de características (linha 03) e informa que o *frame* que está sendo reconhecido é o último (linha 04). Esse método também precisa

ajustar o identificador que deve ser reconhecido para ser o identificador do último *frame* (linha 05).

O estado `SinalDinamicoReconheceuUltimoFrame` é responsável pelo reconhecimento do sinal dinâmico como um todo, ou seja, pela sequência de *frames*. O método `reconhecer` deste estado pode ser visto no Quadro 47.

Quadro 47 – Método `reconhecer` da classe `SinalDinamicoReconheceuUltimoFrame`

```
01 reconhecer: function() {
02     var algoritmoDeSinalDinamico = this._algoritmoDeSinalDinamico;
03     var framesParaReconhecer = this._buffer.getFrames();
04     var idSinal = algoritmoDeSinalDinamico.getSinalId();
05
06     return Signa.Hubs.sinaisDinamicos()
07         .reconhecer(idSinal, framesParaReconhecer)
08         .then(function(sinalReconhecido) {
09             algoritmoDeSinalDinamico.naoReconheceuFrame();
10             return sinalReconhecido;
11         });
12 }
```

O método `reconhecer` (Quadro 47) busca todos os *frames* que estão no *buffer* (linha 03) e os envia ao servidor para que seja feito o reconhecimento da sequência que forma o sinal dinâmico (linhas 06 e 07). Após a execução do reconhecimento, o estado de `SinalDinamico` sempre volta para `SinalDinamicoNaoReconheceuFrame`, independentemente do resultado do reconhecimento (linha 09). Esse método faz o retorno correto do reconhecimento, informando que o sinal foi reconhecido corretamente ou que não foi possível reconhecer o sinal (linha 10).

No servidor, o reconhecimento de um sinal dinâmico ocorre da mesma forma que o reconhecimento de um sinal estático, através do método `Reconhecer` da classe `GerenciadorSinais`. A diferença é que neste caso é utilizado o um HMM para o reconhecimento como demonstrado no Quadro 48.

Quadro 48 – Método `Classificar` da classe `Hmm`

```
01 public int Classificar(IList<Frame> amostra)
02 {
03      if (hmm == null)
04          throw new InvalidOperationException();
05
06      var caracteristicasDoSinal = caracteristicas.DaAmostra(amostra);
07      return hmm.Compute(caracteristicasDoSinal); ;
08 }
```

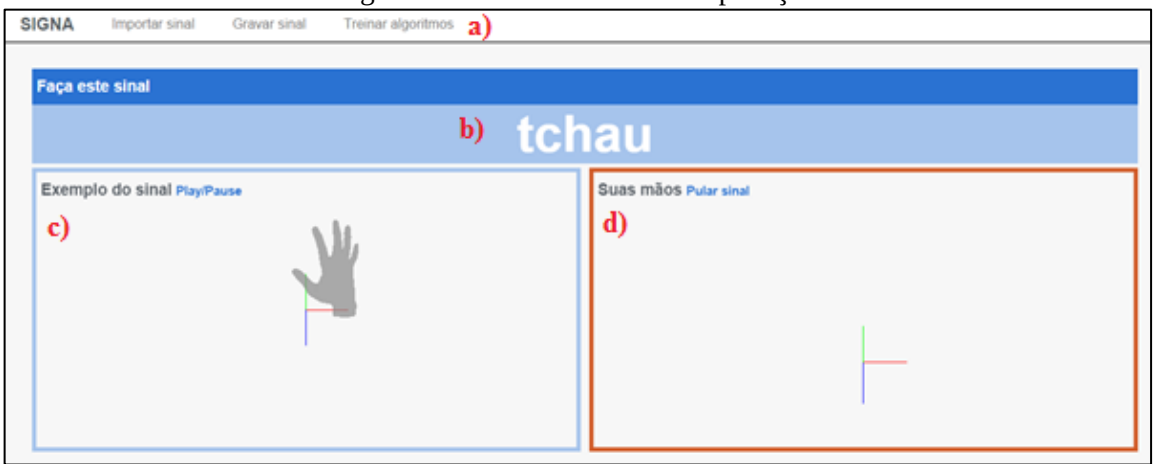
O método `Classificar` (Quadro 48) é semelhante ao método `Classificar` da classe `Svm`. Primeiramente verifica-se se o algoritmo foi treinado e, caso não tenha sido treinado uma exceção é lançada (linha 04). Com o algoritmo treinado então são extraídas as características

da amostra do sinal dinâmico (linha 06) e feito o reconhecimento utilizando a implementação do algoritmo pelo *framework* Accord.NET (linha 07).

3.3.10 Operacionalidade da implementação

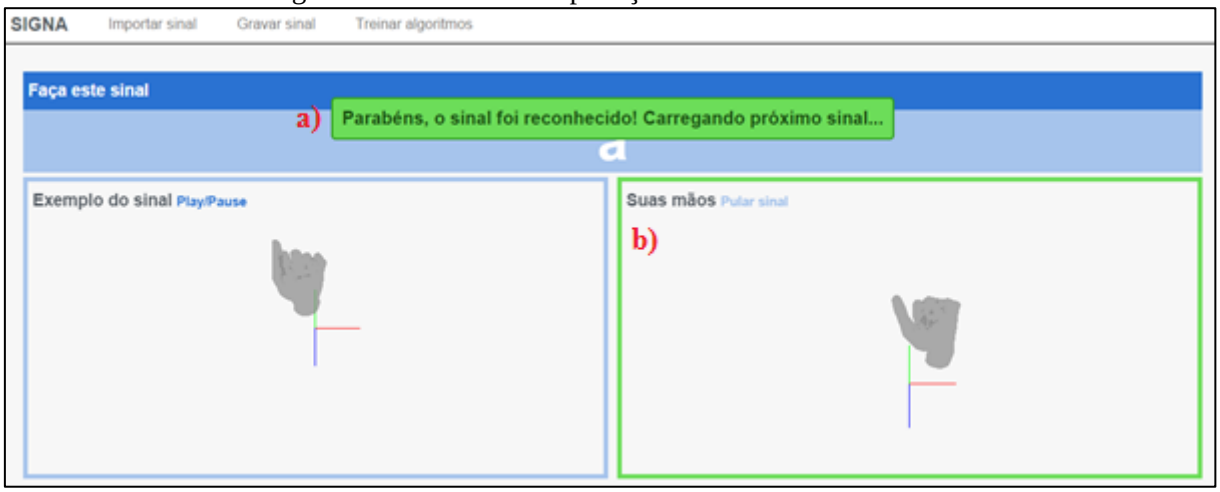
Para acessar a aplicação o usuário deve estar com o Leap Motion conectado e com um navegador web, ele deve acessar a página inicial da aplicação como a Figura 36.

Figura 36 – Interface inicial da aplicação



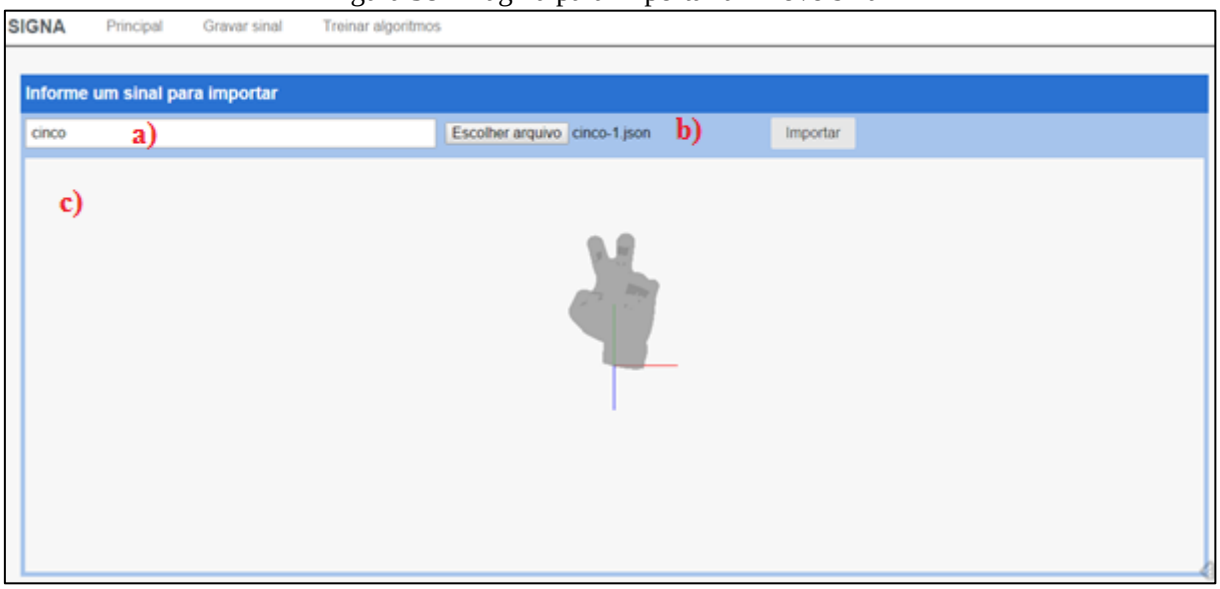
A Figura 36a demonstra o menu principal da aplicação, pelo qual é possível navegar entre as páginas da aplicação. A Figura 36b demonstra a descrição do sinal que o usuário deve reproduzir. A Figura 36c demonstra a seção que exibe um exemplo do sinal que deve ser reproduzido. Caso o sinal contenha movimento, o modelo 3D do exemplo apresenta o mesmo movimento esperado do sinal. Essa região possui controles de câmera permitindo ações como rotação e zoom para ter uma melhor visualização do sinal. A Figura 36d demonstra a região onde o modelo 3D da mão do usuário é exibido enquanto ele reproduz o sinal. Durante a reprodução do sinal, se o sinal for reconhecido pela aplicação a interface fica como na Figura 37.

Figura 37 – Interface da aplicação ao reconhecer um sinal



Quando o sinal reproduzido pelo usuário é reconhecido como o sinal esperado pela aplicação, a mensagem da Figura 37a é exibida e também o contorno da Figura 37b muda de vermelho para verde. Além do reconhecimento de um sinal, é possível importar novos sinais ou adicionar novos exemplos para sinais existentes. A interface que permite essas operações é apresentada na Figura 38.

Figura 38 – Página para importar um novo sinal



Para importar um sinal ou adicionar um novo exemplo de sinal, é necessário informar o nome do sinal e também o arquivo gravado através da ferramenta Leap Recorder com o exemplo do sinal. Um exemplo de como gravar esse arquivo pode ser visto no apêndice D. A Figura 38a demonstra o campo que recebe o nome do sinal e a Figura 38b apresenta o campo para selecionar o arquivo com o exemplo do sinal. Quando o exemplo do sinal é carregado, este é exibido como pré-visualização como demonstrado na Figura 38c. A Figura 38c também apresenta os mesmos controles de câmera da Figura 36c permitindo movimentar a câmera para visualizar se o sinal que está sendo importado está correto.

3.4 RESULTADOS E DISCUSSÕES

Nesta seção são apresentados os experimentos feitos com a aplicação. A seção 3.4.1 demonstra o experimento para adquirir dados dos sinais com o Leap Motion. A seção 3.4.2 demonstra os experimentos utilizando diferentes funções *kernel* para as SVMs. A seção 3.4.3 apresenta o experimento de validação cruzada visando o desempenho de classificação dos algoritmos com as amostras coletadas. A seção 3.4.4 demonstra o experimento de desempenho do treinamento dos algoritmos. A seção 3.4.5 apresenta o experimento de

usabilidade da aplicação. Finalmente, a 3.4.6 apresenta a comparação com os trabalhos correlatos.

3.4.1 Experimento 01: Aquisição de dados através do Leap Motion

O experimento de aquisição de dados dos sinais com o Leap Motion teve a finalidade de verificar quais sinais o Leap Motion conseguiria capturar e prover informações. Foi montado um conjunto com 49 sinais para este teste. Os sinais selecionados foram: as letras do alfabeto datilológico (Figura 2), os números (Figura 3) e também os sinais boa noite, comprar, nome, reais, querer, hoje, onde, poder, salto alto, tênis, oi, tchau. Deste conjunto, os sinais das letras I, J, K, X, Y, Z, Ç, do número oito e das palavras boa noite, comprar, nome, reais, querer, onde, hoje, poder, salto alto, tênis, oi e tchau possuem movimento. Os sinais boa noite, comprar, querer, hoje, onde, poder, salto alto e tênis utilizam as duas mãos.

Para a realização destes testes, o autor deste trabalho utilizou a ferramenta Leap Recorder para reproduzir os sinais. Essa ferramenta foi escolhida porque é utilizada para gravar os sinais que são adicionados na aplicação. Desta forma, caso o Leap Recorder não consiga reproduzir um sinal, ele não pode ser adicionado na aplicação.

Foram utilizadas duas descrições para informar se um sinal pode ou não ser reproduzido. Se o sinal pode ser reproduzido de forma fácil, então o resultado foi OK. Caso o sinal não possa ser reproduzido ou foi muito difícil para reproduzi-lo, então o resultado foi NÃO OK. Portanto, apenas os sinais com resultado OK podem ser adicionados na aplicação sem afetar o uso da mesma pelos usuários. O Quadro 49 apresenta os resultados obtidos para cada sinal.

Quadro 49 – Resultado da aquisição de dados dos sinais com o Leap Motion

Sinal	Possui movimento	Número de mãos	Resultado	Sinal	Possui movimento	Número de mãos	Resultado
A	NÃO	1	OK	Z	SIM	1	OK
B	NÃO	1	OK	Ç	SIM	1	OK
C	NÃO	1	OK	1	NÃO	1	OK
D	NÃO	1	NÃO OK	2	NÃO	1	OK
E	NÃO	1	OK	3	NÃO	1	OK
F	NÃO	1	NÃO OK	4	NÃO	1	OK
G	NÃO	1	NÃO OK	5	NÃO	1	OK
H	SIM	1	NÃO OK	6	NÃO	1	NÃO OK
I	NÃO	1	OK	7	NÃO	1	NÃO OK
J	SIM	1	OK	8	SIM	1	OK
K	SIM	1	NÃO OK	9	NÃO	1	OK
L	NÃO	1	OK	0	NÃO	1	OK
M	NÃO	1	NÃO OK	COMPRAR	SIM	2	NÃO OK
N	NÃO	1	NÃO OK	NOME	SIM	1	OK
O	NÃO	1	OK	REAIS	SIM	1	OK
P	NÃO	1	NÃO OK	QUERER	SIM	2	OK
Q	NÃO	1	OK	HOJE	SIM	2	OK
R	NÃO	1	NÃO OK	ONDE	SIM	2	OK
S	NÃO	1	OK	PODER	SIM	2	OK
T	NÃO	1	NÃO OK	OI	SIM	1	OK
U	NÃO	1	OK	TCHAU	SIM	1	OK
V	NÃO	1	OK	TÊNIS	SIM	2	OK
W	NÃO	1	OK	BOA NOITE	SIM	2	NÃO OK
X	SIM	1	OK	SALTO ALTO	SIM	2	OK
Y	SIM	1	NÃO OK				

A partir do Quadro 49 é possível perceber que 34 sinais podem ser reproduzidos através do Leap Motion sem muita dificuldade. Entre estes sinais é possível perceber que existem sinais com uma ou duas mãos e com ou sem movimento.

De forma geral, os sinais que obtiveram o resultado `NÃO OK` apresentam um nível de oclusão em relação ao Leap Motion. Como exemplo, é possível citar o sinal do `número seis`, que é um sinal feito com uma mão e não apresenta movimento, mas há oclusão dos dedos do ponto de vista do Leap Motion. Outros sinais, como a letra `M`, possuem uma configuração de mãos simples, mas devido a mão estar em uma posição diferente da padrão de uso do Leap Motion pode gerar uma perda do rastreamento. O sinal da letra `T` apresenta uma configuração de mão um pouco mais complexa envolvendo dedos cruzados que também acarretou na perda de confiabilidade do Leap Motion que não conseguiu representar a configuração de mãos.

Com esses 33 sinais que podem ser reproduzidos com o Leap Motion foi montado um conjunto de amostras para os demais testes realizados, conforme mostra a Tabela 1.

Tabela 1 – Quantidade de amostras de cada sinal utilizada para treinar os algoritmos

Sinal	Quantidade de amostras
QUATRO	6
B	9
Ç	8
L	9
NOME	6
OI	6
TCHAU	7
U	5
V	6
A	7
CINCO	10
DOIS	5
E	11
HOJE	4
I	10
NOVE	6
OITO	11
ONDE	4
PODER	5
QUERER	9
REAIS	7
SALTO ALTO	8
TÊNIS	9
UM	6
Z	7
TRÊS	1
C	2
J	2
O	2
Q	2
S	2
X	2
ZERO	3
TOTAL	**197**

A partir da Tabela 1 é possível perceber que o conjunto possui um total de 197 amostras. Destas 197 amostras, 104 são amostras de sinais estáticos e outras 93 são de sinais dinâmicos.

3.4.2 Experimento 02: Funções *kernel*

O experimento de funções *kernel* consiste em testar diferentes funções *kernel* e também diferentes parâmetros de custo (`C`) para as SVMs. Os testes utilizaram as funções *kernel* linear, gaussiana e polinomial com combinações diferentes de parâmetros. A Tabela 2 apresenta os resultados obtidos para o reconhecimento de sinais estáticos com `C=1`.

Tabela 2 – Resultado das funções *kernel* para sinais estáticos

Função *kernel*	Número de classificações corretas	Percentual acerto
Linear (c = 1)	59	81,94%
Linear (c = 3)	61	84,72%
Gaussiano (σ = 1,5)	58	80,55%
Gaussiano (σ = 5)	45	62,5%
Polinomial (d = 3, c = 1)	62	86,11%
Polinomial (d = 2, c = 1)	61	84,72%

A partir da Tabela 2 é possível perceber que a função *kernel* com melhor desempenho foi a função polinomial com os parâmetros d=3 e c=1. Outro ponto que pode ser observado é que a função linear com parâmetro c=3 e a função polinomial com parâmetros d=2 e c=1 obtiveram o mesmo resultado. A Tabela 3 apresenta os resultados obtidos alterando o C da SVM para cada uma das funções com melhor desempenho.

Tabela 3 – Resultado do custo da SVM das funções *kernel* para sinais estáticos

Custo / *Kernel*	Linear (c = 3)	Gaussiano (σ = 1,5)	Polinomial (d = 3, c = 1)
C = 0,1	53 (73,61%)	16 (22,22%)	62 (86,11%)
C = 2	61 (84,72%)	61 (84,72%)	62 (86,11%)
C = 10	61 (84,72%)	61 (84,72%)	62 (86,11%)

A partir da Tabela 3 é possível perceber que a função polinomial não obteve alterações nos resultados referentes ao custo da SVM. Também é possível perceber uma queda no desempenho da função gaussiana que obteve um resultado de 22,22% com c=0,1. Outro ponto que pode ser observado é que as funções linear e gaussiana obtiveram os mesmos resultados para C=2 e C=10. A Tabela 4 apresenta os resultados obtidos para classificação dos *frames* dos sinais dinâmicos com C=1.

Tabela 4 – Resultado das funções *kernel* para *frames* de sinais dinâmicos

Função *kernel*	Número de classificações corretas	Percentual acerto
Linear (c = 1)	110	91,66%
Linear (c = 3)	110	91,66%
Gaussiano (σ = 2,5)	106	88,33%
Gaussiano (σ = 5)	99	82,5%
Polinomial (d = 1, c = 1)	110	91,66%
Polinomial (d = 2, c = 1)	107	89,16%

A partir da Tabela 4 é possível perceber que a função linear obteve o mesmo resultado independente do valor de c. Observa-se também que a função polinomial obteve os mesmos resultados de classificação utilizando os parâmetros d=1 e c=1. A Tabela 5 apresenta os resultados obtidos alterando o C da SVM para cada uma das funções com melhor desempenho.

Tabela 5 – Resultado do custo da SVM das funções *kernel* para *frames* de sinais dinâmicos

Custo / *Kernel*	Linear (`c = 3`)	Gaussiano (σ `= 1,5`)	Polinomial (`d = 3, c = 1`)
`C` = 0,1	104 (86,66%)	97 (80,83%)	104 (86,66%)
`C` = 2	109 (90,83%)	107 (89,16%)	109 (90,83%)
`C` = 10	109 (90,83%)	106 (88,33%)	109 (90,83%)

A partir da Tabela 5 é possível perceber que a alteração do `C` da SVM diminuiu o desempenho das funções linear e polinomial. Percebe-se também que o desempenho da função gaussiana obteve uma melhora com o parâmetro `c=2`.

3.4.3 Experimento 03: Validação cruzada

O experimento de validação cruzada consiste no uso de um conjunto de amostras que não foram utilizadas para treinamento. Para montar este conjunto foram utilizadas quatro amostras para cada sinal, totalizando 132 amostras para validação cruzada. Os testes envolvendo sinais estáticos utilizaram 72 amostras e os testes envolvendo sinais dinâmicos utilizaram 60 amostras.

Os testes para reconhecimento de sinais estáticos foram executados usando uma SVM com `C=1` e uma função *kernel* polinomial com os parâmetros `d=3` e `c=1`. Os resultados podem ser visualizados na Tabela 6.

Tabela 6 – Resultado da validação cruzada de sinais estáticos

Sinal	Número de classificações corretas	Percentual acerto
A	4	100%
B	4	100%
CINCO	4	100%
DOIS	4	100%
E	4	100%
I	4	100%
L	4	100%
NOVE	3	75%
QUATRO	4	100%
U	4	100%
UM	4	100%
V	2	50%
ZERO	3	75%
TRÊS	4	100%
C	4	100%
O	0	0%
Q	4	100%
S	2	50%
TOTAL	**62**	**86,11%**

A partir da Tabela 6 é possível perceber que 62 amostras foram classificadas corretamente, totalizando 86,11% de acerto. Dentre os 18 sinais estáticos, apenas o sinal da

letra O que não foi reconhecido ao menos uma vez. Um ponto que reforça o fator do sinal da letra O não ter sido reconhecido é que a letra O e o número zero tem o mesmo sinal (Figura 2 e Figura 3). O sinal da letra S também obteve um baixo resultado, este resultado é justificado pela semelhança com o sinal da letra A. Também é possível observar que o sinal do número três obteve um ótimo resultado mesmo utilizando apenas uma amostra para treinamento.

Ao observar a árvore de decisão gerada pelas classificações incorretas do sinal do número nove, foi possível observar que a amostra que não foi reconhecida, o sinal reconhecido foi o do número zero. Além do sinal reconhecido, foi possível perceber que o sinal do número zero foi escolhido como a classificação correta desde a primeira escolha. O sinal da letra V também apresentou alguns problemas na classificação e ao observar a árvore de decisão foi possível perceber que ele foi classificado incorretamente como o sinal da letra U. Ao observar a árvore de decisão para a letra O, foi possível observar que o sinal do número nove foi reconhecido incorretamente. Por fim, ao visualizar a árvore de decisão do sinal da letra S, ficou evidente que ele foi classificado corretamente até ser comparado com o sinal da letra A, que foi o sinal escolhido na comparação.

Os testes para reconhecimento de sinais dinâmicos foram executados usando um HMM com topologia Bakis, cinco estados e a convergência do algoritmo Baum-Welch foi 0,001. Os resultados podem ser visualizados na Tabela 7.

Tabela 7 – Resultado da validação cruzado de sinais dinâmicos

Sinal	Número de classificações corretas	Percentual acerto
Ç	4	100%
NOME	3	75%
OI	4	100%
PODER	4	100%
QUERER	4	100%
REAIS	4	100%
SALTO ALTO	4	100%
TCHAU	4	100%
TÊNIS	4	100%
Z	4	100%
ONDE	4	100%
HOJE	4	100%
X	1	25%
OITO	4	100%
J	0	0%
TOTAL	**52**	**86,66%**

A partir da Tabela 7 é possível perceber que 52 amostras de sinais dinâmicos foram reconhecidas corretamente, totalizando 86,66% de acerto. Dentre os 15 sinais dinâmicos,

apenas o sinal da letra J não foi reconhecido nenhuma vez. Observando o resultado da classificação, o sinal da letra J foi classificado como o sinal oi. As classificações incorretas do sinal da letra X foram classificadas como a letra Z. Estes sinais possuem uma configuração de mãos com a maioria dos dedos fechados, mas demonstram movimentos totalmente diferentes. Um ponto positivo foi a boa classificação dos sinais nome e reais. Estes sinais são semelhantes, pois possuem o mesmo movimento, mas tem uma pequena diferença na configuração de mãos. Ao observar o resultado da classificação dos sinais nome e reais foi possível observar que o erro do sinal nome foi classificá-lo como reais.

Os testes para reconhecimento de *frames* sinais dinâmicos foram executados usando uma SVM com C=1 e uma função *kernel* polinomial com d=1 e c=1. Os resultados podem ser visualizados na Tabela 8.

Tabela 8 – Resultado da validação cruzada de *frames* de sinais dinâmicos

Sinal	Número de classificações corretas do primeiro *frame*	Número de classificações corretas do último *frame*	Percentual acerto
Ç	4	4	100%
NOME	3	4	87,5%
OI	4	4	100%
PODER	4	4	100%
QUERER	4	4	100%
REAIS	3	4	87,5%
SALTO ALTO	4	4	100%
TCHAU	4	4	100%
TÊNIS	4	4	50%
Z	4	4	100%
ONDE	4	4	100%
HOJE	4	4	100%
X	4	4	100%
OITO	4	4	100%
J	0	0	0%
TOTAL	**54**	**56**	**91,66%**

Os resultados da Tabela 8 consistem em oito amostras avaliadas para cada sinal porque foram feitas quatro avaliações para o primeiro *frame* e quatro avaliações para o último *frame* de cada sinal dinâmico. Uma observação que pode ser feita em relação aos resultados obtidos na Tabela 7 é que o sinal da letra X foi reconhecido corretamente neste teste, mas o sinal da letra J continua apresentando problemas de classificação.

Ao observar a árvore de decisão para os resultados do sinal nome, foi possível perceber que o sinal classificado foi o sinal reais. No entanto, o sinal nome foi classificado corretamente até ser comparado diretamente com reais que acabou sendo escolhido como a classificação correta. O sinal de reais também teve uma classificação como o sinal nome, e

ao observar a árvore de decisão ficou evidente que a classificação sempre foi para o sinal `nome`. O sinal da letra `J` também foi classificado como `salto alto` durante as decisões, mas resultou na classificação do sinal do `número oito`. Em outro caso, o sinal da letra `J` foi classificado como o sinal da letra `I`, mas as decisões seguiram a mesma linha, classificando o sinal `salto alto` e posteriormente classificando o sinal da letra `I`.

3.4.4 Experimento 04: Desempenho

O experimento de desempenho consiste no tempo demorado para treinar os algoritmos utilizando as amostras apresentadas na Tabela 1 e o tempo para classificação das amostras de validação cruzada. As configurações de SVM e HMM são as mesmas do experimento do item 3.4.3. O experimento foi realizado em um *ultrabook* Dell Vostro 5470 com 8GB de memória RAM, 120GB disco rígido SSD e processador Core i5 4200 de 1.6 GHz. Os resultados sobre o tempo de treinamento podem ser visualizados na Tabela 9 e os resultados sobre o tempo de classificação podem ser visualizados na Tabela 10.

Tabela 9 – Tempo médio para inicializar os algoritmos

Algoritmo	Tempo médio para preparar dados	Tempo médio para treinar
SVM (estáticos)	31 ms	285 ms
HMM (dinâmicos)	103,66 ms	904 ms
SVM (*frames*)	77,66 ms	6785 ms

A partir da Tabela 9 é possível observar que a inicialização dos algoritmos é rápida, onde apenas a inicialização da SVM para reconhecer o começo e o fim de um sinal dinâmico teve um tempo maior de execução. Este ponto pode ser notado, devido ao fato de que a SVM é treinado com todos os *frames* dos sinais dinâmicos da aplicação. Outro ponto que pode ser observado é que o tempo para preparar os dados do HMM foi maior que o da SVM para sinais dinâmicos, isso porque os dados para o HMM levam em consideração a distância de cada *frame* em relação ao primeiro *frame* do sinal.

Tabela 10 – Tempo médio para classificar um sinal

Algoritmo	Tempo médio para classificar
SVM (estáticos)	0,12 ms
HMM (dinâmicos)	3,7 ms
SVM (*frames*)	0,28 ms

A partir da Tabela 10 é possível observar que o tempo classificação utilizando a SVM não aumentou da mesma forma que o tempo de treinamento. Mesmo com um número maior de amostras de treinamento, o tempo de 0,28 ms é pouco mais que o dobro da SVM para sinais estáticos (0,12 ms).

Outra conclusão é que o tempo total para classificação de um sinal dinâmico deve ser em torno de 4,26 ms, visto que devem ser reconhecidos os primeiro e último *frames* (0,28 ms para cada um) e o próprio sinal dinâmico (3,7 ms). Esses tempos de classificação garantem que a aplicação possa ser executada em tempo real, sem um tempo de espera para que os sinais sejam reconhecidos.

3.4.5 Experimento 05: Uso da aplicação

O experimento de usabilidade foi realizado com 27 usuários para avaliar a facilidade de utilização da aplicação.

3.4.5.1 Metodologia

O experimento ocorreu durante o mês de junho por meio de teste individual com os usuários. Para realizar os testes foi disponibilizado um *ultrabook* Dell Vostro 5470 e um dispositivo Leap Motion. Também foram disponibilizados exemplos de sinais que poderiam ser importados na aplicação.

Para a realização dos testes, a aplicação estava configurada para gerar os sinais aleatoriamente, mas sem repetir os sinais até que todos os sinais fossem executados ao menos uma vez. A abordagem foi escolhida para que todos os sinais fossem utilizados e também para evitar que pessoas pegassem sequências de sinais com configurações de mão mais complexas, inviabilizando os testes.

Os usuários também foram orientados para responder o formulário disponibilizado no apêndice E através da ferramenta Google Forms.

3.4.5.2 Aplicação do teste

Para iniciar a avaliação, os usuários foram orientados sobre os objetivos dos testes, dos conceitos envolvendo a aplicação e das tarefas que deveriam ser realizadas. Após a realização das tarefas os usuários foram orientados para responderem o questionário informando os resultados obtidos nos testes. As tarefas executadas durante os testes buscaram contemplar as principais funcionalidades da aplicação.

A lista de tarefas contém sete questões objetivas e duas questões descritivas para que os usuários pudessem fazer alguma observação. O questionário de usabilidade foi composto por quatro perguntas, sendo três objetivas e uma descritiva. As perguntas procuraram obter informações sobre o uso da aplicação e as impressões dos usuários. Os resultados obtidos são apresentados no item 3.4.5.3.

3.4.5.3 Análise e interpretação dos dados coletados

O Quadro 50 apresenta o perfil dos usuários envolvidos nos testes.

Quadro 50 – Perfil dos usuários

Sexo	44,4% feminino 55,6% masculino
Idade	11,1% de 15 à 20 anos 63,0% de 21 à 30 anos 22,2% de 31 à 40 anos 3,7% com mais de 40 anos
Escolaridade	0% ensino fundamental 0% ensino médio 63,0% superior incompleto 37,0% superior completo
Conhece a LIBRAS?	40,7% sim 59,3% não
Já usou algum aplicativo com o intuito de aprendizado de LIBRAS?	3,7% sim 96,3% não
Qual o seu nível de conhecimento em informática?	14,8% básico 25,9% médio 59,3% avançado

A partir do Quadro 50 é possível observar que todos os usuários já tiveram contato com o ensino superior, assim como a maioria dos usuários possui conhecimento médio ou avançado em informática. Outro ponto importante a ser notado é que não há muita diferença na quantidade de pessoas que conhecem e desconhecem a LIBRAS, trazendo informações de pessoas que já tem contato a LIBRAS e de pessoas que a desconhecem. Por fim, o ponto mais importante que pode ser levantado do perfil dos usuários é que quase todos nunca utilizaram uma aplicação com o intuito de aprendizado de LIBRAS, mesmo as pessoas que já tem contato com a LIBRAS. Essa informação abre um conjunto de possibilidades que podem ser exploradas com a finalidade do aprendizado de LIBRAS.

A análise quanto aos resultados das tarefas pode ser vista no item 3.4.5.3.1 e a análise quanto aos resultados de usabilidade pode ser vista no item 3.4.5.3.2.

3.4.5.3.1 Análise dos resultados da lista de tarefas

O Quadro 51 apresenta os resultados obtidos quanto a realização das tarefas.

Quadro 51 – Avaliação das tarefas executadas

Tarefa	Sim	Não
Conseguiu reproduzir o primeiro sinal?	85,2%	14,8%
Conseguiu reproduzir o segundo sinal?	66,7%	33,3%
Conseguiu reproduzir o terceiro sinal?	77,8%	22,2%
Conseguiu reproduzir o quarto sinal?	59,3%	40,7%
Conseguiu reproduzir o quinto sinal?	66,7%	33,3%
Conseguiu importar o primeiro sinal?	92,6%	7,4%
Conseguiu importar o segundo sinal?	70,4%	29,6%

Com base nos resultados do Quadro 51 é possível perceber que nenhuma tarefa foi executada com sucesso por todos os usuários. Mesmo assim, é possível tirar duas conclusões importantes quanto a execução das tarefas. A primeira é que uma quantidade grande de usuários conseguiu reproduzir o primeiro sinal, que era uma espécie de adaptação de uso da aplicação e do Leap Motion, demonstrando não haver muita dificuldade no uso da aplicação. A segunda é que um número grande de usuários também conseguiu fazer a importação de um sinal sem muitos problemas, demonstrando a praticidade para adicionar um novo sinal na aplicação.

Ao responder as questões sobre a execução das tarefas, os usuários também puderam fazer observações referentes as tarefas executadas. Algumas das observações dos usuários referentes à reprodução dos sinais foram: houve dificuldade para reproduzir sinais com duas mãos, alguns sinais foram reconhecidos mesmo quando o usuário não havia feito o movimento correto, o modelo da mão de exemplo é difícil de reconhecer em alguns casos e que os sinais com movimento poderiam ser reproduzidos mais lentamente. Dentre as observações, alguns usuários fizeram sugestões de melhorias como: adicionar um ponto de referência no exemplo do sinal para que o usuário saiba qual a direção do sinal, utilizar um modelo de mão mais real para poder identificar melhor os dedos e a palma da mão, mover automaticamente a câmera do modelo da mão para que o usuário não precise movê-la com o mouse e adicionar duas câmeras para visualização do exemplo do sinal. Quanto a importação de um sinal à observação que pode ser relatada é que foi fácil e intuitiva.

Enquanto os usuários realizavam as tarefas de reconhecimento de sinais, foi medido o tempo médio demorado para o usuário receber o *feedback* desde a visualização do sinal. Quando o usuário acertava o sinal na primeira tentativa, o tempo médio foi de 3,17 segundos. Quando o usuário precisava tentar mais de uma vez, o tempo médio foi de 20,7 segundos.

Durante a reprodução dos sinais também foi possível observar que a aplicação apresentou aproximadamente 30% de resultados falso-positivos. No caso, os usuários estavam com uma configuração de mãos próxima a esperada, mas ainda não era a configuração de mãos correta e a aplicação reconhecia o sinal corretamente.

3.4.5.3.2 Análise quanto a usabilidade

O Quadro 52 apresenta os resultados referentes a usabilidade da aplicação.

Quadro 52 – Avaliação de usabilidade

Você achou fácil usar a aplicação?	100% sim 0% não
Qual a sua avaliação sobre a aplicação?	37,0% ótima 51,9% boa 11,1% regular 0% ruim 0% péssima
Você acha que essa aplicação serve para o aprendizado da LIBRAS?	96,3% sim 3,7% não

A partir do Quadro 52 é possível perceber que a aplicação desenvolvida obteve bons resultados quanto à usabilidade, sendo que nenhum usuário considerou a aplicação ruim ou péssima. No entanto, um ponto muito importante que pode ser associado ao perfil dos usuários é que todos os usuários acharam a aplicação fácil de usar, mesmo os que não conheciam LIBRAS ou os que tinham conhecimento básico em informática. O resultado mais importante desta avaliação é que 96,3% dos usuários acham que a aplicação serve para o aprendizado de LIBRAS. Esse resultado indica que pessoas que já tem contato com LIBRAS, utilizariam a aplicação para treinar o conhecimento e pessoas que desconhecem LIBRAS poderiam utilizá-la para aprender conceitos básicos.

Além das questões objetivas, os usuários também tinham a opção de deixar observações quanto à usabilidade da aplicação. As principais observações dos usuários foram: aplicação intuitiva, prática de usar e promissora e também que há a necessidade de alguns aperfeiçoamentos. Além das observações, alguns usuários também fizeram sugestões de melhorias como: apresentar qual sinal está sendo reconhecido enquanto o sinal esperado não é reconhecido, deixar uma pausa quando reconhece um sinal para evitar que o próximo sinal seja reconhecido acidentalmente e também que poderia ser indicado na aplicação que o Leap Motion reconhece melhor a mão aberta inicialmente.

3.4.6 Comparação com os trabalhos correlatos

O Quadro 53 apresenta um comparativo entre a aplicação desenvolvida e os trabalhos correlatos. Os dados utilizados para comparação são baseados nos experimentos realizados.

Quadro 53 – Comparação com os trabalhos correlatos

Características / Trabalhos	Nowiki et al. (2014)	Avola et al. (2014)	Souza (2013)	Batista (2015)
Dispositivo de entrada	Leap Motion	Leap Motion	Microsoft Kinect	Leap Motion
Arquitetura	Stand-alone	Não informado	Não informado	Cliente-servidor
Reconhecimento de gestos/sinais estáticos	Utilizando SVM	Não informado	Utilizando SVM	Utilizando SVM
Reconhecimento de gestos/sinais dinâmicos	Utilizando HMM	Utilizando um algoritmo de desenho a mão livre	Utilizando HCRF	Utilizando HMM
Base de amostras	5240	Não informado	14739	197

Com base no Quadro 53 é possível perceber que o diferencial da aplicação é a arquitetura cliente-servidor, permitindo que os usuários utilizem a aplicação tendo apenas o dispositivo Leap Motion e acesso à Internet. Também é possível citar a base de amostras utilizadas para treinar os algoritmos, onde a aplicação desenvolvida utilizou uma base pequena comparada aos demais trabalhos, que pode ter comprometido os resultados de classificação.

Da mesma forma que nos trabalhos desenvolvidos por Nowiki et al. (2014) e Souza (2013), a aplicação desenvolvida conseguiu trabalhar bem com o algoritmo SVM para o reconhecimento de gestos/sinais estáticos e também com modelos probabilísticos para o reconhecimento de gestos/sinais dinâmicos.

Diferentemente do trabalho desenvolvido por Souza (2013) onde os parâmetros linguísticos de LIBRAS foram utilizados para a classificação, a aplicação desenvolvida levou em consideração apenas a configuração de mãos, o movimento e a direção. O parâmetro de expressão não manual não foi utilizado na aplicação desenvolvida porque envolve o reconhecimento da face do usuário e também uma câmera. O parâmetro de localização não foi utilizado devido ao posicionamento do Leap Motion e também porque uma melhor precisão seria adquirida através de uma câmera com visão frontal do usuário.

O uso do Leap Motion foi importante, pois evitou todo o processamento envolvendo visão computacional, permitindo o foco nos algoritmos de classificação e extração de características. Assim como o *framework* Accord.NET, desenvolvido por Souza (2013), providenciou as implementações dos algoritmos utilizados, permitindo o foco no desenvolvimento da aplicação final.

4 CONCLUSÕES

Esse trabalho apresentou o desenvolvimento de uma aplicação para ensino-aprendizagem de LIBRAS utilizando o dispositivo Leap Motion e os algoritmos de classificação SVM e HMM tendo como principais objetivos o reconhecimento de sinais estáticos e dinâmicos de LIBRAS.

A aplicação foi implementada em uma arquitetura cliente-servidor. A camada cliente foi desenvolvida utilizando as linguagens HTML, CSS e Javascript utilizando as bibliotecas LeapJS e ThreeJS para captura de dados do Leap Motion e apresentação de modelos 3D. A camada servidor foi implementada com a linguagem C# utilizando o *framework* Accord.NET para efetuar a classificação com SVMs e HMMs.

Os resultados obtidos a partir dos testes realizados com os algoritmos se mostraram satisfatórios, com aproximadamente 86% de precisão para classificação de sinais estáticos e dinâmicos mesmo utilizando uma base de amostras pequena. Além do bom desempenho para classificação, a SVM demonstrou que pode ser utilizada para classificação em tempo real demorando menos de 1 ms para classificar um sinal. O HMM apresentou resultados que também permitem a utilização em tempo real, demorando menos de 5 ms para classificar um sinal.

Os resultados obtidos a partir de testes com usuários também mostraram-se satisfatórios com um total de 96,3% de usuário que acham que a aplicação serve para o auxílio do ensino-aprendizagem de LIBRAS. Também é possível citar que 100% dos usuários acham a aplicação fácil de usar.

A principal limitação da aplicação é não utilizar todos os parâmetros de LIBRAS para classificação, destacando que não é possível classificar sinais que usem expressões faciais. Outra limitação é a localidade do sinal, porque não foi utilizado nenhum parâmetro de posição para efetuar o reconhecimento de um sinal. Também é possível citar que alguns sinais não podem ser detectados pelo Leap Motion, o que acaba limitando a quantidade de sinais que podem ser disponibilizados para aprendizado.

A partir da aplicação desenvolvida foram apresentadas técnicas e ferramentas que podem servir para o desenvolvimento de novas aplicações. Também foi desenvolvida uma API web que pode ser utilizada para classificação de sinais de LIBRAS permitindo o desenvolvimento de novas aplicações utilizando *smartphones*, *tablets* ou *webcams*. Finalmente, a aplicação desenvolvida pode ser utilizada por usuários que possuem o dispositivo Leap Motion para o aprendizado básico de LIBRAS.

4.1 EXTENSÕES

Algumas extensões possíveis para este trabalho são:

a) fazer a aplicação ficar executando em um servidor web utilizando bancos de dados;
b) utilizar técnicas de educação para melhorar o aprendizado de LIBRAS;
c) utilizar uma câmera para obter os parâmetros de expressão não manual e localização;
d) melhorar a base de amostras, as características utilizadas para classificação e treinamento e resolver os problemas com sinais estáticos semelhantes;
e) permitir o reconhecimento de frases e soletração de palavras;
f) utilizar HCRFs para reconhecer sinais com movimento.

REFERÊNCIAS

ALBRES, Neiva de Aquino. **História da Língua Brasileira de Sinais em Campo Grande - MS**. Petrópolis, 2005. Disponível em: <http://www.editora-arara-azul.com.br/pdf/artigo15.pdf>. Acesso em: 28 ago. 2014.

ARAÚJO, Ana Paula de. **Língua Brasileira de Sinais (LIBRAS)**. [S.l.], [2009]. Disponível em: <http://www.infoescola.com/portugues/lingua-brasileira-de-sinais-libras/>. Acesso em: 28 ago. 2014.

AVOLA, Danilo et al. **Markerless hand gesture interface based on Leap Motion controller**. [S.l.], [2014]. Disponível em: http://ksiresearchorg.ipage.com/seke/dms14paper/paper50.pdf. Acesso em: 18 jun. 2015.

BENTO, Nanci A. **Os parâmetros fonológicos:** configuração de mãos, ponto de articulação e movimento na aquisição da Língua Brasileira de Sinais – um estudo de caso. 2010. 143f. Dissertação (Mestrado em Letras e Linguística) – Programa de Pós-Graduação em Letras e Linguística, Universidade Federal da Bahia, Salvador.

BISHOP, Christopher. **Pattern recognition and machine learning**. New York: Springer-Verlag, 2006.

BONESSO, Diego. **Estimação dos parâmetros do kernel em um classificador SVM na classificação de imagens hiperespectrais em uma abordagem multiclasse**. 2013. 108f. Dissertação (Mestrado em Sensoriamento Remoto) – Programa de Pós-Graduação em Sensoriamento Remoto, Universidade Federal do Rio Grande do Sul, Porto Alegre.

CARVALHO, Rodrigo J. Língua de Sinais Brasileiras e breve histórico da educação surda. **Revista Virtual de Cultura Surda e Diversidade**, [S.l.], n. 7, 2011. Disponível em: <http://editora-arara-azul.com.br/novoeaa/revista/?p=466>. Acesso em: 28 ago. 2014.

COLGAN, Alex. **How does the Leap Motion controller work?**. [San Francisco], 2014. Disponível em: <http://blog.leapmotion.com/hardware-to-software-how-does-the-leap-motion-controller-work/>. Acesso em: 30 ago. 2014.

DAVIS, Alan. **Getting started with the Leap Motion SDK**. [San Francisco], 2014. Disponível em: <http://blog.leapmotion.com/getting-started-leap-motion-sdk/>. Acesso em: 30 ago. 2014.

FERREIRA, Adir L. et al. **Aprendendo Libras**: Módulo 2. Natal: EDUFRN, 2011. Disponível em: <http://sedis.ufrn.br/bibliotecadigital/site/pdf/TICS/Livro_MOD2_LIBRAS_Z_WEB.pdf>. Acesso em: 05 maio 2015.

FLACH, Peter. **Machine learning**: The art and science of algorithms that make sense of data. Cambridge: Cambridge University Press, 2012.

GESSER, Audrei. **Libras?**: que língua é essa? : crenças e preconceitos em torno da língua de sinais e da realidade surda. São Paulo: Parábola, 2009. 87 p, il.

INSTITUTO BRASILEIRO DE GEOGRAFIA E ESTATÍSTICA. **Censo demográfico 2010**: Características gerais da população, religião e pessoas com deficiência. Rio de Janeiro, 2012. Disponível em: < http://biblioteca.ibge.gov.br/visualizacao/periodicos/94/cd_2010_religiao_deficiencia.pdf>. Acesso em: 13 set. 2014.

JORDAN, Michael. **Graphical models**. [Berkeley], [2010?]. Disponível em: < http://www.cs.cmu.edu/~aarti/Class/10701/readings/graphical_model_Jordan.pdf>. Acesso em: 16 jun. 2015.

JURAFSKY, Daniel; MARTIN, James H. **Speech and language processing**: an introduction to natural language processing, computational linguistics, and speech recognition. 2. ed. New Jersey: Pearson, 2008.

KOTSIANTIS, Sotiris B. Supervised machine learning**:** a review of classification techniques. **Informatica**, [Liubliana], v. 31, n. 3, 2007. Disponível em: < http://www.informatica.si/PDF/31-3/11_Kotsiantis%20-%20Supervised%20Machine%20Learning%20-%20A%20Review%20of...pdf >. Acesso em: 12 set. 2014.

LEAP MOTION INC. **API overview**: Leap Motion C++ SDK v2.2 documentation. [San Francisco], [2014?]. Disponível em: <https://developer.leapmotion.com/documentation/cpp/devguide/Leap_Overview.html>. Acesso em: 04 abr. 2015.

LORENA, Ana C. CARVALHO, André C. P. L. F. de. Uma introdução às support vector machines. **Revista de Informática Teórica e Aplicada**, [Porto Alegre], v. 14, n. 2, 2007. Disponível em: < http://www.seer.ufrgs.br/index.php/rita/article/view/rita_v14_n2_p43-67/3543 >. Acesso em: 12 set. 2014.

MITCHELL, Tom M. (Tom Michael). **Machine learning**. New York : McGraw-Hill, 1997. 414 p, il. (McGraw-Hill series in computer science).

NOWIKI, Michal et al. **Gesture recognition library for Leap Motion controller**. 2014. 65f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Instituto de Ciência da Computação, Universidade de Tecnologia de Poznan, Poznan.

PACHECO, Jonas; ESTRUC, Ricardo. **Curso básico da LIBRAS (Língua Brasileira de Sinais)**. [S.l.], 2011. Disponível em: < http://www.surdo.org.br/Apostila.pdf >. Acesso em: 12 set. 2014.

PLATT, John C. **Sequential minimal optimization**: a fast algorithm for training support vector machines. [S.l.], 1998. Disponível em: <http://research.microsoft.com/pubs/69644/tr-98-14.pdf>. Acesso em: 18 jun. 2015.

QUADROS, Ronice Müller de; KARNOPP, Lodenir Becker. **Língua de sinais brasileira**: estudos lingüísticos. Porto Alegre : Artmed, 2004. xi, 221 p, il. (Biblioteca Artmed. Alfabetização e lingüística).

RUSSELL, Stuart J. (Stuart Jonathan); NORVIG, Peter. **Artificial intelligence**: a modern approach.2nd ed. Upper Saddle River, N.J : Prentice-Hall, 2003. xxviii,1080p, il. (Prentice-Hall series in artificial intelligence).

SANCHES, Marcelo Kaminski. **Aprendizado de máquina semi-supervisionado**: proposta de um algoritmo para rotular exemplos a partir de poucos exemplos rotulados. 2003. 120f. Dissertação (Mestrado em Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, 2003. Disponível em: <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-12102003-140536/>. Acesso em: 19 set. 2014..

SILVA, Marcelo M. **Uma abordagem evolucionária para o aprendizado semi-supervisionado em máquinas de vetores de suporte**. 2008. 106f. Dissertação (Mestrado em Engenharia Elétrica) – Programa de Pós-Graduação em Engenharia Elétrica, Universidade Federal de Minas Gerais, Belo Horizonte.

SILVA, Regiane Ferreira da. **Língua Brasileira de Sinais**. [S.l.], [2011]. Disponível em: <http://academico.fead.br/informativo/html/pdf/003_6_0.pdf>. Acesso em: 28 ago. 2014.

SMITH, Chris et al. **The history of artificial intelligence**. [Washington], 2006. Disponível em: < http://courses.cs.washington.edu/courses/csep590/06au/projects/history-ai.pdf>. Acesso em: 13 set. 2014.

SOUZA, César R. de. **Hidden Markov model-based sequence classifiers in C#**. [S.l.], 2010. Disponível em: < http://crsouza.com/2010/03/hidden-markov-model-based-sequence-classifiers-in-c/>. Acesso em: 22 jun. 2015.

SOUZA, César R. de. **Reconhecimento de gestos da Língua Brasileira de Sinais através de Máquinas de Vetores de Suporte e Campos Aleatórios Condicionais Ocultos**. 2013. 218f. Dissertação (Mestrado em Ciência da Computação) – Programa de Pós-Graduação em Ciência da Computação, Universidade Federal de São Carlos, São Carlos.

SOUZA, César R. de. **Sequence classifiers in C# - Part I:** hidden Markov models. [S.l.], 2014. Disponível em: < http://www.codeproject.com/Articles/541428/Sequence-Classifiers-in-Csharp-Part-I-Hidden-Marko>. Acesso em: 16 jun. 2015.

TAHIM, André P. N. **Máquinas de vetores de suporte (SVM)**. Florianópolis, 2010. Disponível em: < http://www.andretahim.com/publicacoes/Tutoriais/SVM/principalSvmTutorial.pdf >. Acesso em: 12 set. 2014.

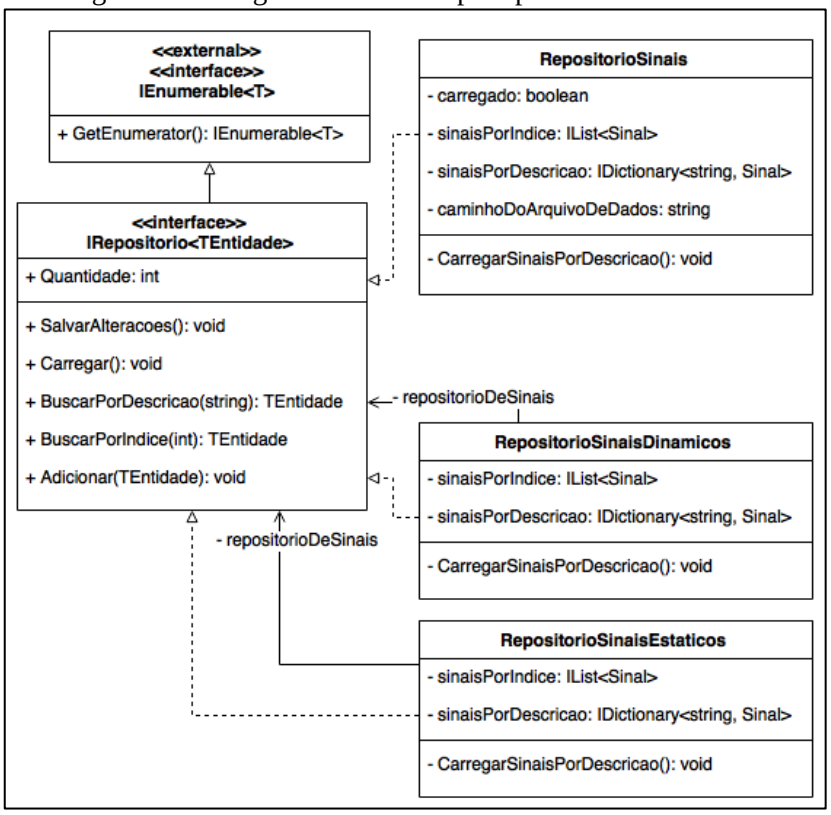
WANG, Sy Bor et al. **Hidden conditiona random fields for gesture recognition**. Massachussets, [2006?]. Disponível em: < http://people.csail.mit.edu/lmorency/Papers/wang06cvpr.pdf>. Acesso em: 18 jun. 2015.

WEICHERT, Frank et al. Analysis of the accuracy and robustness of the Leap Motion controller. **Sensors**, Basel, v. 13, n. 5, 2013. Disponível em: <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3690061/>. Acesso em: 12 set. 2014.

APÊNDICE A – Diagrama de classes completo

Neste apêndice estão os diagramas de classes complementares da aplicação. A Figura 39 demonstra o diagrama com as classes referentes à persistência de dados. A Figura 40 demonstra classes auxiliares para gerenciamento de sinais. A Figura 41 demonstra classes auxiliares da camada servidor. A Figura 42 demonstra as classes para inicialização dos algoritmos. Na Figura 43 são demonstradas as classes que permitem a comunicação com a camada servidor. Na Figura 44 são apresentadas as classes auxiliares da camada cliente. Na Figura 45 são apresentadas as classes para inicialização de câmera. Finalmente, na Figura 46 são apresentadas as classes para inicialização de cena.

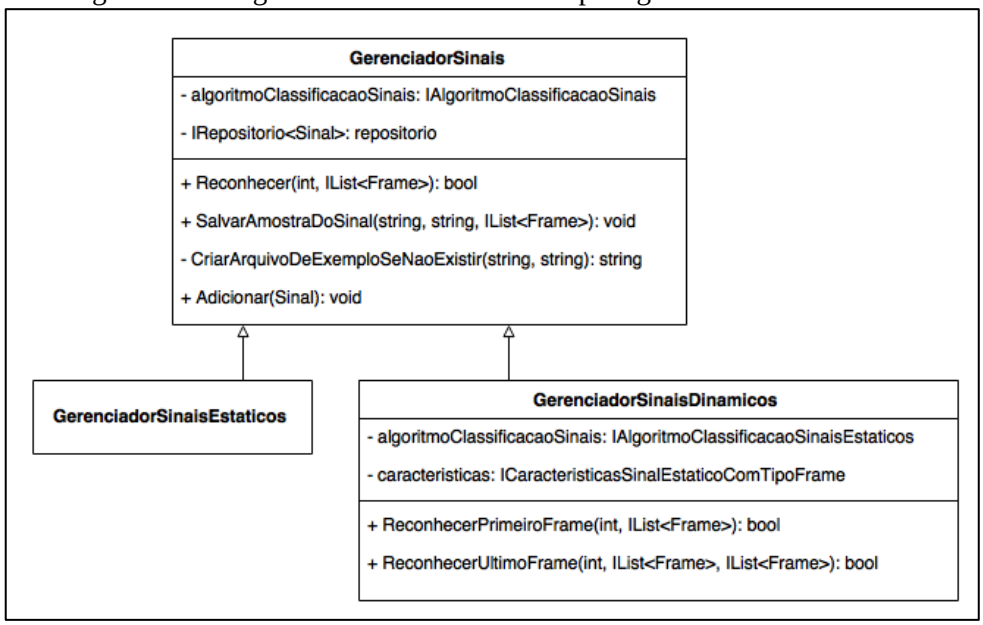
Figura 39 – Diagrama de classes para persistência de dados



A persistência dos dados parte do princípio de usar um repositório, representado pela interface `IRepositorio` (Figura 39). Um repositório é uma abstração para uma lista, dessa forma não importa se os dados são armazenados em um arquivo ou em um banco de dados. A interface `IRepositorio` também estende a interface `IEnumerable` para ser uma lista que permite a iteração sobre os seus itens. O repositório também trabalha com a ideia do padrão `Unit of Work`, porque todas as alterações são efetuadas em memória até que o método `SalvarAlteracoes` seja executado para efetuar a persistência. A classe `RepositorioSinais` fornece a implementação padrão dessa interface. As classes `RepositorioSinaisEstaticos` e

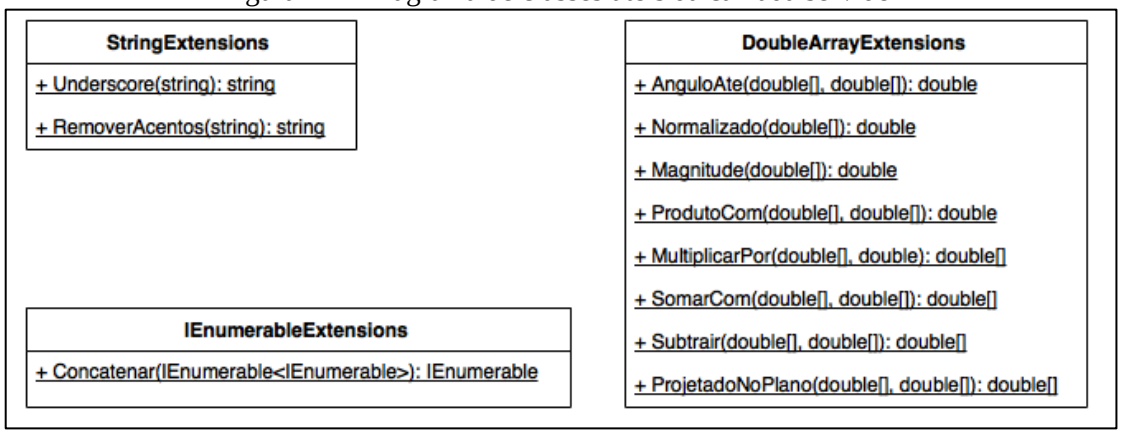
`RepositorioSinaisDinamicos` trabalham com o padrão de projetos `Decorator` porque adicionam comportamentos específicos à classe `RepositorioSinais`.

Figura 40 – Diagrama de classes auxiliares para gerenciamento de sinais



A classe `GerenciadorSinais` (Figura 40) é a classe que fornece o comportamento para reconhecer um sinal, através do método `Reconhecer` e para adicionar uma nova amostra de sinal, através do método `SalvarAmostraDoSinal`. A classe `GerenciadorSinaisEstaticos` apenas estende essa classe porque não precisa adicionar nenhum comportamento para trabalhar com sinais estáticos. A classe `GerenciadorSinaisDinaimcos` é reponsável pelo reconhecimento de sinais dinâmicos e estende o comportamento padrão da classe `GerenciadorSinais`. Essa classe adiciona o método `ReconhecerPrimeiroFrame` para reconhecer o primeiro *frame* de um sinal dinâmico e o método `ReconhecerUltimoFrame` para reconhecer o último *frame* de um sinal dinâmico.

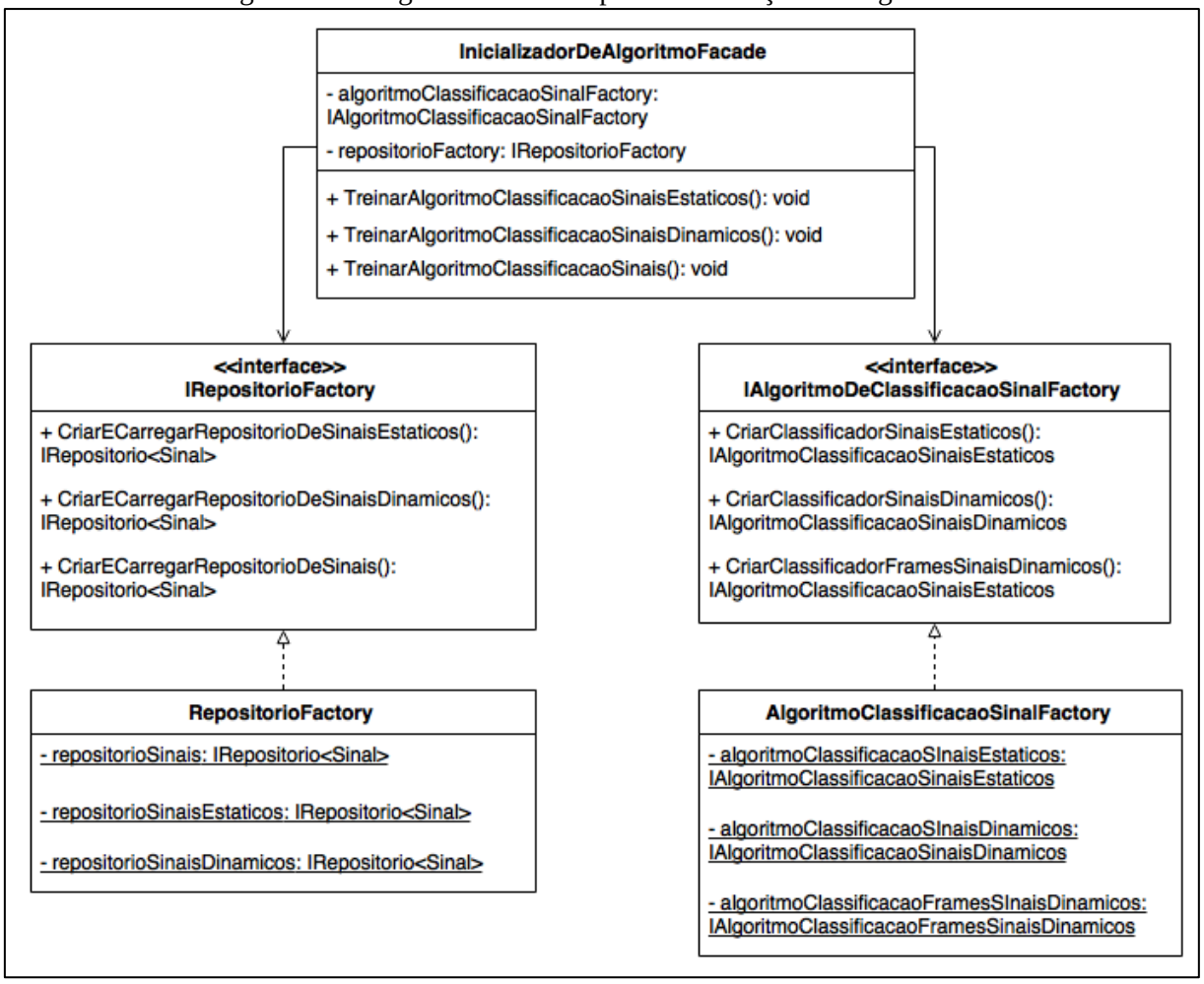
Figura 41 – Diagrama de classes úteis da camada servidor



As classes `StringExtensions`, `IEnumerableExtensions` e `DoubleArrayExtensions` (Figura 41) adicionam métodos úteis para algumas classes através do conceito de *Extension*

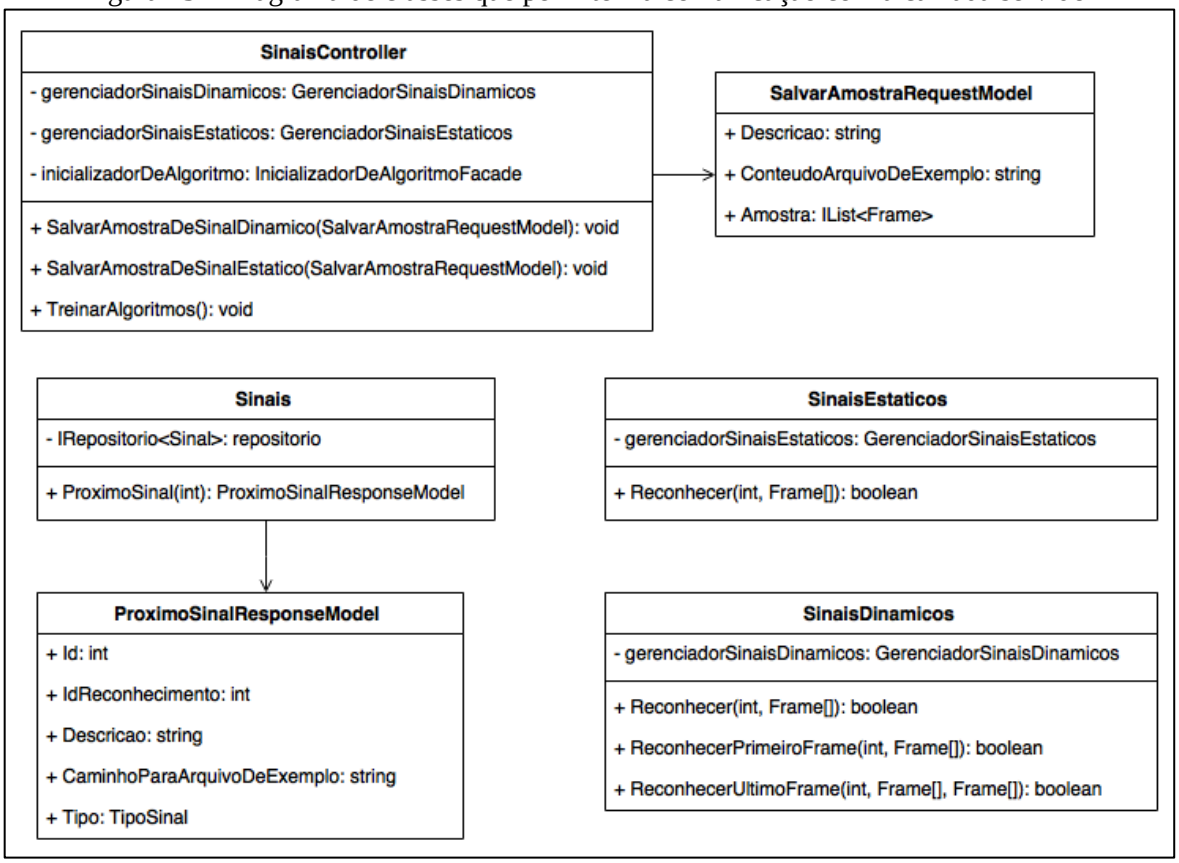
Methods da linguagem C#. O método `RemoverAcentos` da classe `StringExtensions` remove os acentos de uma `string` e o método `Underscore` troca espaços pelo caractere *underline*. O método `Concatenar` da classe `IEnumerableExtensions` concatena uma matriz em um vetor. Finalmente, os métodos da classe `DoubleArrayExtensions` são métodos auxiliares para matemática vetorial 3D, onde um vetor é representado por um *array* do tipo `double`.

Figura 42 – Diagrama de classes para inicialização dos algoritmos



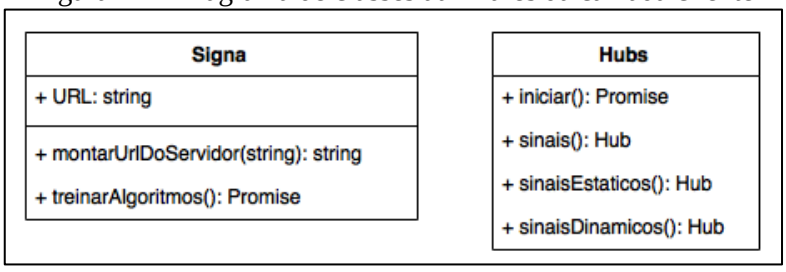
A classe `InicializadorDeAlgoritmoFacade` (Figura 42) é responsável pela inicialização dos repositórios e do treinamento dos algorimos de classificação. O método `TreinarAlgoritmoClassificacaoSinaisEstaticos` treina o algoritmo de classificação de sinais estáticos com os sinais carregados no repositório. O método `TreinarAlgoritmoClassificacaoSinaisDinamicos` treina os algoritmos de classificação de sinais dinâmicos e também dos *frames* de um sinal dinâmico. As classes `RepositorioFactory` e `AlgoritmoClassificacaoSinalFactory` são responsáveis pela criação dos devidos repositórios e algoritmos.

Figura 43 – Diagrama de classes que permitem a comunicação com a camada servidor



A classe `SinaisController` (Figura 43) permite a comunicação com o servidor através do protocolo HTTP utilizando o *framework* ASP.NET WebAPI. Essa classe é responsável por salvar amostras dos sinais e também por treinar os algoritmos de classificação. As classes `Sinais`, `SinaisEstaticos` e `SinaisDinamicos` (Figura 43) são as classes que permitem a comunicação com o servidor através de *WebSocket* utilizando o *framework* ASP.NET SignalR. A classe `Sinais` é responsável por gerar o próximo sinal que deve ser reconhecido pela aplicação. As classes `SinaisEstaticos` e `SinaisDinamicos` são responsáveis pelo reconhecimento de sinais estáticos e dinâmicos respectivamente.

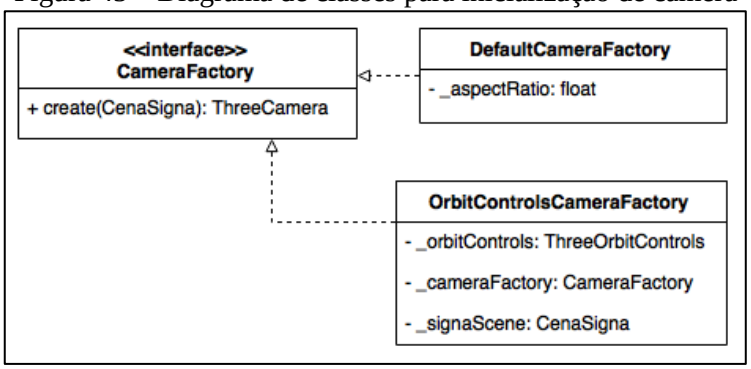
Figura 44 – Diagrama de classes auxiliares da camada cliente



A classe `Signa` (Figura 44) provê a possibilidade de iniciar o treinamento dos algoritmos de classificação pelo método `treinarAlgoritmos`. O método `montarUrlDoServidor` é um método auxiliar para gerar a URL de comunicação com o

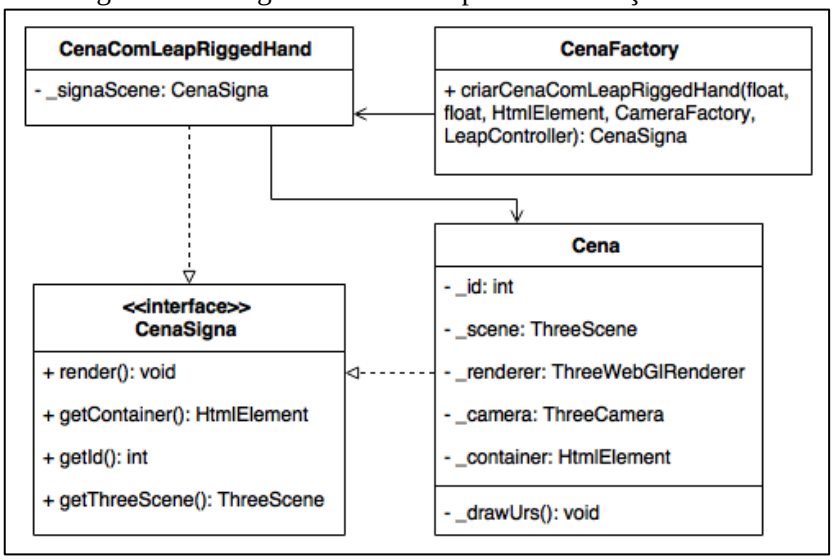
servidor. A classe `Hubs` é responsável por permitir a comunicação com a camada servidor através de *WebSocket* utilizando o *framework* ASP.NET SignalR. O método `iniciar` estabelece a comunicação com o servidor para a troca de mensagens. Os métodos `sinais`, `sinaisEstaticos` e `sinaisDinamicos` retornam interfaces que permitem a execução dos métodos das classes `Sinais`, `SinaisEstaticos` e `SinaisDinamicos` (Figura 43) na camada servidor.

Figura 45 – Diagrama de classes para inicialização de câmera



A interface `CameraFactory` (Figura 45) permite a criação de uma câmera para uma cena 3D. A classe `DefaultCameraFactory` cria uma câmera perspectiva padrão desenhando o sistema de referência do universo na cena. A classe `OrbitControlsCameraFactory` funciona com o padrão `Decorator` adicionando o comportamento de controles de órbita com o mouse para a câmera criada pela interface `CameraFactory`.

Figura 46 – Diagrama de classes para inicialização de cena

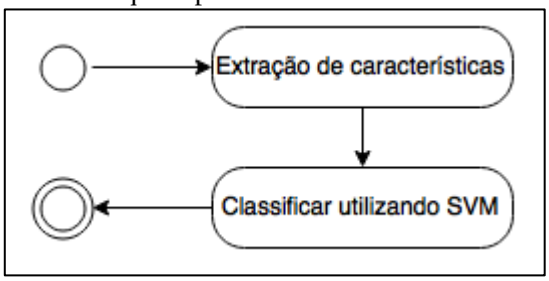


A interface `CenaSigna` (Figura 46) representa uma cena 3D, sendo que a classe `Cena` implementa a cena padrão e a classe `CenaComLeapRiggedHand` é uma cena que contém a visualização do modelo de mão 3D disponibilizado pelo *plugin* LeapJS RiggedHand. A classe `CenaFactory` é um facilitador para criar cenas da classe `CenaComLeapRiggedHand`.

APÊNDICE B – Atividades para reconhecimento de sinais estáticos e dinâmicos

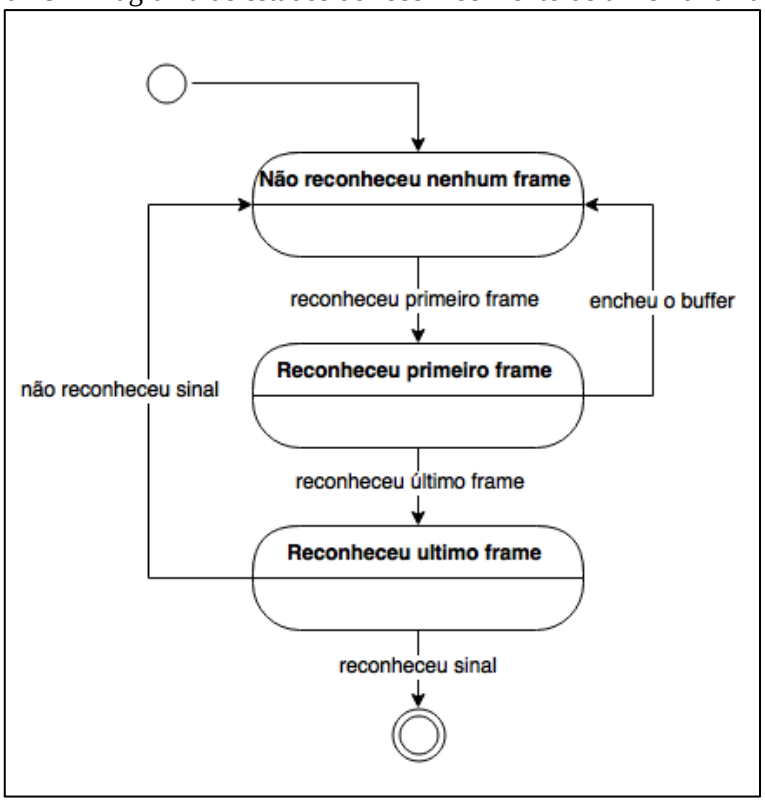
Neste apêndice são demonstrados os fluxos que fazem parte da classificação de sinais estáticos e dinâmicos. A Figura 47 apresenta o fluxo da classificação de um sinal estático, a Figura 48 e a Figura 49 apresentam o fluxo da classificação de um sinal dinâmico.

Figura 47 – Diagrama de atividades que representa o fluxo de reconhecimento de um sinal estático



O ponto de partida da Figura 47 é um *frame* do Leap Motion adaptado como uma amostra. Deste ponto em diante, são extraídas as características de um sinal estático posteriormente as características são classificadas utilizando o algoritmo SVM. O ponto final é a decisão se o sinal classificado é o sinal esperado.

Figura 48 – Diagrama de estados do reconhecimento de um sinal dinâmico

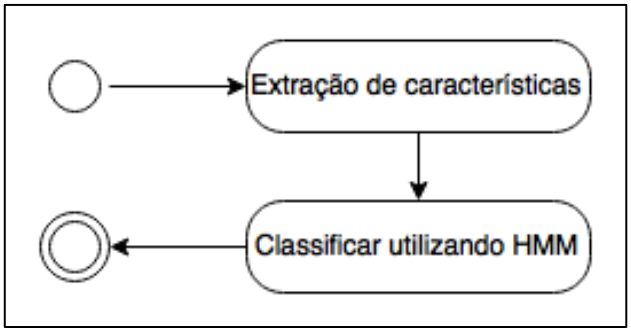


A Figura 48 demonstra os estados envolvidos no reconhecimento de um sinal dinâmico. O ponto de partida, assim como no reconhecimento de um sinal estático, é ter o *frame* do Leap Motion adaptado como uma amostra. O estado `Não reconheceu nenhum frame` é o estado inicial e indica que o primeiro *frame* de um sinal dinâmico ainda não foi

reconhecido. O estado `Reconheceu primeiro frame` indica que o primeiro *frame* foi reconhecido com sucesso, mas o último *frame* ainda não foi reconhecido. O estado `Reconheceu último frame` indica que o último *frame* também foi reconhecido corretamente. Da mesma forma que em um sinal estático, o ponto final é a decisão se o sinal classificado é o sinal esperado.

O estado `Não reconheceu nenhum frame` tenta reconhecer a amostra recebida como sendo o primeiro *frame*. Para isso, ele faz um processamento semelhante ao demonstrado na Figura 47. O estado `Reconheceu primeiro frame` tenta reconhecer a amostra recebida como sendo o último *frame* do sinal dinâmico. O processamento deste estado é semelhante ao do estado `Não reconheceu nenhum frame`. O estado `Reconheceu ultimo frame` tenta reconhecer todos os *frames* recebidos, enquanto eram executados os estados `Não reconheceu nenhum frame` e `Reconheceu primeiro frame`, como um sinal dinâmico. O processamento deste estado é demonstrado na Figura 49.

Figura 49 – Atividades no reconhecimento de um sinal dinâmico



O processamento para classificação de um sinal dinâmico é semelhante ao de um sinal estático demonstrado na Figura 47. O ponto de partida é ter todos os *frames* que devem ser classificados adaptados em uma amostra. Posteriormente são extraídas as características da amostra e então o HMM classifica essa amostra. O ponto final do processamento é a decisão que verifica se o sinal classificado é o sinal esperado.

APÊNDICE C – Formato JSON de um sinal armazenado

Neste apêndice é demonstrado o formato JSON do arquivo onde os sinais são armazenados. O Quadro 54 demonstra o formato JSON de um sinal estático que utiliza uma mão, com apenas uma amostra. A propriedade `MaoEsquerda` não foi apresentada porque ela é semelhante à propriedade `MaoDireita`, porém com os valores iguais a zero. Através do JSON apresentado é possível perceber que a estrutura do arquivo é a mesma que a da classe `Sinal`.

Quadro 54 – Formato JSON de um sinal com uma mão

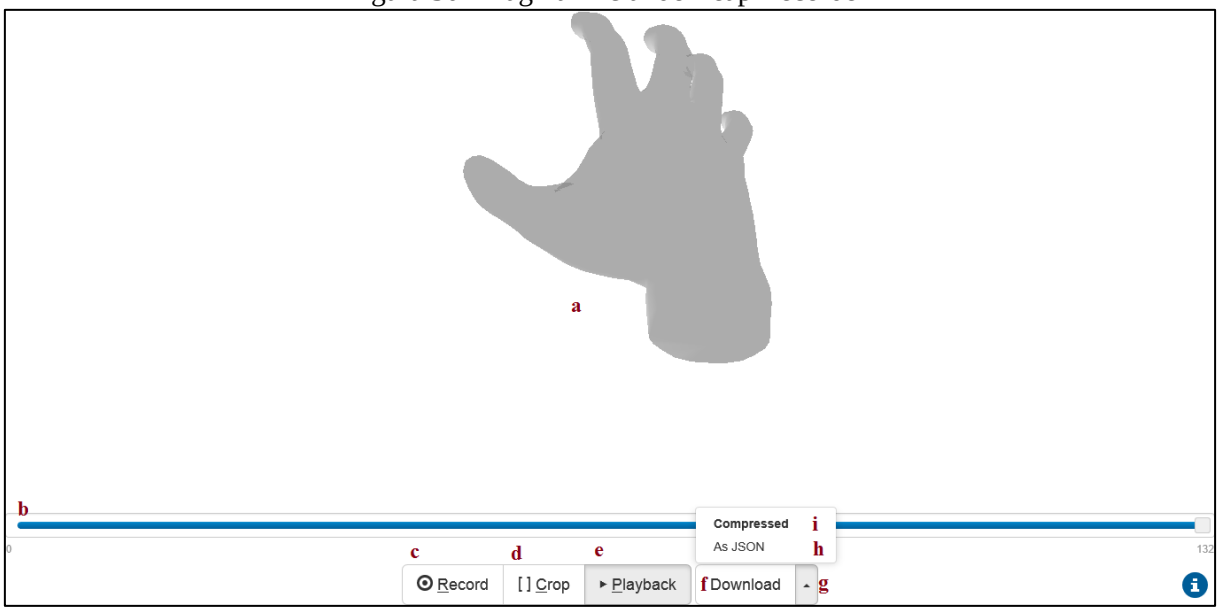
```
[{
    "Descricao":"quatro",
    "CaminhoParaArquivoDeExemplo":"exemplos/quatro.json", Tipo":0, "Id":0,
    "Amostras":[[{
            "MaoEsquerda":{ /* Mesma estrutura de "MaoDireita" */ },
            "MaoDireita":{
                "RaioDaEsfera":0.0, "Pitch":-0.057062513092749456,
                "Roll":-2.215495547052412, Yaw":-1.2148642376459453,
                "PosicaoDaPalma":[58.3643,129.997,37.2552],
                "VelocidadeDaPalma":[-50.6658,26.8428,-8.95046],
                "VetorNormalDaPalma":[-0.341564,0.256813,0.904091],
                "Direcao":[-0.937136,-0.0199019,-0.348395],
                "Dedos":[{
                    "Apontando":false, "Tipo":0,
                    "PosicaoDaPonta":[33.386,164.314,39.5813],
                    "VelocidadeDaPonta":[-60.7646,19.1581,-33.6061],
                    "Direcao":[-0.993063,-0.0126097,-0.116907]
                }, {
                    "Apontando":false, "Tipo":1,
                    "PosicaoDaPonta":[-34.681,156.476,33.4542],
                    "VelocidadeDaPonta":[-54.7837,1.2378,-39.4213],
                    "Direcao":[-0.977459,0.0529099,0.204386]
                }, {
                    "Apontando":false, "Tipo":2,
                    "PosicaoDaPonta":[-47.6066,138.338,33.783],
                    "VelocidadeDaPonta":[-46.3553,6.72023,-8.79303],
                    "Direcao":[-0.988346,0.0813691,0.128651]
                }, {
                    "Apontando":false, Tipo":3,
                    "PosicaoDaPonta":[-39.4949,114.423,33.618],
                    "VelocidadeDaPonta":[-44.8511,-0.47011,-25.588],
                    "Direcao":[-0.997474,0.0530614,0.0472242]
                }, {
                    "Apontando":false, "Tipo":4,
                    "PosicaoDaPonta":[-16.8618,88.2225,45.799],
                    "VelocidadeDaPonta":[-42.2473,11.5465,-15.4686],
                    "Direcao":[-0.995932,-0.0537258,0.0723413]
                }]
            }
        }]]
}]
```

APÊNDICE D – Tutorial de uso do Leap Recorder

Neste apêndice é descrito um rápido tutorial de como usar a ferramenta Leap Recorder para gravar sinais que serão adicionados à aplicação.

Inicialmente o Leap Motion deve estar conectado ao computador e em seguida deve-se acessar a página http://leapmotion.github.io/leapjs-playback/recorder/. A Figura 50 demonstra como é a página inicial da ferramenta.

Figura 50 – Página inicial do Leap Recorder

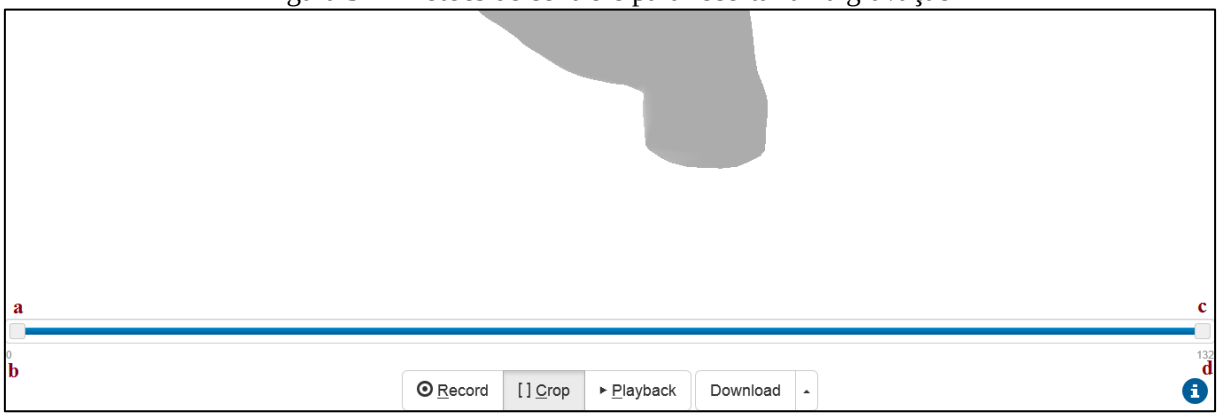


Na Figura 50 é possível perceber que existe uma cena, onde as mãos do usuário são exibidas (Figura 50a), uma barra de tempo que demonstra quantos *frames* foram gravados (Figura 50b) e sete botões de controle, sendo eles: `Record` (Figura 50c), `Crop` (Figura 50d), `Playback` (Figura 50e), `Download` (Figura 50f), mais opções de *download* (Figura 50g), `As JSON` (Figura 50h) e `Compressed` (Figura 50i).

Para iniciar a gravação de um sinal é preciso clicar no botão `Record` (Figura 50c) e reproduzir o sinal sobre o Leap Motion. Quando a reprodução for finalizada, deve-se clicar novamente no botão `Record` (Figura 50c) ou retirar as mãos de cima do Leap Motion.

Como a gravação do sinal possui algumas imperfeições e passos iniciais para que o Leap Motion possa reconhecer a configuração de mãos esperada, é possível diminuir a quantidade de *frames* gravados. Para isso, basta clicar no botão `Crop` (Figura 50d) e na Figura 51 é possível visualizar como fica a ferramenta.

Figura 51 – Botões de controle para recortar uma gravação



Se o sinal gravado for um sinal estático, então deve-se mover os botões Figura 51a e Figura 51c até que eles tenham no máximo cinco *frames* de diferença. O botão Figura 51a representa o *frame* inicial desejado e o número aparece na Figura 51b. O botão Figura 51c representa o *frame* final desejado e o número aparece na Figura 51d. Se o sinal gravado for dinâmico, o número máximo de *frames* gravados deve ser 250.

Para visualizar o recorte e se ele realmente representa o sinal desejado, é possível clicar no botão `Playback` (Figura 50e). Se a gravação do sinal estiver correta, é possível clicar no botão mais opções de *download* (Figura 50g) e então no botão `As JSON` (Figura 50h). O arquivo salvo desta gravação deve ser utilizado para importar um novo sinal na aplicação.

APÊNDICE E – Questionário

Neste apêndice constam os questionários de perfil do usuário e usabilidade, assim como o roteiro de tarefas que deveriam ser seguidas. O Quadro 55 demonstra o questionário do perfil do usuário. O Quadro 56 demonstra a lista de tarefas que os usuários deveriam reproduzir. No Quadro 57 consta o questionário de usabilidade da aplicação.

Quadro 55 – Questionário de perfil do usuário

PERFIL DO USUÁRIO

Observação: Todo o questionário é anônimo e confidencial.

Sexo:
( ) Feminino
( ) Masculino

Idade:
( ) 15 – 20 anos
( ) 21 – 30 anos
( ) 31 – 40 anos
( ) Mais de 40 anos

Escolaridade:
( ) Ensino fundamental
( ) Ensino médio
( ) Superior incompleto
( ) Superior completo

Conhece a LIBRAS?
( ) Sim
( ) Não

Já usou algum aplicativo com o intuito do aprendizado da LIBRAS?
( ) Sim
( ) Não

Qual o seu nível de conhecimento em informática?
( ) Básico
( ) Médio
( ) Avançado

Quadro 56 – Lista de tarefas

INSTRUÇÕES

Com este questionário buscamos avaliar a utilização da aplicação para auxílio no ensino-aprendizagem de LIBRAS.
Você pode utilizar a aplicação livremente para se ambientar com o Leap Motion e com a interface da aplicação.
Com base nas instruções de uso da aplicação, solicitamos que prossiga nos testes conforme as orientações abaixo.

Lista de tarefas a serem executadas:

Reproduza 5 sinais na aplicação e informe os resultados.
[ ] Conseguiu reproduzir o primeiro sinal?
[ ] Conseguiu reproduzir o segundo sinal?
[ ] Conseguiu reproduzir o terceiro sinal?
[ ] Conseguiu reproduzir o quarto sinal?
[ ] Conseguiu reproduzir o quinto sinal?

Considerações sobre a reprodução de um sinal?

_____
_____

Importe dois sinais para a aplicação.
[ ] Conseguiu importar o primeiro sinal?
[ ] Conseguiu importar o segundo sinal?

Considerações sobre a importação de um sinal?

_____
_____

Quadro 57 – Questionário de usabilidade

QUESTIONÁRIO DE USABILIDADE

Você achou fácil usar a aplicação?
( ) Sim
( ) Não

Considerações sobre a o uso da aplicação?

_____
_____

Qual a sua avaliação sobre a aplicação?
( ) Ótima
( ) Boa
( ) Regular
( ) Ruim
( ) Péssima

Você acha que essa aplicação serve para o aprendizado da LIBRAS?
( ) Sim
( ) Não