UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

LEONARDO DE SOUZA LIMA

"*Class-Test*: Classificação automática de testes para auxílio à criação de suítes de teste"

Dissertação apresentada ao Curso de Mestrado em Ciência da Computação como requisito parcial à obtenção do grau de Mestre em Ciência da Computação

ORIENTADORA: Profa. Flávia de Almeida Barros

RECIFE, ABRIL/2009

Resumo

Este trabalho apresenta o *Class-Test*, uma ferramenta idealizada para auxiliar os profissionais de testes na criação de suítes de testes extensas. Em geral, as suítes de testes devem conter um determinado número de testes de cada tipo (e.g., testes negativos, testes de fronteira, testes de interação, etc), número este fixado pelo engenheiros/designers de testes da empresa. Um dos maiores problemas enfrentados pelos testadores para montar essas suítes é o tempo gasto na classificação manual dos testes pré-selecionados para compor suítes extensas (com 1.000 testes, por exemplo).

O *Class-Test* é uma ferramenta para classificação automática de casos de testes, que visa diminuir o esforço e o tempo gasto no processo de classificação dos testes. A ferramenta foi construída com base em técnicas de Aprendizagem de Máquina, em particular, da área de Classificação de Texto. Três classificadores automáticos foram construídos utilizando-se um corpus composto por 879 casos de testes, com a distribuição de 191 casos de testes do tipo Fronteira (*Test Boundary*), 338 do tipo Negativo (*Test Negative*), e 350 do tipo interação (*Test Interaction*). Cada classificador é especializado em apenas um desses três tipos de teste. Foi necessário criar três classificadores porque alguns casos de teste podem ser associados a mais de uma classe de teste ao mesmo tempo. Foram realizados dois estudos de casos. O primeiro estudo teve como objetivo avaliar, dentre os quatro algoritmos de aprendizagem selecionados, qual apresentava melhor precisão para o corpus em questão. O algoritmo SVM – Máquina de Vetores de Suporte apresentou melhor desempenho nesse estudo. O segundo estudo de caso visou comparar a precisão da classificação automática *versus* a classificação manual.

Este trabalho é parte do projeto Test Research Project do CIn/BTC, que está sendo desenvolvido em uma parceria entre o CIn-UFPE e a Motorola. O propósito geral desse projeto é automatizar a geração, seleção e avaliação de casos de teste para aplicações de telefonia móvel.

Palavras-chave: Teste de software, Aprendizagem de Máquina, Classificação de texto.

Abstract

This document presents the *Class-Test*, a tool designed to assist professionals in creating long test suites. In general, each test suite must contain a number of test cases of each type (e.g., Negative tests, Boundary tests, Interaction tests, etc.). This number is determined by test engineers/test designers of the company. One of the main problems faced by testers to create these suites is the time spent on manual classification of the tests pre-selected to compose large suites (around 1,000 tests, for example).

The *Class-Test* is a tool for automatic classification of cases of test, which aims to reduce the effort and time spent in the manual classification of tests. The tool was built based on machine-learning techniques, particularly in the area of text classification. Three classifiers were built using a corpus composed by 879 tests cases, with 191 Boundary test cases, 338 Negative test cases, and 350 Interaction test cases. Each classifier is specialized in only one of these three types of tests. It was necessary to create three classifiers because some test cases can be linked to more than one class at the same time. Two case studies were accomplished. The first case study aimed to evaluate, among the four learning algorithms selected, which one would show better accuracy for the corpus in question (the algorithm SVM - Support Vector Machine showed better performance in this study). The second case study aimed to compare the accuracy of automatic classification versus the classification manual.

This work is part of the Test Research Project of CIn-BTC, which is being developed in partnership between CIn-UFPE and Motorola. The general purpose of this project is to automate the generation, selection and evaluation of test cases for mobile applications.

Keywords: Software Test, Machine Learning, Text Classification

Agradecimento

Em primeiro lugar agradeço a Deus.

A minha mãe, onde ela esteja, pois sempre senti seu apoio e incentivo dentro de mim, Sem isso não teria conseguido superar os desafios.

Aos meus irmãos pelo companheirismo.

Agradeço à professora Flávia Barros pela sua orientação, dedicação e paciência.

Ao Projeto de Pesquisa CIn-BTC Motorola , que através do projeto de pesquisa, ofereceu um ambiente para desenvolver projetos e experimentações. Agradeço também ao CNPq pelo incentivo financeiro.

A Erica Hori e a Carolina Fernandes por todo o apoio dado nos experimentos realizado com a classificação manual.

Ao professor Ricardo Prudêncio por todo o apoio dado na concepção dos experimentos e na utilização da ferramenta Weka.

Aos professores Paulo Borba e Augusto Sampaio pelos preciosos comentários visando melhorar a qualidade do trabalho.

Por fim, todas as pessoas do projeto de pesquisa que, direta ou indiretamente, contribuíram para o sucesso deste trabalho.

Sumário

1. Introdução 1
1.1 Trabalho Realizado 2
1.2 Organização da dissertação 4
2. Teste de Software 5
2.1 Processo de Teste de Software 5
2.1.1 Planejamento dos testes 7
2.1.2 Análise e Projeto 9
2.1.3 Codificação e Execução 9
2.2 Estágios de Teste 10
2.3 Tipos de Teste Organizacional 12
2.4 Abordagens de Teste 13
2.4.1 Teste Funcional 14
2.4.2 Teste Estrutural 16
2.4.3 Teste Híbrido 19
2.4.4 Teste Baseado em Falhas 19
2.5 Considerações Finais 22
3 Classificação de Texto 23
3.1 Algumas Aplicações 23
3.2 Indução de Classificadores 24
3.2.1 Criação da Representação dos Documentos 25
3.2.2 Seleção de Atributos 28
3.3 Algoritmos de classificação 28
3.3.1 k-Nearest Neighbor (k-NN) 28
3.3.2 Classificador Naïve Bayes 30
3.3.3 Árvores de Decisão 31
3.3.4 Máquinas de Vetores Suporte 33
3.4 Considerações Finais 35
4 *Class-Test:* Classificação automática de teste de software 36
4.1 Criação de Planos de Teste 37
4.2 Visão geral do Class-Test 40
4.3 Desenvolvimento do Class-Test 41
4.3.1 Aquisição dos Documentos 42
4.3.2 Criação da representação dos Casos de Teste 44
4.3.3 Indução dos Classificadores 47
4.4 Criando Planos de Teste com o Class-Test 51
4.5 Considerações Finais 52
5 Testes e Resultados 53
5.1 Estudo de Caso 1 53
5.1.1 Metodologia do Experimento 54
5.1.2 Resultados dos Testes 57
5.2 Estudo de Caso 2 61
5.2.1 Metodologia do Experimento e Resultados 62
5.2.2 Classificação Manual *vs* Classificação Automática 62
5.3 Considerações Finais 64

6 Conclusão 65
6.1 Contribuições 66
6.2 Trabalhos Futuros 66
7. Referências Bibliográficas 68

Lista de Figuras

Figura 2.1 6
Figura 2.2 7
Figura 2.3 11
Figura 2.4 16
Figura 2.5 17
Figura 2.6 18
Figura 2.7 19
Figura 3.1 29
Figura 3.2 29
Figura 4.1 40
Figura 4.2 41
Figura 4.3 42
Figura 4.4 43
Figura 4.5 43
Figura 4.6 45
Figura 4.7 46
Figura 4.8 50
Figura 4.9 51
Figura 5.1 57
Figura 5.2 59
Figura 5.3 60

Tabelas

Tabela 4.1 48
Tabela 5.1 54
Tabela 5.2 55
Tabela 5.3 58
Tabela 5.4 60
Tabela 5.5 61
Tabela 5.5 62
Tabela 5.6 63
Tabela 5.7 63

1. Introdução

Atualmente, empresas e indivíduos estão cada vez mais dependentes de informações textuais disponíveis em formato eletrônico, tanto na Web (e.g., *sites* de notícias, *blogs*, mensagens de correio eletrônico), como nas empresas (e.g., relatórios, documentos, casos de testes). Contudo, a grande quantidade de documentos textuais disponíveis dificulta a execução (automática ou manual) de diversas tarefas que requerem a manipulação desses documentos (e.g., recuperação de documentos relevantes, construção de bases de documentos sobre um tema particular, etc).

Nesse contexto, a classificação de documentos textuais pode ser usada como uma técnica que auxilia na seleção de documentos relevantes para auxiliar a tarefa a ser realizada. Contudo, devido à grande quantidade de documentos envolvidos nessas tarefas, a classificação manual torna-se ineficiente. Com isso, surge a necessidade de utilização do computador para a tarefa de classificação de grandes massas de documentos textuais.

A *classificação automática de texto* é uma técnica utilizada como uma forma de organizar documentos em categorias previamente determinadas, tanto para armazenagem quanto para recuperação, reduzindo o tempo de busca por informação relevante, e facilitando o acesso a essa informação [Sebastiani 2002]. Ao invés de selecionar um documento entre milhares existentes, pode-se analisar apenas os documentos pertencentes às categorias de interesse.

A construção de classificadores automáticos pode ser *manual*, seguindo a abordagem de Sistemas Baseados em Conhecimento [Russel & Norvig 2002], ou *automática*, seguindo a abordagem de Aprendizagem de Máquina [Mitchell 1997]. Este trabalho de mestrado foi desenvolvido dentro da abordagem de construção automática de classificadores. Aqui, a criação de um classificador pode ser dividida em quatro etapas principais: (1) Seleção e etiquetagem do *corpus* de documentos usados para treinamento e teste do classificador; (2) Criação da representação interna dos documentos; (3) Redução da dimensionalidade da representação inicial, a fim de escolher os termos que melhor representam cada documento; e (4) Indução do classificador, pelo uso de um algoritmo de

aprendizado, a partir do *corpus* etiquetado. A etapa final engloba testes e validação do classificador induzido.

Uma vez validado, o classificador pode ser usado na tarefa de categorização de novos documentos, desconhecidos pelo classificador. Os novos documentos devem também passar pelas etapas (2) e (3) descritas acima. A etapa (4) acima será então substituída pela classificação em si, isto é, associação de uma categoria a cada novo documento de entrada.

Na seção a seguir, apresentaremos brevemente como se deu o desenvolvimento deste trabalho de mestrado. Por fim, descreveremos a estrutura deste documento.

1.1 Trabalho Realizado

Este trabalho investigou técnicas de classificação automática de texto com o objetivo de desenvolver classificadores de casos de testes para auxiliar os profissionais na criação de planos de teste extensos.

Em geral, os planos de teste devem conter um determinado número de testes de cada tipo (e.g., testes negativos, de *boundary*, interação, etc), número este fixado pelo arquiteto/designer de testes da empresa. Um dos maiores problemas enfrentados por esses profissionais é o tempo gasto na classificação manual dos testes pré-selecionados para compor planos extensos (com 1.000 testes, por exemplo). Neste contexto, esta pesquisa teve como motivação principal diminuir o esforço e o tempo gasto no processo de seleção e classificação dos testes que irão compor os planos a serem executados pelos testadores.

A principal contribuição desta pesquisa foi a construção do *Class-test*, uma ferramenta que auxilia o time de arquitetura de testes do CIn-BTC Motorola[1] na criação dos planos de testes. Sua função principal é agrupar os casos de testes dados como entrada em categorias (classes) pré-definidas. Esse sistema dispõe de três classificadores, um para cada classe alvo: teste de Fronteira (*Bondary Test*), teste Negativo (*Negative Test),* e teste de interação (*Interaction Test*). É necessário usar três classificadores nesta

[1] CIn-BTC Motorola – Brazil Test Center - projeto de pesquisa realizado em convênio entre o CIn-UFPE e a Motorola.

tarefa porque alguns casos de teste podem ser associados a mais de uma classe de teste ao mesmo tempo.

O *Class-test* foi concebido utilizando técnicas clássicas de Aprendizagem de Máquina para Classificação de Texto. O *corpus* utilizado para treinamento e teste dos classificadores foi composto por 879 casos de testes, manualmente coletados e etiquetados. Os casos de teste utilizados foram oriundos do repositório de casos de testes do CIn-BTC Motorola, o *Test Central*.

Antes da criação da representação interna dos casos de teste (etapa (2) descrita acima), utilizamos técnicas de *stemming* e eliminação de *stopwords* para limpeza dos dados. A seguir, reduzimos a dimensionalidade da representação inicial. Nesta tarefa, o Luke (da ferramenta LUCENE) [Luke, 2008] foi utilizado para selecionar os atributos mais representativos do corpus, com base na freqüência de ocorrência de cada termo no conjunto de casos de testes.

A seguir, demos início ao processo de indução dos três classificadores de casos de testes. Dois experimentos distintos foram realizados. O primeiro estudo teve como objetivo avaliar, dentre quatro algoritmos de aprendizagem selecionados, qual apresentava melhor precisão para o *corpus* em questão. O algoritmo SVM (Máquina de Vetores de Suporte) [Cortes & Vapnik 1995] apresentou melhor desempenho nesse estudo. Já o segundo estudo de caso teve como objetivo comparar a precisão da classificação automática versus a classificação manual. Foi possível constatar que a classificação automática obteve precisão equivalente à alcançada pela classificação manual, contudo em um tempo de execução muito menor. Assim, a principal contribuição desta ferramenta se dá na redução do tempo e do esforço nas atividades de construção de plano de teste longos no CIn-BTC.

A pesquisa aqui relatada faz parte de um projeto mais abrangente de automação de testes, o CIn/BTC (*Brazil Test Center*), em desenvolvimento por uma parceria entre o CIn-UFPE e a Motorola. O propósito geral desse projeto é automatizar a geração, seleção e avaliação de casos de teste para aplicações de telefonia móvel. Assim, este trabalho teve como base os documentos de caso de testes, com objetivo de classificar e organizar todos os casos de testes em suas respectivas categorias.

1.2 Organização da dissertação

O restante do documento está estruturado em mais quatro capítulos, descritos a seguir.

Capítulo 2 - *Testes de Software*: Nesse capítulo é apresentada uma visão geral sobre testes de software, destacando sua importância dentro do processo geral de desenvolvimento de software. Veremos também abordagens e tipos de testes.

Capítulo 3 - *Classificação de Texto*: Esse capítulo apresenta uma breve introdução sobre Classificação de texto, incluindo suas etapas e sub-etapas. Por fim, veremos os algoritmos de classificação de texto utilizados neste projeto: k-NN, Naive Bayes, Árvore de decisão e o SVM.

Capítulo 4 - *Class-Test: Classificação automática de Teste de Software*: Neste capítulo são apresentados detalhes da concepção do *Class-test*. Teremos uma visão geral do processo de criação dos planos de teste, a metodologia de desenvolvimento do *Class-test,* e como se dá a criação de planos de teste usando o *Class-test*.

Capítulo 5 - *Testes e Resultados*: Neste capítulo são apresentados dois estudos de casos. O primeiro estudo teve como objetivo avaliar que algoritmo apresentava melhor precisão para o *corpus* em questão. Já o segundo estudo de caso teve como objetivo comparar a precisão da classificação automática versus a classificação manual.

Capítulo 6 - *Conclusão*: Neste capítulo final, o trabalho desenvolvido é concluído através da discussão dos resultados alcançados e da apresentação de perspectivas para trabalhos futuros.

2. Teste de Software

De acordo com [Jorgensen, 2002], afirma que a engenharia de software, com seu processo concorrente de ciclo de vida, tenta medir e melhorar a qualidade da aplicação que está sendo testada. O teste de software contribui para o aumento da confiabilidade e da qualidade do produto, conseqüentemente, reduzindo custo, tempo e esforço nas atividades de construção de um software.

De acordo com [McGregor & Sykes, 2001], o teste de software é uma atividade que consiste no esforço de encontrar defeitos introduzidos durante qualquer fase do desenvolvimento ou manutenção de sistemas e, de forma geral, esses defeitos podem ser decorrentes de omissões, inconsistências ou de má interpretação dos requisitos ou especificações por parte do desenvolvedor.

Este capítulo tem por objetivo apresentar alguns conceitos básicos sobre teste de software e suas características no ciclo de desenvolvimento de sistemas de software. A seção 2.1 apresenta conceitos básicos relacionados ao processo de teste de software, necessários para o bom entendimento deste texto. A seção 2.2 explica os vários estágios de teste encontrado na literatura. Na seção 2.3, explicamos os tipos de testes organizacionais que vamos abordar neste trabalho. A seção 2.4 mostra as abordagens de testes existentes na literatura. A seção 2.5 finaliza o capítulo mostrando uma síntese de seu conteúdo nas considerações finais.

2.1 Processo de Teste de Software

No desenvolvimento de software, é fundamental que os processos e práticas estejam bem definidos para que um produto de qualidade seja obtido [Pressman, 2002]. Com a evolução da tecnologia, a exigência por produtos de qualidade tem se tornado cada vez maior. Os profissionais necessitam diagnosticar com precisão os problemas que podem ocorrer nas várias etapas do desenvolvimento dos sistemas. Com isso, as atividades de testes durante o ciclo de vida de um software têm ganhado bastante atenção

das empresas de desenvolvimento de software, pois podem reduzir os custos e o tempo em torno de 30% a 50% do total do projeto do sistema [Peters, 2001].

Teste é importante porque contribui para verificar se uma aplicação faz tudo o que é esperado que faça. Alguns esforços de teste vão além, e visam assegurar que as aplicações não fazem nada além do especificado. De qualquer maneira, o teste fornece meios de avaliar a existência de defeitos que poderiam resultar em vários prejuízos, como perda de tempo, dinheiro, insatisfação do cliente. [McGregor & Sykes, 2001].

Inicialmente, o teste de software foi definido como uma atividade à parte dos processos de desenvolvimento, e que só era realizada ao final do desenvolvimento dos sistemas. Essa visão tradicional não se mostrou eficiente, devido aos altos custos associados a correções dos erros encontrados e manutenção do software. Isso contribuiu para a definição de métodos e técnicas sistemáticas de teste que constituíssem um processo à parte, que pudesse ser aplicado em paralelo ao longo do processo de desenvolvimento do software [McGregor & Sykes, 2001].

A figura 2.1 indica onde está inserido o plano de teste no processo de desenvolvimento de software.

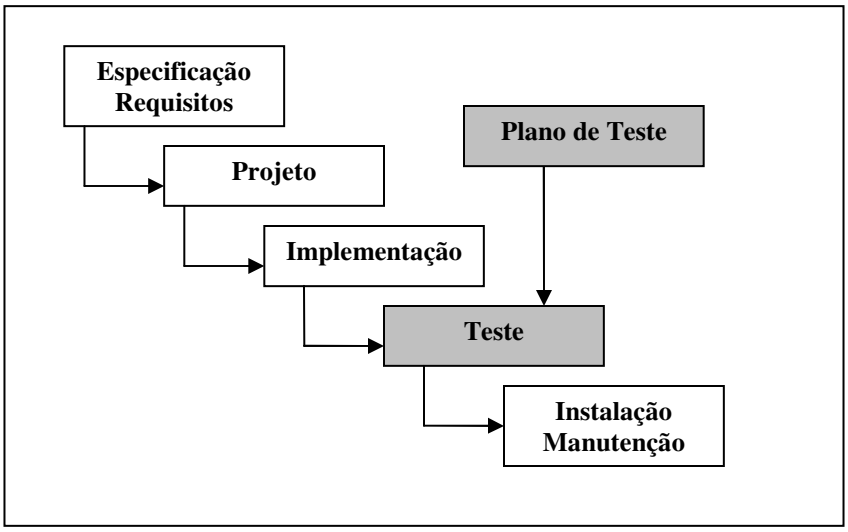

Figura 2.1 - Visão tradicional do Teste de Software

Em meados da década de 1980, as atividades de testes tornaram-se mais disseminadas e evoluíram com novas metodologias, sendo distribuídas ao longo do processo de desenvolvimento de software [Moreira, 2003]. De acordo com [Peters,

2001], o teste de software deve ser desmembrado em várias fases (ver Figura 2.2 e seção 2.2), sendo inserido no processo de desenvolvimento de software.

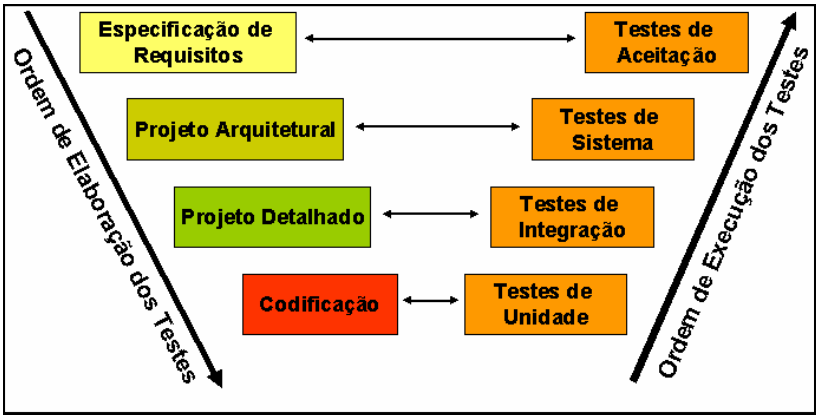

Figura 2.2 - Correspondência entre processo de desenvolvimento de software e de testes [Myers, 2004].

O teste apresenta uma idéia de melhoria de qualidade, mas é preciso se preocupar em elaborar um planejamento e seguir todas as fases dentro do ciclo de desenvolvimento do software. Para isso, define-se um *processo de teste*, que inclui atividades de planejamento, análise e projeto, codificação e execução (descritas a seguir).

As atividades de testes de software seguem o processo mais completo chamado Verificação e Validação (V&V), também chamado de (Modelo V).

- **Verificação** – é o conjunto de atividades que tenta garantir que o software implementa corretamente suas funcionalidades específicas.
- **Validação** – é o conjunto de atividades que tenta garantir que o software é construído de maneira que atenda às exigências do cliente.

2.1.1 Planejamento dos testes

No processo de teste, é essencial haver um planejamento global, que cubra todo o ciclo de desenvolvimento, com as atividades de teste e seus artefatos, tornando possível controlar o nível de qualidade do software que esta sendo produzido. [Rocha et al, 2001].

A atividade de planejamento consiste em descrever e documentar todo o processo de teste baseado nas expectativas do cliente ou dos stakeholders, ou seja, os clientes são responsáveis pela identificação do nível de qualidade que o produto se encontra. Também nesta atividade é necessário definir os recursos, alocação de pessoal, o tamanho do escopo e os riscos.

Com uma boa organização e um planejamento bem estruturado, é preciso ter um documento onde são registrados todos os passos iniciais para execução do teste. Este documento é chamado de *Plano de Teste* [IEEE, 1998]. O Plano de Teste deve ser elaborado no início do processo de planejamento da execução de teste, apresentando as *abordagens* de teste a serem utilizados, os tipos de testes a serem realizados, os *recursos* disponíveis e o *cronograma* das atividades de teste. Este documento deve especificar também quais *funcionalidades* serão abordadas, bem como as tarefas e os riscos das atividades de testes. Abaixo são enumerados os itens que compõem um plano de teste [IEEE, 1998].

- Identificação: a identificação do plano de teste, versão, e data do início e término das atividades.
- Pré-requisitos: todos os dados necessários para iniciar os testes.
- Ambiente: informações necessárias para a preparação do ambiente para iniciar a execução dos testes, incluindo a configuração dos itens necessários, como hardware e software.
- Priorização: definição da ordem de prioridade de execução dos casos de testes com base no fluxo de controle existente no sistema.
- Testes: é um conjunto de informações que descreve as entradas/saídas necessárias para que o testador possa executar os testes.

2.1.2 Análise e Projeto

Nestas atividades, são criados os vários cenários de caso de teste. Os profissionais de teste analisam a especificação do sistema para identificar os vários cenários e condições de teste, a fim de criar os casos de testes.

Para a atividade de análise, temos os seguintes artefatos [IEEE, 1998]:

- **Especificação de Projeto de Teste** – é um documento que aborda o plano de teste no modo mais refinado, ou seja, tenta levantar características e funcionalidade que vão ser testada. Este documento também tenta identificar quais procedimentos serão executados, e os critérios a serem utilizados.
- **Especificação de Caso de Teste** – é um documento que traz a descrição dos casos de teste, seu resultado esperado, tipos de dados, ações e condições necessárias para execução dos testes.

O projeto de teste tem objetivo de preocupar-se com a maneira que os requisitos serão atendidos e como os procedimentos serão executados. A partir deste momento, o projeto de teste passa para uma atividade posterior que visa à codificação e execução dos testes.

2.1.3 Codificação e Execução

Nestas atividades são abordados todos os detalhes do ambiente para dar suporte à execução da atividade de teste. Os casos de teste mostram as condições e os fluxos a serem testados por procedimentos manuais ou automáticos. Estes procedimentos contêm execução, avaliação e coleta dos resultados, para no final se ter uma análise e um maior controle do sistema em desenvolvimento.

Também temos presentes na fase de codificação e execução:

- **Veredicto** – é o estado final do teste, seus possíveis valores são: “passou” e “falhou”. Assim podemos definir que o resultado de um teste pode ter alguma falha ou se o resultado foi como esperado, passou. [UML2TP, 2003].

- **Teste de Oráculo** – é o processo de criar resultados que são esperados em um caso de teste, e depois são comparados aos resultados reais de execução dos casos de testes [British ST, 1998].

De acordo com [Walton et al., 1995], testes de software estão baseados nas idéias de confiabilidade e detecção de falhas . No entanto, podemos definir que um software é confiável quando não apresenta falhas em sua execução (ou uso).

Temos algumas outras definições para o definição de testes de software, tais como falha, erro e defeito [Jalote , 1994]. Entender esses conceitos é necessário para uma boa compreensão desta área da engenharia de software [IEEE, 1998]:

- **Erro (*Error*):** Corresponde a uma falha de uma pessoa quando está especificando o sistema, causando assim diferença no código fonte e conseqüentemente um mau funcionamento do software.
- **Falta ou Defeito (*Fault*):** Corresponde a uma condição irregular no produto, causada por erros no desenvolvimento do projeto, como por exemplo, a omissão de uma funcionalidade especificada do projeto, erro no código fonte, etc. Também podemos definir o conceito de Defeito como sinônimo de falta.
- **Falha (Failure):** Corresponde a um mau funcionamento de uma funcionalidade em realizar o que foi projetado; também pode ser detectada quando uma falta é encontrada no sistema.

Portanto, o software que não executa da forma como foi especificado pode apresentar defeitos, erros e/ou falhas. Assim, segundo [Myers, 1979], teste é o processo de executar um programa com o propósito de encontrar erros e tentar garantir um nível de qualidade dos produtos testados.

2.2 Estágios de Teste

Com aumento no nível de complexidade dos sistemas de software, as atividades de testes vêm sendo realizadas ao longo de vários estágios, ou fases de teste. O processo

de testes mais utilizado no momento é representado por cinco fases, como mostramos na Figura 2.3.

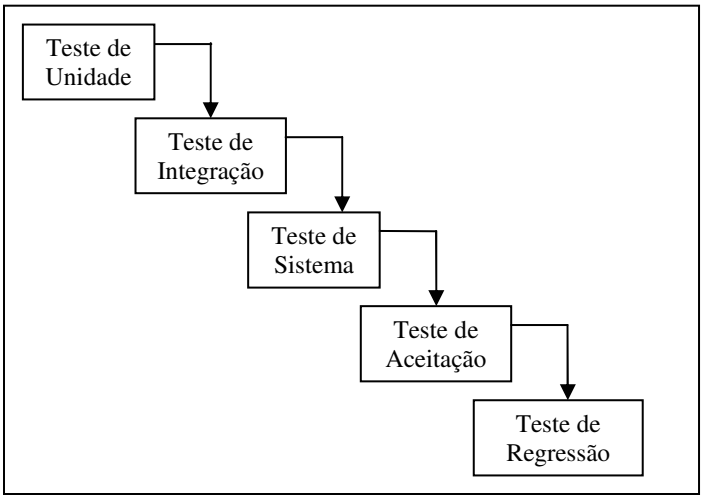

Figura 2.3 - Fases de Teste de software

- **Teste de unidade** - Consiste em testar a menor unidade do software, uma classe de métodos, funções isoladas e unidades lógicas. Cada unidade que compõe o software é testada em diferentes partes, sendo necessário que as unidades sejam pequenas (e possam ser testadas separadamente), assim possibilitando encontrar erros de especificação e implementação em cada unidade [Jorgensen, 2002]. A idéia por trás desse tipo de teste é que pode ser mais fácil corrigir o software observando-se pequenas partes do que o sistema todo de uma vez. Contudo, em geral esse tipo de teste só consegue detectar erros na lógica de programação.

- **Teste de integração** - Após a conclusão das duas fases iniciais de teste, o processo de desenvolvimento do software entra na sua fase de integração, na qual as unidades testadas são combinadas para formar unidades maiores, até que o subsistema ou sistema esteja completo. A seguir, as unidades são novamente testadas, para se verificar se ainda funcionam corretamente depois da integração. O t*este de integração* é responsável por detectar erros de interface entre as unidades sendo integradas.

- **Teste de sistema** - Essa fase tem por finalidade exercitar o sistema por completo, incluindo a execução de diferentes tipos de teste (e.g., teste funcional, teste de

recuperação, teste de desempenho, teste de estresse, teste de segurança, teste de interfaces com o usuário) [Pressman, 2006]. Esses testes devem tentar reproduzir as condições reais em que os usuários utilizarão o sistema, incluindo ambiente físico, interface gráfica e dados de entrada.

- **Teste de aceitação** – Esta fase busca avaliar a confiabilidade e o desempenho do sistema em uso operacional. Esses testes devem ser realizados pelo usuário final do sistema, a fim de verificar se os requisitos definidos por ele estão sendo atendidos. Assim, o usuário pode decidir se aceita ou não o sistema. Essa fase é geralmente desmembrada duas etapas: (1) o *teste alfa*, realizado pelo usuário na empresa que desenvolveu o sistema (a fim de permitir ao desenvolvedor registrar erros e inconsistências no uso do sistema); e (2) o *teste beta*, realizado pelo usuário nas suas instalações, sem a presença do desenvolvedor. Nesta fase, podem vir à tona erros anteriormente não constatados, uma vez que o uso de dados reais pode criar situações não pensadas nas fases anteriores [Pressman, 2000].
- **Teste de regressão** - Tem a finalidade de testar uma nova versão do software, para assegurar que adições e modificações no código já testado não apresentam erros que poderiam ocasionar a perda de confiabilidade no software [Pressman, 2000]. Testes de regressão também têm por objetivo conduzir um novo ciclo de teste durante o desenvolvimento do sistema, quando subconjuntos de testes que já foram realizados precisam ser repetidos.

2.3 Tipos de Teste Organizacional

Nesta seção são descritos alguns tipos de teste, introduzindo assim alguns conceitos básicos para o entendimento deste trabalho de pesquisa. Os tipos de teste que vão ser apresentados a seguir são: teste de fronteira (*boundary test*), teste negativo (*negative test*) e teste de interação de funcionalidades (*interaction test*). Exemplos de testes de cada um desses tipos podem ser vistos no Capítulo 4.

Teste de Fronteira é um tipo de caso de teste que define os seus valores em suas fronteiras, ou seja, as variáveis ou constantes são testadas utilizando valores Maximo ou

mínimos. Ao invés de utilizar um valor aleatório, este tipo de teste verifica seus limites superiores ou inferiores das variáveis do sistema.

Teste Negativo é um tipo de caso de teste que define uma condição ou uma informação inaceitável, inválida, anormal ou inesperada, a fim de garantir que a especificação do sistema e seus fluxos só sejam executados quando as condições estiverem corretas. O intuito do caso de teste negativo é ter um comportamento contrário ao daqueles testes que tentam verificar se a funcionalidade se comporta de acordo com o que foi especificado.

Teste de Interação é utilizado quando há uma necessidade de verificar a comunicação entre funcionalidades presente no software, ou seja, as funcionalidades são testados separadamente, depois são integradas para verificar o comportamento da junção das partes do software, enquanto nos testes de interação o objetivo é tenta verificar os eventos (ou comunicação) das funcionalidades se estão interagindo corretamente com outras funcionalidades.

2.4 Abordagens de Teste

As abordagens de teste mais conhecidas na literatura são os teste estrutural e teste funcional [Jorgensen, 2002]. Nos testes estruturais, são verificados os fluxos de caminho, de controle e as informações dos fluxos de dados, que dependem da implementação do sistema. Já a abordagem de teste funcional se baseia na especificação do sistema para detectar problemas no software.

Existem ainda outras abordagens mais recentes, que são os testes híbridos e os testes baseados em falhas. A abordagem híbrida, também chamada de caixa-cinza, originou-se da necessidade de se juntar as abordagens de teste estrutural e funcional, trazendo para o profissional de teste um melhor entendimento do comportamento da estrutura interna e da especificação do sistema, tentando assim reduzir a quantidade de testes que percorrem o mesmo caminho.

Por fim, a abordagem de teste baseado em falhas tem a finalidade de criar hipóteses sobre falhas essenciais nos programas que esta sendo testado, ou seja, bem

como criar ou avaliar conjuntos de testes baseado na sua eficácia em detectar as falhas hipotéticas. Alguns autores dizem que análise de mutantes é a forma mais comum de teste baseado em falhas [Demillo et al., 1978]. Veremos a seguir mais detalhes da cada uma dessas abordagens. Neste trabalho de mestrado, vamos trabalhar com testes dentro da abordagem funcional.

2.4.1 Teste Funcional

O teste funcional (conhecido também como "caixa preta") baseia-se na especificação do sistema para verificar se o código está funcionando corretamente [Myers, 1979]. Essa abordagem trata o software como uma caixa preta cujo conteúdo é desconhecido, e do qual só é possível visualizar o lado externo. Por isso, os dados fornecidos de entrada e as respostas esperadas dos sistemas são as únicas informações disponíveis nesta abordagem. O código do sistema é verificado sem se ter acesso aos detalhes de implementação.

Normalmente, as técnicas de teste funcional buscam derivar casos de testes a partir de especificações do funcionamento do sistema. Assim, esses casos de testes são independentes da implementação do sistema (ao contrário dos casos de teste estruturais). Deste modo, se a implementação do sistema for alterada, os mesmos casos de teste continuam sendo úteis. Outra importante vantagem é que as atividades de teste podem ser executadas em paralelo ao desenvolvimento da aplicação, contribuindo para um melhor entendimento e correção dos modelos e especificações das etapas iniciais dos processos, evitando detecção de problemas tardiamente, diminuindo assim o impacto e os custos associados às mudanças.

O teste funcional tem por base a visão de que o software pode ser visto com uma caixa-preta, consiste da obtenção de vetores de teste a partir de uma análise realizada sob a funcionalidade do programa, sem levar em conta a estrutura interna do mesmo [Myers, 1979]. Neste tipo de abordagem de teste tem como objetivo de observar se, para uma dada entrada, o software produz a saída correta.

Nesta abordagem de teste funcional é baseada nas especificações do sistema, ou seja, consiste em definir um conjunto de dados que será utilizada para verificar as

funções presente no sistema [Price, 1991], esta abordagem não aborda a verificação de como é feita a codificação do sistema.

Abaixo, vamos apresentar alguns métodos utilizados dentro dessa abordagem de teste [Pressman, 1997]:

- **Particionamento em Classes de Equivalência** (*Equivalence Partition*) - Esse método divide o domínio de entrada de um programa em classes, com base nas informações de entrada representadas na especificação do sistema, para então derivar casos de teste a partir dessas classes. Assumindo-se a hipótese de que cada classe pode ser representada por um único item, basta criar um caso de teste para cada classe identificada. Aqui, um caso de teste só é considerado eficaz se é capaz de detectar classes de erros, reduzindo assim o número total de casos de teste necessário para verificar o funcionamento do sistema.
- **Análise do Valor Limite** (*Boundary Value Analyses*) – essa análise testa valores nas fronteiras (limite superior e inferior) do domínio de entrada das classes. O método busca selecionar casos de teste que exercitem essas fronteiras, pois um grande número de erros é encontrado nestes pontos. Essa análise complementa o processo de particionamento em classes de equivalência.
- **Grafo de Causa-Efeito** – este método usa grafos para representar as condições lógicas do sistema, facilitando a visualização das possíveis condições de entrada (causa) e nas ações (saída).

Um dos maiores desafios da abordagem de teste funcional é que muitas vezes a especificação do sistema é feita de modo descritivo, e não formal. Com isso, as especificações de teste derivadas de tais requisitos são também, de certa forma, imprecisas e informais. Portanto, é muito importante descobrir quais são as entradas e saídas possíveis, e quais os melhores dados para a execução do conjunto de teste, tornando assim estes testes aplicáveis praticamente em todas as fases de teste de software [Delamaro, 1997].

2.4.2 Teste Estrutural

O teste estrutural é também chamando de "caixa branca", por ser baseado no código do sistema. Seu objetivo é testar partes do código que não foram testados pelos dados de teste [Tomazela & Maldonado, 1997].

A finalidade do teste estrutural é percorrer todos os caminhos independentes existente do software. Devemos utilizar este processo ao menos uma vez, usar valores repetitivos que fiquem dentro dos limites e nas suas extremidades, também utilizar situações de decisões lógicas com valores verdadeiros ou falsos [Pressman, 2000].

Como dito, essa abordagem busca criar testes que exercitem todos os caminhos do código. Contudo, podemos ter muitas possibilidades de caminhos, decorrentes do grande número de condições existentes em um sistema, tornando-se impossível percorrer todos os caminhos existentes no software. Neste caso, é necessário utilizar *critérios de cobertura* para escolher os testes que serão realmente aplicados ao software.

O teste estrutural complementa o teste funcional. Como visto, testes funcionais se baseiam apenas nas saídas geradas pelo sistema para as entradas fornecidas, de acordo com a especificação do sistema. Contudo, podem existir erros no código do sistema que não são detectados pelas entradas fornecidas, não gerando assim uma saída com erro.

A figura 2.4 ilustra uma estrutura de controle exercitada pelo teste estrutural.

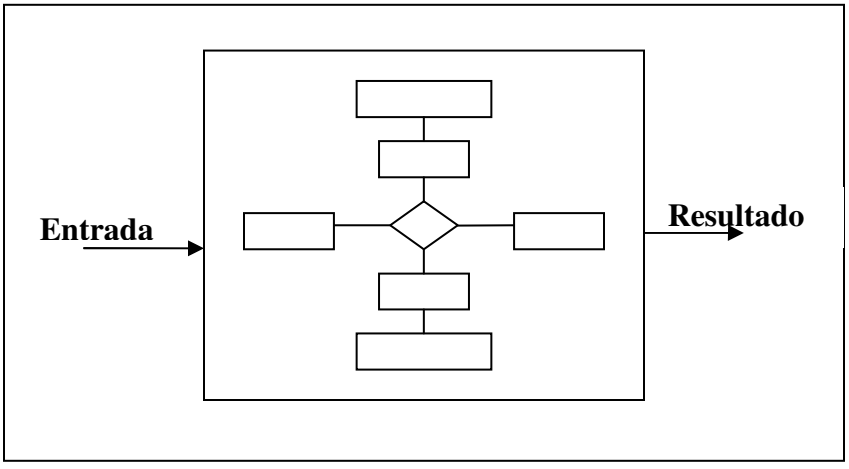

Figura 2.4 – Exemplo de estrutura para Teste Estrutural

A seguir, veremos mais detalhes sobre testes de caminho e teste de fluxo de dados, duas alternativas para se usar teste estrutural.

Teste de Caminho

O teste de caminho, também chamado de teste de fluxo de controle, observa a ordem de execução dos itens (comandos) e a relação do fluxo de controle para ajudar na análise do sistema [Peters, 2001].

A visualização do teste de caminho é possibilitada pelos seus grafos de fluxo de controle. Os elementos de fluxo de controle são utilizados para representar os caminhos de todo o sistema. A execução dos comandos é representada pelos nodos, e os caminhos pelas arestas. Na figura 2.5 são ilustradas estruturas utilizadas para representar fluxos de controle em um teste de caminho.

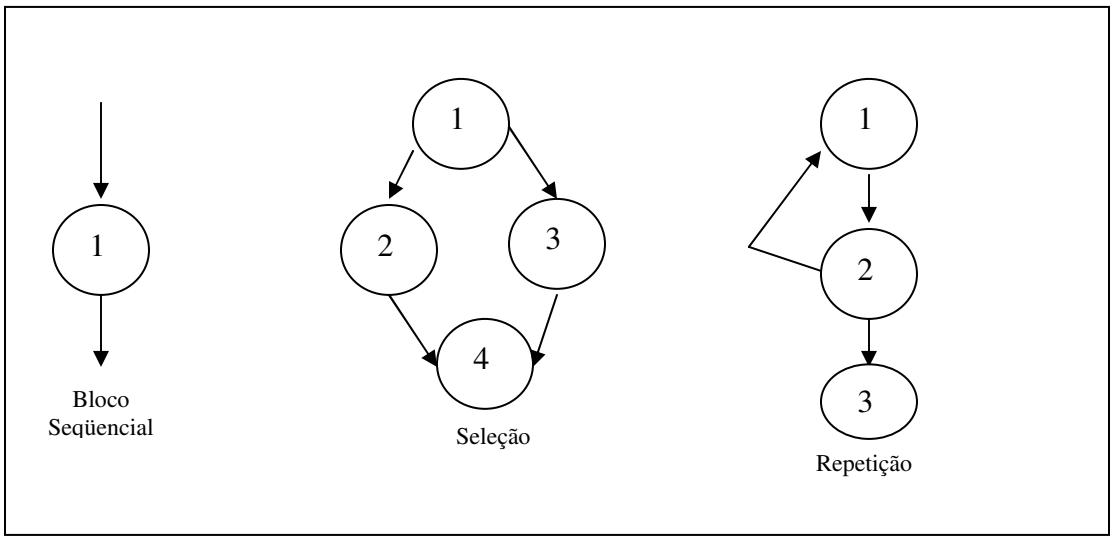

Figura 2.5 - Tipos de Fluxo de Controle

Com os testes estruturais e seus fluxos de controle, conseguimos criar vetores que são definidos a partir de grafos. Esses vetores de testes conseguem garantir a execução para cada instrução do sistema. O fluxo de controle tem as etapas importante no planejamento para construção de um bom grafo, e com ele é possível determinar os

possíveis caminhos, selecionar os caminhos a serem seguidos e verificar se os resultados de saída estão em conformidade com os dados de entrada.

Teste de Fluxo de Dados

O teste de fluxo de dados consiste em uma função de caminhos em um sistema, conforme as variáveis e as definições do sistema [Pressman, 2002]. O fluxo de dados do sistema é a base para determinar as especificações de teste, estas definições representam as interações das variáveis presentes no sistema [Rapps & Weyuker, 1985].

As informações adquiridas sobre as variáveis através dos grafos de caminho são usadas para gerar os melhores caminhos que representam as diferentes transformações de estados das variáveis. Os *nodos de definição* são representados pelos itens de comando de atribuição e de entrada, e os *nodos de uso de variáveis* podem ser vistos como comandos de saída e de condição.

1) Uma representação de *uso em predicado* pode apresentar uma condição lógica de um fluxo de dados.

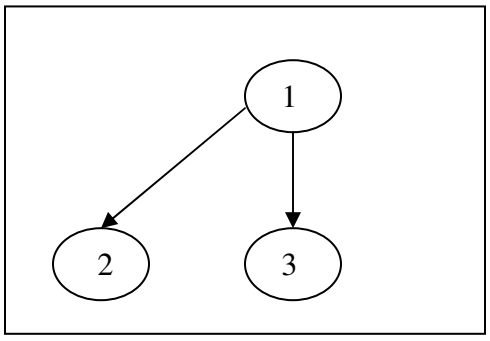

Figura 2.6 -Uso-p (uso em predicado)

2) Quando a representação é constituída de um comando não condicional, podemos afirmar que ocorre o *uso em computação*, ou Uso-c.

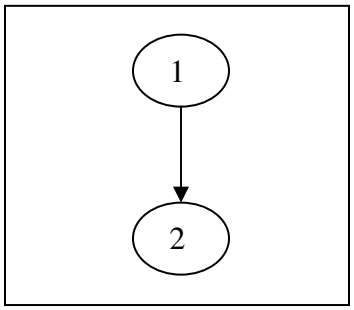

Figura 2.7 - Uso-c (Uso em Computação)

O fluxo de controle tem uma maior cobertura dos caminhos para cobrir as combinações de definição e de uso de variáveis.

2.4.3 Teste Híbrido

Como visto, a abordagem caixa preta testa o sistema a partir da sua especificação, enquanto a abordagem caixa branca verifica os caminhos lógicos internos do sistema. Considerando-se as vantagens e desvantagens de cada abordagem, surgiu a idéia de se criar a abordagem de teste híbrida, ou teste de "caixa cinza", que representa a junção das abordagens funcional e estrutural.

Nesta abordagem, o profissional de testes tenta entender as especificações do sistema, e também se comunica com a equipe de desenvolvimento para entender a arquitetura e a estrutura interna do código. O objetivo aqui é eliminar especificações ambíguas, e tentar criar testes mais claros e eficazes. Se o engenheiro de teste está a par da estrutura do código, será capaz de reduzir o conjunto de casos de testes, não correndo o risco de testar funcionalidades mais de uma vez. [Lewis, 2000].

2.4.4 Teste Baseado em Falhas

Esta abordagem toma por base informações sobre falhas comumente encontradas em qualquer software, tentando detectar a ocorrência de erros hipotéticos. Também podem ser consideradas falhas específicas que o testador está tentando descobrir naquele software em particular [Cavalcanti & Gaudel, 2007].

Uma das técnicas usadas nessa abordagem de teste é *Error Seeding* (Também conhecido como *Fault Seeding* , tem o objetivo de inserir um bug e saber se é possível encotrar o bug inserido ou não no sistema) Aqui, o testador insere erros dentro do programa sendo testado para, a partir daí, estimar o número real de falhas nesse código [Offut & Hayes, 1996]. O resultado do teste é avaliado com base no número de falhas inseridas que foram encontradas.

A técnica mais utilizada para se realizar teste baseado em falhas é a Análise de Mutantes (*Mutation Testing*) [Delamaro, 1997]. Como no caso acima, defeitos hipotéticos (falhas) também são artificialmente introduzidos no código, porém neste caso existem padrões para alterar o programa, Cada variação do programa gerada é chamada um mutante do código original, sendo vista como uma nova versão deste programa baseados no modelo de falhas adotado. o modelo de falha tem que seguir alguns critérios de adequação baseado em falhas, pois dado um programa e um conjunto de teste a analise mutante consiste em alguns passos tais como:

- Seleção dos operadores de mutação – Consiste em obter classes especificas de falhas, devendo selecionar um conjunto de operadores de mutação relevantes para estas falhas.
- Geração dos mutantes – Mutantes são gerados mecanicamente pela aplicação de operadores de mutação sobre o programa original.
- Distinguir os mutantes – Executa o programa original e cada mutante gerado com os casos de teste. Um mutante é eliminado quando pode ser diferenciado do programa original.

As falhas podem ser introduzidas automaticamente no código, através do uso de um *gerador de falhas*. Esta técnica utiliza os chamados operadores de mutação (*mutation operators*) para criar os mutantes. Na prática, as falhas geradas são pequenas alterações de sintaxe, tais como substituir uma referência para uma variável por outra, em uma expressão, ou alterar uma comparação de < para <=. Outro exemplo de alteração de mutação seria a troca de um operador AND por um operador OR em uma expressão

condicional do código. Aqui, parte-se da hipótese de que as falhas introduzidas são representativas das falhas realmente presentes no programa.

A análise dos mutantes pode ser usada para avaliar a eficácia de um conjunto de casos de teste (como um critério de adequação), para selecionar casos de teste, para aumentar o conjunto de testes, ou para estimar o número de falhas em um programa. O objetivo principal da analise mutante é selecionar casos de teste capazes de distinguir o programa que está sendo testado com os programas alternativos que são utilizados com o intuito de achar as falhas hipotéticas.

Se o programa em teste possui uma falha real, pode-se presumir que difere do programa corrigido apenas por uma pequena alteração no código. Sendo assim, é necessário apenas distinguir o programa de suas pequenas variações (selecionando-se casos de teste nos quais cada uma das variações do programa falhe) para garantir a detecção de todas estas falhas.

Algumas falhas são simples erros tipográficos, ou seja, erros de escrita no código e outras que envolvem erros lógicos mais profundos, De qualquer forma, algumas vezes um erro de lógica irá resultar em diferenças mais complexas no código do programa. Isso não invalida o teste baseado em falhas que usa um modelo de falhas mais simples, uma vez que casos de teste capazes de detectar falhas simples também são suficientes para detectar falhas mais complexas. Isso é conhecido como efeito do acoplamento.

A hipótese do efeito do acoplamento pode visto como uma alteração no código do programa. Uma alteração complexa é equivalentes a várias pequenas alterações no código de um programa. Se o efeito de uma destas pequenas alterações não é mascarado pelos efeitos das demais, então um caso de teste que diferencia uma variação ou uma alteração também pode servir para detectar erros mais complexos.

Teste baseado em falhas pode garantir a detecção de falhas apenas se a hipótese do programador competente e a hipótese do efeito do acoplamento se aplicam. Mas garantias são mais do que se espera de outras abordagens de projeto ou avaliação de conjunto de testes, incluindo os critérios de adequação de teste estrutural ou funcional discutidos anteriormente, o que é essencial é reconhecer as dependências dessas técnicas,

e de qualquer inferência sobre qualidade de software a partir de teste baseado em falhas, da qualidade do modelo de falhas utilizado.

A eficácia do teste baseado em falhas depende da qualidade do modelo de falhas utilizado, e da relação dos geradores de falhas com as falhas que realmente podem aparecer.

2.5 Considerações Finais

Neste capítulo foram apresentados alguns aspectos importantes das atividades de teste, abordando-se seus conceitos fundamentais. Suas principais atividades foram apresentadas dentro de um processo de desenvolvimento de software, para ajudar nos objetivos de qualidade de um produto de software. Também vimos os tipos de teste organizacionais que foram utilizados neste trabalho: teste de fronteira, teste negativo e interação de funcionalidades. Em seguida, foram vistas as abordagens principais de teste de software: teste estrutural (ou caixa branca), teste funcional, teste híbrido, e por último os testes baseados em falhas.

O próximo capítulo introduz conceitos sobre a classificação automática de texto, uma atividade investigada pela área de pesquisa da Inteligência Artificial (em particular, da subárea de Aprendizagem de Máquina). Serão vistas as técnicas mais utilizadas na construção de classificadores de texto, que são bastante úteis em tarefas de recuperação de informação, por exemplo.

As técnicas apresentadas no capítulo 3 foram usadas na construção de um classificador automático de tipos de testes com base sua descrição textual (detalhado no capitulo 4) com o objetivo de auxiliar na identificação dos casos de testes para construção de suítes, Esta ferramenta de classificação de teste permitirá aos profissionais de teste uma diminuição em esforço e tempos gasto na construção das suítes de testes.

3 Classificação de Texto

A classificação de textos vem sendo bastante utilizada em tarefas de organização e recuperação da informação. Podemos definir a classificação (ou categorização) de texto como uma forma de associar documentos de texto a categorias, visando auxiliar a organização de bases de documentos em estruturas de categorias bem estruturadas e conhecidas a priori.

Classificação de texto é uma técnica utilizada ainda para auxiliar a recuperação de informação, a partir dos diretórios de documentos já categorizados, possibilitando uma redução de tempo na busca da informação. Nesse uso, ao invés de recuperar documentos específicos dentre milhares de documentos presentes em uma base, podem-se analisar apenas os documentos pertencentes às categorias de interesse.

Este capítulo apresenta inicialmente algumas aplicações de classificação de texto (seção 3.1). Em seguida, na seção 3.2, apresentamos o processo geral de indução de classificadores, incluindo a aquisição de corpus, criação da representação de documentos, incluindo aspectos sobre redução de dimensionalidade. Essa etapa é uma das mais importantes durante o processo de classificação e requer um grande esforço e cuidado para se obter um conjunto de treinamento com documentos pré-processados contendo informação relevante a ser utilizada na classificação dos textos. Na seção 3.3, algoritmos de classificação são apresentados, incluindo o algoritmo k-NN, o Naive Bayes, as Árvores de Decisão e finalmente, as Máquinas de Vetores Suporte. Na seção 3.4, concluímos o capítulo com algumas considerações finais.

3.1 Algumas Aplicações

Podemos ver na literatura, várias perspectivas sobre o conceito de classificação (ou categorização) de textos, como em [Rizzi et al., 2000] que define classificação de textos como uma técnica utilizada para organizar um conjunto de documentos em uma ou mais classes (ou categorias) existentes. Documentos de texto são organizados em categorias pré-definidas, de acordo com os conteúdos que os compõem [Sebastiani,

2002]. Segundo [Moens, 2000], "o homem executa a categorização de texto lendo o texto e deduzindo as classes de expressões específicas e seus padrões de contexto. A categorização de documentos de texto simula este processo e reconhece os padrões de classificação como uma combinação de características dos textos. Estes padrões devem ser gerais o bastante para ter grande aplicabilidade, mas específicos o suficiente para serem seguros quanto à categorização de uma grande quantidade de textos".

Com a classificação de texto, podemos organizar com eficiência grandes volumes de documentos disponíveis em diversos contextos de informação. Documentos podem ser organizados em estruturas de diretórios (de forma plana ou hierárquica), e essas estruturas permitem ao usuário navegar na base de documentos, até encontrar a informação pertinente a ele. Essa tarefa pode facilitar a recuperação dos documentos, que em muitos casos é feita de forma manual por seres humanos. Como exemplos, esta técnica pode se aplicada para classificar mensagens, notícias, resumos e publicações.

Também existem outros usos para a categorização de textos, como na filtragem e disseminação de informação, que é o procedimento de encaminhar (ou recomendar) informações conforme categorias de interesses dos usuários. De modo geral, podemos dizer que o sistema de disseminação recolhe o fluxo de comunicação da informação e faz sua seleção, encaminhando para usuários específicos em seguida, conforme o tipo de necessidade de informação. Podemos citar na disseminação, a classificação de mensagens de correio eletrônico em categorias associadas a funcionamentos de serviços.

Com esta abrangência em diversos tipos de aplicação, técnicas de Classificação de Textos podem contribuir para a produção de sistemas em empresas de forma a ajudar no processo de coleta, análise e distribuição de informações e, conseqüentemente, na gestão e na estratégia competitiva da empresa.

3.2 Indução de Classificadores

A construção de classificadores utilizando técnicas de Aprendizagem de Máquina pode ser dividida em três etapas principais: aquisição do corpus de documentos, criação da representação dos documentos (incluindo a redução da dimensionalidade do vocabulário), e indução do classificador.

A etapa inicial do processo de indução consiste na aquisição do corpus para treinamento do classificador. É necessário coletar uma quantidade suficiente de documentos representativos de cada classe a ser aprendida. Idealmente, deve-se adquirir uma quantidade semelhante de documentos por classe. Por exemplo, se o algoritmo deve aprender apenas uma classe, o ideal é adquirir uma quantidade semelhante de exemplos positivos (documentos que pertencem à classe-alvo) e negativos (documentos que não pertencem à classe-alvo). Isso produz um corpus de exemplos dito balanceado.

Em seguida, é criada uma representação de cada documento através de características que são definidas comumente pelos termos presentes no documento (ver seção 3.2.1).

Finalmente, um classificador é induzido por um algoritmo de aprendizado a partir de um conjunto de exemplos de treinamento. Cada exemplo de treinamento é gerado a partir de um documento, e armazena: (1) a representação do documento; e (2) uma ou mais categorias associadas ao documento. O classificador induzido deverá ser capaz de relacionar as características que representam os documentos com as suas categorias associadas. Assim, os valores de classe para novos documentos não vistos no treinamento poderão ser previstas pelo classificador.

Veremos a seguir mais detalhes sobre a criação da representação dos documentos e da seleção de atributos.

3.2.1 Criação da Representação dos Documentos

Na abordagem de Aprendizagem de Máquina, a categorização de documentos é realizada por classificadores induzidos a partir de um conjunto de treinamento. Esse conjunto contém exemplos de documentos pertencentes às diferentes categorias, comumente representados através de um vocabulário de *termos* de indexação.

De uma forma geral, um *termo* seria qualquer seqüência de caracteres presentes entre dois espaços em branco ou sinais de pontuação em um texto. Para representação dos documentos, deve ser realizada inicialmente a identificação de todos os termos presentes na base completa de documentos. Em geral, não são considerados como termos os caracteres especiais e sinais de pontuação.

Em problemas de classificação de texto, cada atributo que descreve documentos é definido comumente como um peso associado a um dado termo de indexação. Na representação booleana, o peso do termo para um documento é igual a 1 se o termo aparece no documento e 0 caso contrário. Assim, cada documento será representado como um vetor de atributos binários, onde cada atributo indica a ocorrência ou não de um termo no documento.

Outras abordagens podem ser usadas na representação dos documentos de forma a quantificar melhor a relevância de cada termo no documento. Uma das abordagens mais conhecidas na literatura de classificação do texto é o uso do esquema TF-IDF proposto em modelos clássicos de Recuperação de Informação [Salton & Buckley, 1987], [Wiener et. al., 1995]. Nesse, esquema o peso de um termo para um dado documento é definido através da combinação de dois componentes: (1) *Term Frequency* (TF), que é a freqüência do termo no documento (quanto maior o valor de TF maior o peso do termo); e (2) *Inverse Document Frequency* (IDF), que é o inverso da freqüência de documentos da base que contêm o termo. O termo IDF é considerado de forma que um peso menor é dado a termos que aparecem na maior parte dos documentos da base, e assim são pouco úteis para representar documentos específicos. No esquema TF-IDF, cada documento será então representado por um vetor de pesos numéricos associados ao termos de indexação.

Um ponto importante na representação de documentos é que o número total de termos presentes em uma base, e que potencialmente seriam usados para representar documentos, é muito grande mesmo se considerarmos bases relativamente pequenas. Usar todos os termos da base para representar os documentos pode tornar o aprendizado lento. Além disso, muitos dos termos presentes em uma base podem ser irrelevantes para a tarefa de classificação. Assim, é necessário realizar uma etapa de redução da dimensionalidade para definição de um conjunto reduzido de termos que de fato serão relevantes para representar os documentos. A seguir, discutimos alguns procedimentos usados para redução de dimensionalidade da representação de documentos de texto.

Eliminação de Stopwords

Um dos primeiros passos aplicados no processo de redução de dimensionalidade é a eliminação de *stopwords*, que geralmente incluem artigos, preposições, conjunções, alguns verbos, dentre outras classes de palavras. Esses termos são palavras auxiliares ou conectivas (e.g., e, para, a, eles) e que não apresentam nenhuma informação discriminativa para o conteúdo dos textos. Segundo [Fox, 1992] [Korfhage, 1997] [Rijsbergen, 1979] [Salton & Macgill, 1983], as stopwords são palavras que aparecem na maioria dos documentos de uma base e não apresentam relevância para discriminação de conteúdo e, portanto inúteis para distinguir documentos pertencentes a classes diferentes.

O conjunto de palavras definidas como stopwords é conhecido como *stoplist*. Para criação de uma stoplist precisamos definir alguns itens gramáticas a serem removidos. Além disso, também podem ser consideradas palavras específicas do contexto da coleção de documentos em questão [Fox, 1992].

Stemming

O *stemming* é um método para remoção de sufixos e prefixos de cada palavra, de forma a reduzir palavras que contêm o mesmo radical para um único termo de indexação. Por exemplo, os termos "considerar", "considerado", "consideração" e "considerações" seriam reduzidos a um único termo "consider". Portanto, após a remoção dos diferentes sufixos, "-ar", "-ado", "-ação" e "ações", apenas o radical comum "consider" é usado como termo na representação dos documentos.

Existem diferentes algoritmos que realizam o procedimento de stemming. Dentre eles citamos, o algoritmo de Porter [Porter, 1980] (o mais conhecido e usado na literatura), o *Stemmer S* [Harman, 1991] e o método de Lovins [Lovins, 1968]. Esses algoritmos adotam regras explícitas para identificação e remoção de prefixos e sufixos dos termos. Em geral, as regras não levam em consideração o contexto das palavras o que pode ocasionar erros, como a unificação de palavras que contêm significados pouco relacionados. O valor do uso de stemming varia conforme a aplicação.

3.2.2 Seleção de Atributos

Os operadores de eliminação de stopwords e stemming são usados para definir um vocabulário de T termos, com T é menor que o número de total de termos contidos na base. Uma vez definido o vocabulário de termos e calculados os pesos associados aos termos, é possível aplicar técnicas de seleção de atributos que levem em consideração as classes dos exemplos de treinamento. Nesse caso, serão selecionados os termos que serão relevantes para discriminar documentos entre classes diferentes. Assim, serão selecionados M atributos mais relevantes da representação inicial (com $M < T$).

Seleção de atributos pode ser realizada usando diferentes critérios de relevância de atributos. Dentre esses critérios, podemos citar alguns comumente utilizados em classificação de texto, como Ganho de Informação, Informação Mútua, X2 statistic (Chi-quadrado), dentre outros [Yang & Pedersen, 1997].

3.3 Algoritmos de classificação

Nesta seção, vamos abordar conceitos de algoritmos de classificação comumente utilizados na literatura como o k-NN (k-Nearest Neighbors, ou k-vizinhos mais próximos), o algoritmo Naïve Bayes, as Árvores de Decisão e as Máquinas de Vetores Suporte. De uma forma geral, esses algoritmos recebem um conjunto de exemplos (ou instâncias) de treinamento e induzem (ou aprendem) um classificador. Esse classificador pode ser usado então para classificar novas instâncias não vistas durante o processo de aprendizado.

3.3.1 k-Nearest Neighbor (k-NN)

O k-NN [Aha & Kibler, 1991] é um algoritmo que classifica cada nova instância a partir dos exemplos de treinamento considerados mais similares à instância a ser classificada. Suponha que $D^n = \{x_1,..., x_n\}$ seja um conjunto de exemplos previamente rotulados com um valor de classe. Dado um exemplo x a ser classificado, são recuperados de D^n os k exemplos mais similares a x. A classe sugerida para x pode ser definida como a classe mais freqüente observada nos k exemplos mais similares. A definição da

vizinhança de cada exemplo pode feita, por exemplo, através de distância Euclidiana, medida do Co-seno, entre outras medidas. O algoritmo k-NN é bastante utilizado, principalmente por causa de sua simplicidade conceitual e implementação direta.

O procedimento de classificação com k-NN é exemplificado na figura 3.1. Nessa figura, temos um conjunto de exemplos de treinamento em um espaço bi-dimensional, onde cada exemplo pertence a uma das classes: elipse ou cruz. Essa figura mostra um exemplo hipotético a ser classificado identificado com um identificado "*" (asterisco). Usando k=1, o exemplo é classificado como sendo pertencente à classe elipse, que estava associada ao seu vizinho mais próximo. A figura 3.2, por sua vez, mostra a classificação do exemplo usando com k=3, que neste caso recebe a classe cruz. O tamanho da vizinhança (k) é um parâmetro do algoritmo k-NN, cujo valor mais adequado pode variar conforme o problema e é definido comumente de forma experimental.

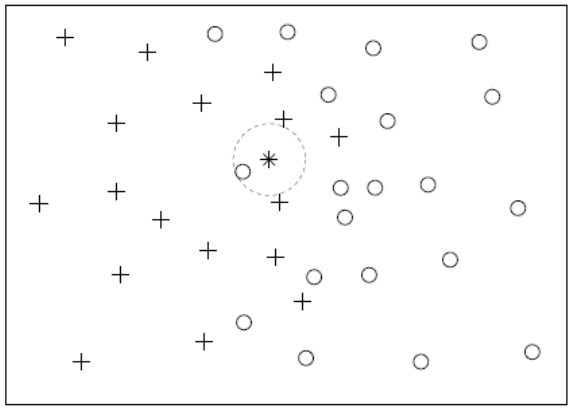

Figura 3.1 - Regra do vizinho-mais-próximo (k = 1).

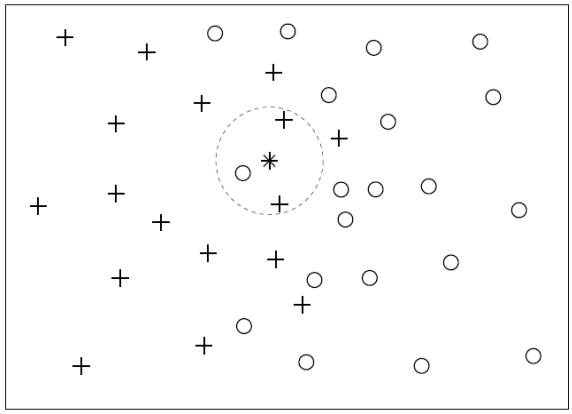

Figura 3.2 - Regra do vizinho-mais-próximo (k = 3).

Um dos principais problemas enfrentados pelo algoritmo k-NN é o alto custo computacional no cálculo das distâncias entre os exemplos de treinamento e os exemplos a serem classificados. Quando o conjunto de dados de treinamento é muito grande, o custo computacional de classificar novas instâncias pode tornar a aplicação do algoritmo pouco viável.

3.3.2 Classificador Naïve Bayes

O classificador Naïve Bayes (NB) é baseado no teorema de Bayes e é um dos classificadores mais usados em categorização de textos [McCallum & Nigam, 1998]. É um algoritmo para o aprendizado indutivo com abordagem probabilística. Baseado na probabilidade condicional de termos em relação a classes, esta técnica permite calcular as probabilidades de um novo documento pertencer a cada uma das categorias, e com isso, definir a classe com maior probabilidade para o documento [Lewis & Ringuete, 1994].

Como visto, um documento a ser classificado é representado por um conjunto de termos <T1, T2,..., Tn>. O classificador deverá atribuir a cada novo documento a sua categoria mais provável, por meio de uma função que devolve valores (categorias) pertencentes a um conjunto determinado C. No algoritmo NB, a classificação de um novo documento é definida como a classe C_{NB} que maximiza a probabilidade condicional da ocorrência do conjunto de termos presentes no documento, definida como:

$$C_{NB} = \underset{C_j \in C}{\operatorname{argmax}} P(T_1, T_2, \ldots, T_n | C_j) \cdot P(C_j) \qquad (3.1)$$

Na fórmula acima, $P(C_j)$ é estimado pela freqüência de ocorrência da classe C_j no conjunto de exemplos de treinamento. Estimar a probabilidade $P(T_1,\ldots, T_n| C_j)$ através de freqüência, no entanto, não é fácil uma vez que seria necessário observar no conjunto de treinamento cada combinação possível dos valores de $T_1,\ldots, T_n$. Para viabilizar o cálculo da probabilidade $P(T_1,\ldots, T_n| C_j)$, o algoritmo NB se baseia na suposição simplificada de que os termos usados na representação dos documentos são condicionalmente independentes. Desta forma, o algoritmo considera que a probabilidade de ocorrência de um conjunto de termos para os documentos de uma dada classe é igual ao produtório das probabilidades de ocorrência de cada atributo isoladamente:

$$P(T_1, T_2,..., T_n \mid C_j) = \prod_i P(T_i \mid C_j) \qquad (3.2)$$

Na fórmula acima, as probabilidades $P(T_i \mid C_j)$ são estimadas observando a freqüência no conjunto de treinamento dos valores de Ti para a classe Cj. Finalmente, o classificador NB é definido como:

$$C_{NB} = \underset{Cj \in C}{\operatorname{argmax}}\ P(Cj) \prod_i P(T_i \mid C_j) \qquad (3.3)$$

O algoritmo Naive Bayes recebe esse nome (*"inocente",* do termo inglês naive), por assumir que os atributos são condicionalmente independentes. No nosso, caso isso significa que o classificador assume que existe independência entre as palavras de um texto, ou seja, o método classifica documentos assumindo que a probabilidade da ocorrência de seus termos independe da posição no texto. Apesar desta suposição ser vista como não representativa da realidade, segundo [Domingo & Pazzani, 1997] a suposição de independência de termos na maioria dos casos não prejudica drasticamente a eficiência do classificador.

3.3.3 Árvores de Decisão

A árvore de decisão é uma técnica para classificar dados, podendo ser definida através de nós de teste e nós de decisão. Cada nó de teste contém um teste sobre um dos atributos do domínio. Cada resposta possível de um teste gera uma sub-árvore (com novos testes) ou um nó de decisão (uma folha). Cada folha da árvore é um nó de decisão associado a uma das classes do problema. Para classificar um novo exemplo, é realizada uma série de testes na árvore, iniciando pelo seu nó raiz até chegar a um nó folha que indicará a classe prevista para o exemplo. Árvores de decisão foram propostas inicialmente em [Hunt, 1966] e investigada por vários outros autores [Breiman, 1984], [Quinlan, 1986], [Quinlan, 1993].

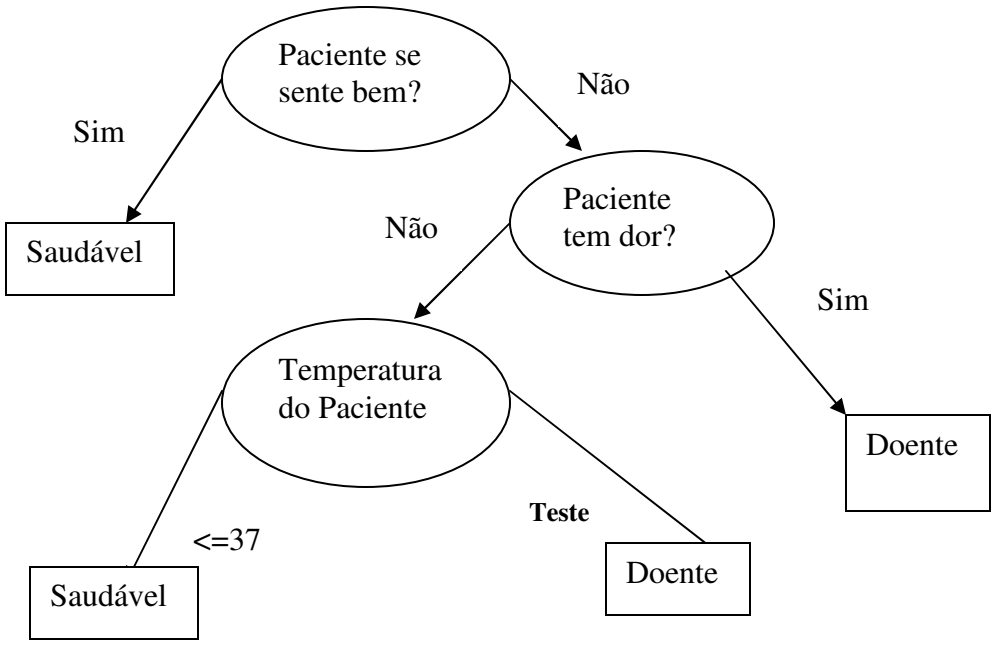

Figura 3.3 - Uma árvore de decisão simples para o diagnostico de um paciente

A figura 3.3 mostra uma representação gráfica de uma árvore de decisão para diagnostico de um paciente. Cada círculo contém um teste em um atributo para um dado conjunto de dados de paciente. Cada quadrado representa um diagnóstico, ou seja, a classe. Para classificar um paciente (definir um diagnóstico), basta iniciar os testes do paciente pelo nó raiz, seguindo cada teste até que uma folha seja alcançada.

Uma árvore de decisão pode ser transformada em um conjunto de regras, onde para cada folha é gerada uma regra que contém os testes realizados a partir raiz até a folha. Dependendo da aplicação, o uso de regras pode ser mais adequado, uma vez que cada regra pode ser analisada de forma modular sem que haja a necessidade de se referenciar outras regras [Ingargiola, 1996].

Os algoritmos usados para indução de árvores de decisão (e.g., o algoritmo C4.5 proposto em [Quinlan, 1993]), trabalham comumente de forma recursiva através dos seguintes passos: (1) a partir do nó raiz, escolher um teste de forma a maximizar a separação das classes nos exemplos de treinamento que atingem o nó, i.e., cada valor de

teste deverá ser associado a exemplos de classes diferentes; (2) gerar um novo nó para cada valor do teste escolhido; (3) se um determinado critério de parada for verificado para o novo nó (e.g., todos os exemplos que atingem o nó pertencem a uma mesma classe), então o novo nó é definido como uma folha associada a classe mais freqüente dos exemplos do nó; (4) caso contrário, retornar ao passo 1 para escolher um novo teste no nó e criar uma nova sub-árvore.

Podemos mencionar, algumas vantagens importantes para as árvores de decisão. As árvores de decisão podem ser aplicadas para qualquer tipo de dados, tanto dados numéricos como categóricos. O classificador gerado pode ser armazenado e manipulado de forma eficiente, além de gerar conhecimento em uma linguagem de alto nível que pode ser interpretado por humano de forma mais fácil comparado com outras técnicas de aprendizado.

3.3.4 Máquinas de Vetores Suporte

O algoritmo de SVM (Support Vector Machines) [Burges, 1998] é uma técnica derivada da teoria de aprendizado estatístico [Cristianini & Shawe-Taylor, 2000] [Vapnik, 1999] [Vapnik, 1998] [Cortes & Vapnik, 1995] [Scholkopf & Smola, 2002] [Cherkassky & Mulier, 1998] e se baseia na idéia de encontrar um hiperplano ótimo que separe dois conjuntos de exemplos linearmente separáveis no espaço de atributos. Ao se encontrar um hiperplano que separe os conjuntos de exemplos de classes diferentes, a classificação de um novo exemplo torna-se trivial, pois basta verificar em que região (a esquerda ou à direita do hiperplano) se encontra o novo exemplo.

A figura 3.6 mostra um exemplo de pontos representados em um espaço bidimensional associados a duas classes distintas. Na figura, vemos um hiperplano de separação possível para os exemplos definido pela equação $w^t x + b = 0$. No entanto, podem existir infinitos hiperplanos que separam dois conjuntos de pontos linearmente separáveis no espaço. Nas SVMs, é escolhido como hiperplano ótimo aquele com maior margem de separação para os exemplos mais próximos de classes diferentes (os vetores suporte) [Cherkassky & Mulier, 1998]. A margem de separação é definida como a distância entre os hiperplanos inferior e superior que passam pelos vetores suporte de

cada classe, e é definida como 2/||w||. Assim, a tarefa básica de aprendizado nas SVMs é encontrar parâmetros w e b maximizem a margem de separação (2/||w||).

No caso de pontos não linearmente separáveis, as SVMs realizam um mapeamento dos exemplos em um espaço de características maior, onde é definido um hiperplano de divisão como solução do problema de classificação. Esse mapeamento é feito usando uma função de Kernel. A idéia é tentar mapear os exemplos originais em um novo espaço onde eles sejam linearmente separáveis, e em seguida encontrar um hiperplano ótimo de separação. Diferentes funções de Kernel têm sido usadas em SVMs, como funções polinomiais (lineares e quadráticas), e funções RBF ou gaussianas.

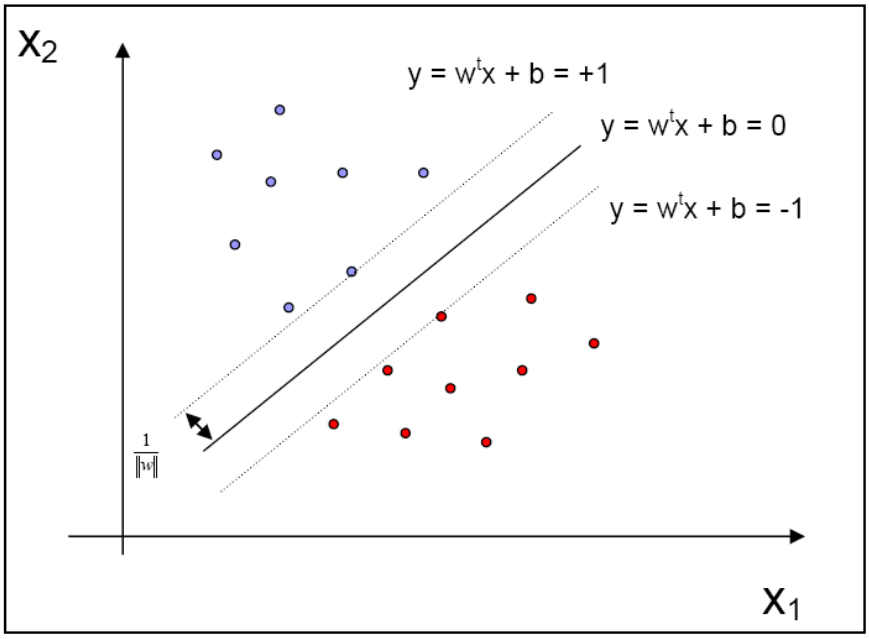

Figura 3.4 - Hiperplanos de Separação, superior e inferior formados no espaço m = 2.

As SVMs têm se mostrado bem sucedidas quando comparadas com outras técnicas de aprendizado, alcançando precisão superior em muitos problemas. Em classificação de texto, SVMs têm também apresentado bom desempenho, sendo um dos algoritmos com maior precisão obtida empiricamente em diferentes trabalhos da literatura [Sebastiani, 2002].

3.4 Considerações Finais

Neste capítulo, foram apresentados diversos aspectos básicos do tema de Classificação de Texto. Apresentamos, técnicas básicas de processamento de textos cujo objetivo era gerar representações dos documentos com informação relevante para a classificação. Em seguida, foram apresentados alguns algoritmos de Aprendizado de Máquina usados comumente para classificação de texto, incluindo o k-NN, o Naive Bayes, as Árvores de Decisão e as SVMs.

No próximo capitulo, veremos como técnicas de classificação de texto poderão ser aplicadas para classificar casos de teste em um contexto específico, de forma a dar suporte à tarefa de Teste de Software.

4 *Class-Test:* Classificação automática de teste de software

De acordo com o que foi discutido em capítulos anteriores, este trabalho de mestrado investigou o uso da *classificação automática de texto* como uma forma de contribuir na diminuição do tempo e do esforço no *processo de teste de software*. Como visto, essa técnica tem sido utilizada com sucesso para organizar documentos em categorias previamente determinadas, tanto para armazenagem quanto para recuperação, reduzindo o tempo de busca por documentos relevantes.

O Capítulo 2 apresentou uma revisão de conceitos e práticas na área de Teste de Software como parte do processo de desenvolvimento de software, com suas fases, tipos e abordagens. Já o Capítulo 3 teve como foco a Classificação de Texto, discutindo conceitos básicos da área, e as etapas na construção automática de classificadores. Por fim, foram apresentados alguns dos algoritmos para indução de classificadores (k-NN, Naive Bayes, Árvore de Decisão e o SVM). Com base nesses estudos, a classificação de texto se mostrou bastante promissora para o objetivo do nosso projeto de mestrado.

Este capítulo apresenta o *Class-Test*, um classificador automático de casos de teste concebido para auxiliar os profissionais na criação de planos de teste extensos. Como já dito, os planos de teste devem conter um determinado número de testes de cada tipo (e.g., testes negativos, de fronteira, interação, etc), número este fixado pelo arquiteto/designer de testes da empresa. Um dos maiores problemas enfrentados por esses profissionais é o tempo gasto na classificação manual dos testes pré-selecionados para compor planos extensos (com 1.000 testes, por exemplo).

Neste contexto, esta pesquisa teve como motivação principal diminuir o esforço e o tempo gasto no processo de seleção e classificação de testes. Assim, a função principal do *Class-Test* é classificar os casos de testes dados como entrada em categorias pré-definidas, facilitando a escolha dos testes que irão compor o plano sendo construído.

O *Class-Test* foi concebido utilizando técnicas clássicas de Aprendizagem de Máquina para Classificação de Texto. Esse sistema dispõe de três classificadores, um

para cada classe alvo: teste de Fronteira, teste Negativo, e teste de Interação. Foi necessário usar três classificadores nesta tarefa porque alguns casos de teste podem ser associados a mais de uma classe de teste ao mesmo tempo.

Quatro algoritmos foram usados na fase de aprendizagem de cada classe-alvo, totalizando 12 classificadores induzidos. Com base na precisão alcançada, foram selecionados os classificadores mais eficientes para compor o *Class-Test*. O capítulo 5 traz o detalhamento dos experimentos realizados com esses classificadores. A taxa de precisão alcançada na classificação automática foi superior a 90%, sendo equivalente à precisão da classificação manual dos mesmos dados, o que atesta a grande contribuição desse sistema em relação ao ganho de tempo.

A seção 4.1 apresenta o processo de concepção de novos planos de teste, desde a sua solicitação até a criação, ressaltando as dificuldades encontradas neste processo manual. Na seção 4.2, mostraremos uma visão geral do *Class-Test,* apresentado suas duas fases, o treinamento e a utilização do classificador. A seção 4.3 traz a metodologia de desenvolvimento do *Class-Test*. A seção 4.4 vis amostrar na prática o uso do *Class-Test* na criação de planos de teste. A seção 4.6 traz considerações finais sobre o trabalho aqui apresentado.

4.1 Criação de Planos de Teste

O CIn-BTC Motorola é um projeto que envolve diversos profissionais de Informática, de áreas distintas, com um só objetivo: executar testes em dispositivos móveis até encontrar faltas no software. Contudo, até chegar à execução das suítes de testes, é percorrido um longo caminho, no qual cada profissional tem seu papel e suas responsabilidades no processo geral de teste.

O primeiro passo desse processo é a *solicitação do teste*, feita através de um documento chamado de *Plano de Teste* (*test plan*). Esse documento contém todas as informações para iniciar uma execução do teste em um software (*Build*), incluindo:

1. Quando vai ser iniciada a execução;

2. O procedimento de configuração do ambiente para sua instalação;

3. A versão do software a ser testado;

4. Quantos testes serão executados em cada componente;

5. Quanto tempo vai levar esta execução e quantos testadores serão necessários.

Também podemos encontrar no Plano de Teste a definição da estratégia de teste, informando que tipo de teste organizacional (Fronteira, Negativo e Interação) será adicionado nesta execução, e quais abordagens (funcional ou estrutural) serão contempladas. Um plano de teste bem estruturado facilita a seleção dos testes a serem distribuído para cada componente na sua suíte de teste.

Os Casos de Teste usados no CIn-BTC são armazenados e gerenciados por uma ferramenta chamada de Central de Teste (*Test Central*). Essa ferramenta é um grande repositório de testes, sendo utilizados para criação de planos, acompanhamento do planejamento, execução de teste, etc.

O Plano de Teste é um documento onde o arquiteto de teste faz todo o planejamento das suas atividades, e a definição de como serão criados os outros planos. Além do Plano de Teste, existem outros documentos de planejamento no CIn-BTC: *Planning Pool*, Plano Mestre (*Master Plan*), Plano de Ciclo (*Cycle Plan*). Esses planos são criados totalmente independentes, para auxiliar na melhor distribuição dos testes. Esses planos são brevemente descritos a seguir.

- *Planning Pool* – é um pool de teste montado pelo arquiteto de teste, que seleciona de dentro do *Test Central* um conjunto máximo de testes para uma determinada linha de produto. Também podemos ver o *planning pool* como um repositório de teste, contendo testes de várias classes (boundary, negative, interaction, primary functionality, second functionality, stress, performance, load) e quantidades que serão aplicáveis ao produto a ser testado.

- Plano Mestre (*Master Plan*) - é um plano que contém um subconjunto dos testes do repositório *Planning Pool*; também pode ser visto como um plano base que serve como ponto de partida para criação do plano de ciclo (*Cycle Plan).*

- Plano de Ciclo (*Cycle Plan*) – este plano pode conter todos os testes do *Master Plan,* ou apenas um subconjunto deste. Este plano é definido pelo processo de

teste do CIn-BTC como plano onde os testes são populados, para serem executados pelos profissionais de testes.

No projeto CIn-BTC, os responsáveis pela criação desses planos são os arquitetos de testes. Os arquitetos podem estar associados a um ou mais componentes, tendo responsabilidade de planejar, criar, avaliar e executar os casos de testes, e ainda coletar os resultados dos planos. Para cada componente, é necessário ter um *planning pool* contendo seus casos de testes.

Os arquitetos enfrentam grandes dificuldades na criação e população dos Ciclos de teste, por causa da grande quantidade de casos de testes disponíveis para seleção, e por essa seleção ser manual, sendo necessário verificar todos os casos de teste disponíveis um a um. Esse processo manual é muito lento e demanda grande esforço por parte do arquiteto. De fato, esse esforço é requerido na construção de todos os tipos de planos: Planejamento Pool, Plano Mestre, Plano de Ciclo. Este processo de seleção é iniciado quando recebe o plano de teste solicitando a quantidade de teste de cada componente, estas quantidades são distribuídas com através dos seus tipos de testes organizacional (*boundary, negative e interaction*).

Examinando o processo de criação desses planos de teste, verificou-se a possibilidade de automatizar parte desse processo, realizando a classificação automática dos testes do *Planning Pool* nas suas classes correspondentes. Isso contribui muito para a diminuição do tempo e do esforço para popular os ciclos de teste.

Escolhemos abordar inicialmente as classes teste do tipo Fronteira, teste Negativo, teste de Interação, por serem muito comuns e numerosos, além de serem indispensáveis na criação de qualquer plano de teste. Além disso, esses casos de teste apresentam regularidade que podem ser capturadas por algoritmos de aprendizagem automática, técnica adotada neste trabalho de mestrado.

Veremos a seguir detalhes sobre o *Class-Test*, que contém os classificadores específicos desenvolvidos para cada classe alvo descrita acima.

4.2 Visão geral do Class-Test

Esta seção tem como objetivo dar uma visão geral do *Class-Test*, cuja função principal é classificar casos de testes em categorias pré-definidas, facilitando a escolha dos testes que irão compor o plano de teste sendo construído. Esse sistema dispõe de três classificadores, um para cada classe alvo: teste de Fronteira (*Boundary Test*), teste Negativo (*Negative Test),* e teste de interação (*Interaction Test*).

O *Class-Test* foi construído utilizando técnicas de Aprendizagem de Máquina, tendo duas fases: treinamento e uso. A fase de treinamento foi realizada três vezes, uma para cada classificador induzido. Como visto no capítulo 3, essa fase pode ser dividida em três etapas principais: aquisição do corpus, criação da representação dos documentos (incluindo a redução da dimensionalidade do vocabulário), e indução do classificador. A Figura 4.1 ilustra o processo geral de treinamento de classificadores.

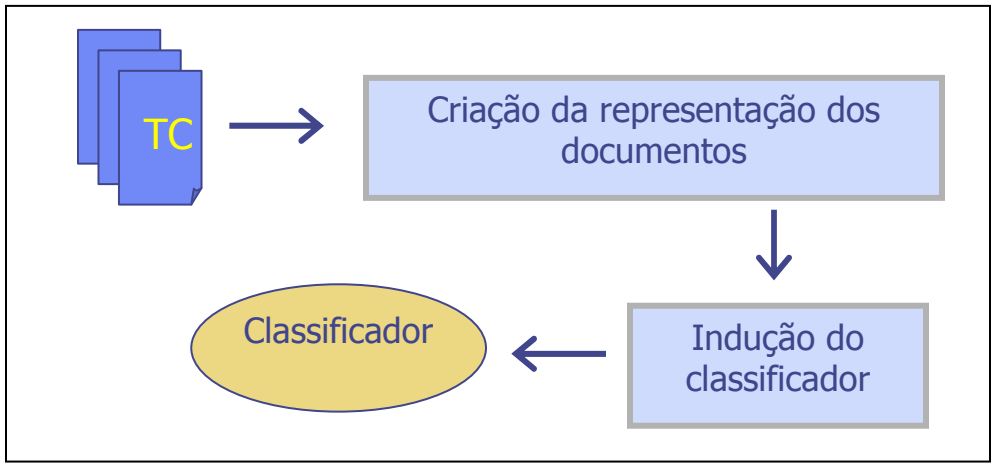

Figura 4.1 - Fase de treinamento de classificadores de texto.

O *corpus* utilizado para treinamento e teste dos classificadores foi composto por 879 casos de testes coletados do repositório de casos de testes do CIn-BTC, o *Test Central.* Os dados foram manualmente etiquetados com a ajuda de especialistas na área.

Quatro algoritmos foram usados na fase de aprendizagem de cada classe-alvo, com o intuito de verificarmos qual apresentava melhor desempenho para este problema. No total, foram criados 12 classificadores. A partir da precisão alcançada, foram

selecionados os classificadores mais eficientes para compor o *Class-Test*. A seção 4.3, a seguir, apresenta como cada etapa dessa fase foi realizada neste trabalho.

Na fase de uso, o *Class-Test* recebe como entrada Casos de Teste desconhecidos, e associa cada CT de entrada a uma ou mais das categorias-alvo (Figura 4.2). Os casos de testes de entrada são analisados pelos 3 classificados do *Class-Test*, e cada classificador indica se o CT pertence ou não à sua classe-alvo.

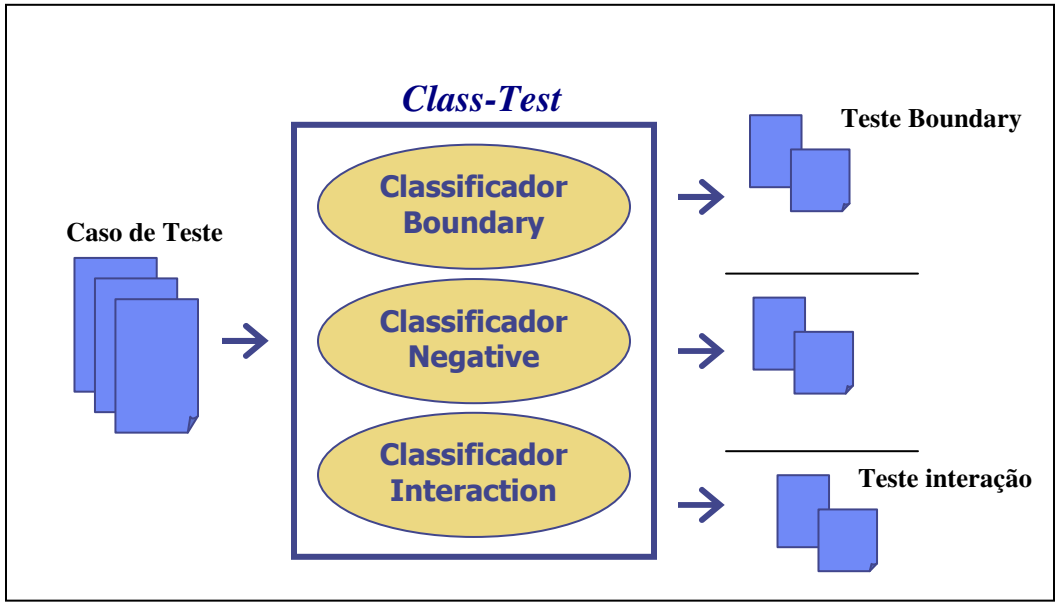

Figura 4.2 - Fase de Uso do `Class-Test`

O capítulo 5 traz o detalhamento dos experimentos realizados com esses classificadores. A taxa de precisão alcançada na classificação automática foi, em média, 90%, sendo equivalente à precisão da classificação manual dos mesmos dados, o que atesta a grande contribuição desse sistema em relação ao ganho de tempo.

A seção 4.3 a seguir detalha a fase de construção do *Class-Test*, seguida da seção 4.4, que apresenta a sua fase de uso.

4.3 Desenvolvimento do Class-Test

Veremos a seguir as etapas de criação dos classificadores que fazem parte do *Class-Test*: a aquisição dos documentos para montagem do corpus, pré-processamento

dos documentos, por último a utilização dos algoritmos de aprendizagem para indução dos classificadores através da ferramenta *WEKA* [Witten & Frank, 2000].

4.3.1 Aquisição dos Documentos

O projeto CIn-BTC utiliza com um repositório de casos de testes chamado Central de Teste (*Test Central*), que possui milhares de CTs das classes definidas neste trabalho (Capítulo 2). Os CTs são armazenados em planilhas Excel. Este repositório é utilizado por um grande número de pessoas espalhadas pelo mundo, e dá suporte a várias atividades, como de criação, seleção, atualização e remoção de casos de teste.

Os CTs utilizados para criar o corpus de treinamento e teste dos classificadores foram coletados a partir do *Test Central* com a ajuda da equipe de Arquitetura de Testes do CIn-BTC. Esse corpus consiste em 879 casos de testes de diferentes componentes, com a distribuição de 191 CTs da classe Fronteira, 338 CTs da classe Negativo e 350 CTs da classe Interação. Os dados foram manualmente etiquetados com a ajuda de especialistas na área.

As figuras 4.3, 4.4 e 4.5, a seguir, trazem exemplos de CTs de cada uma dessas classes (o texto dos CTs, originalmente em inglês, foi traduzido para Português).

Tipo de Execucao	Descrição do Caso	Procedimento	Resultado Esperado
Manual	..... Editar Contato - Apelido - Nome Longo	Objetivo: O caso de teste se o processo de edição do apelido com um nome longo é aceito pelo campo Apelido.	
	Requisitos:	....	
	Setup:	Marque o Item ... ON.	
	Condição Inicial	- Usuario está logado ... - A lista de contato não esta cheia.... - Um contato com o Apelido foi selecionado.	
	Notas:	Quando o apelido ..... maior do que o espaço disponivel	
	Numero dos Passos:		
	1	Edite o apelido usando um nome longo	o apelido é mostrado corretamente
	2	Confirme a operação	O contato foi editado com sucesso. O contato é mostrado com o nome longo corretamente.
	Condição Final:		
	Limpeza:		

Figura 4.3 - Exemplo de Caso de Teste da classe *Fronteira*.

Como visto no capitulo 2, o teste de Fronteira verifica se as variáveis do sistema estão obedecendo as fronteiras definidas (valores máximo e mínimo que a variável pode assumir), em busca de falhas.

Tipo de Execução	Descrição do Caso	Condição inicial	Ação do Passo	Resultados Esperado.
Manual	Login - Loger no IM quando ISP não esta configurado.	webSessiong esta configurado para acessa internet. O ISP não esta Configurado.	Conecte com o IM Server.	A conexao do IM Server não esta se conectando ao servidor. Uma tela é mostrada informando que o ISP não esta Configurado.

Figura 4.4 - Exemplo de Caso de teste *Negativo*.

O teste Negativo utiliza como entrada uma condição ou uma informação inaceitável, inválida, anormal ou inesperada, e verifica se o sistema devolve a mensagem de erro esperada.

Tipo de Execução	Descrição do Caso	Setup	N.	Executar passos	Resultado Esperado
Manual	Interação / Bluetooth / Rádio com diferentes Acessórios Bluetooth	Rádio FM é sintonizado para uma estação FM estéreo válido. Sinal de áudio é encaminhada para dispositivo Bluetooth.	1	Conectar o equipamento bluetooth no kit carro.	O Radio continua tocando normalmente.
			2	Troque a estação do radio no kit carro.	O Radio continua tocando normalmente.
			3	Deslique o Radio kit carro	o Bluetooth não esta tocando o son do kit carro .

Figura 4.5 - Exemplo do Caso de teste de *Interação*.

O teste de Interação é utilizado quando existe a necessidade de verificar a comunicação entre funcionalidades que foram testadas separadamente, e depois foram integradas em um mesmo software (i.e., o objetivo é verificar se essas funcionalidades estão se comunicando corretamente). Na figura 4.5, é mostrado um exemplo de CT que verifica a interação entre um dispositivo Bluetooth e um kit de som de carro.

4.3.2 Criação da representação dos Casos de Teste

A representação dos CTs selecionados do *Test Central* em planilhas Excel não pode ser diretamente usada para a indução automática de classificadores, sendo necessário criar-se uma representação de cada CT como uma lista de palavras.

O maior problema que enfrentamos para obter essa representação em forma de lista foi a falta de padronização da representação dos CTs na planilha. Observando-se os exemplos das figuras acima, pode-se notar que não existe um único padrão para a representação desses CTs, i.e., cada CT foi representado em uma estrutura diferente. Isso trouxe sérias dificuldades para a extração automática das palavras representativas dos CTs, que são oriundas dos campos Descrição, Condições iniciais, Passos e Resultados esperados.

A solução adotada não foi tentar extrair seus campos que compõem o caso de teste, devido à grande variedade de formatos existentes. Depois de muito trabalho, a solução encontrada foi a utilização de uma Api POI [Poi apache, 2008], que é uma solução *open source*, capaz de ler os formatos mais variados identificados inicialmente.

O CT no formato Excel é lido e transformado em uma única string, que contém todos os termos encontrados nos seus campos. Essa string é então transformada em uma lista que contém apenas os termos encontrados na descrição dos casos de teste, pois os sinais de pontuação são eliminados. Essa lista constitui a representação inicial do caso de teste. Como exemplo, vemos na Figura 4.6, abaixo, a lista de termos que representa o caso de teste da figura 4.4.

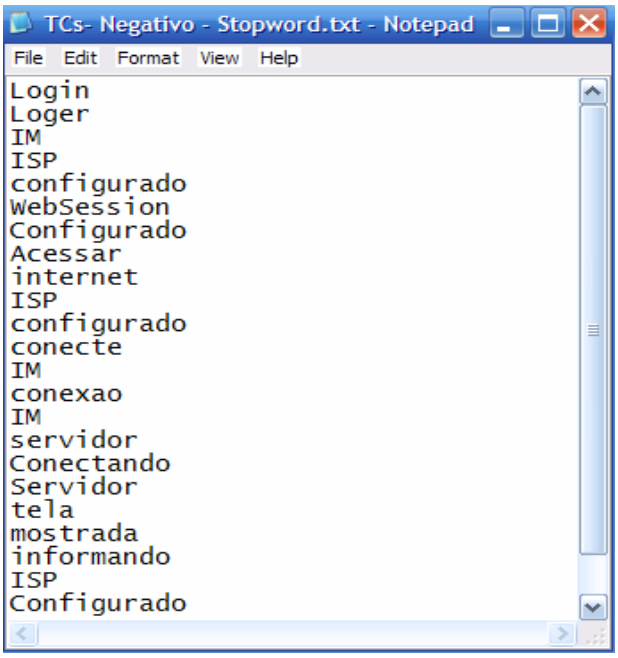

Figura 4.6 – Representação inicial do CT Negativo da Figura 4.4.

As representações iniciais dos CTs, em forma de lista, serão ainda processadas, até se obter uma representação adequada do corpus de treinamento e teste. Essas etapas serão vistas nas seções a seguir.

Redução da Dimensionalidade

Esta etapa visa reduzir a dimensionalidade das representações dos CTs obtidas na etapa anterior, uma vez que a quantidade de termos nas listas originais é muito grande, e nem todos os termos são relevantes para representar o documento em questão. Esta etapa pode ser decomposta em várias sub-etapas, sendo as mais importantes a eliminação de *Stopword* e aplicação de algoritmos de *Stemming*.

Como visto no capitulo 3, [Fox, 1992] define a 1ª sub-etapa de redução como um processo de eliminação de palavras que não discriminam bem o texto e, portanto, são inúteis para representar e distinguir os casos de testes uns dos outros. Este processo visou retirar todos os artigos, preposições e conjunções presentes nas representações iniciais dos CTs. Esta eliminação foi realizada através da implementação do algoritmo de Poter [Porter, 1980] e aplicada nos casos de testes adquiridos.

Com as listas de palavras já reduzidas, dá-se início à próxima sub-etapa, o uso de algoritmos de *Stemming*, que visa reduzir cada palavra da lista à sua provável raiz. Neste trabalho utilizamos o algoritmo Porter [Porter, 1980], que consiste na identificação das diferentes inflexões referentes à mesma palavra e sua substituição por um radical comum. Como resultado, cada CT passou a ser representado por uma lista de radicais, e não mais de palavras completas.

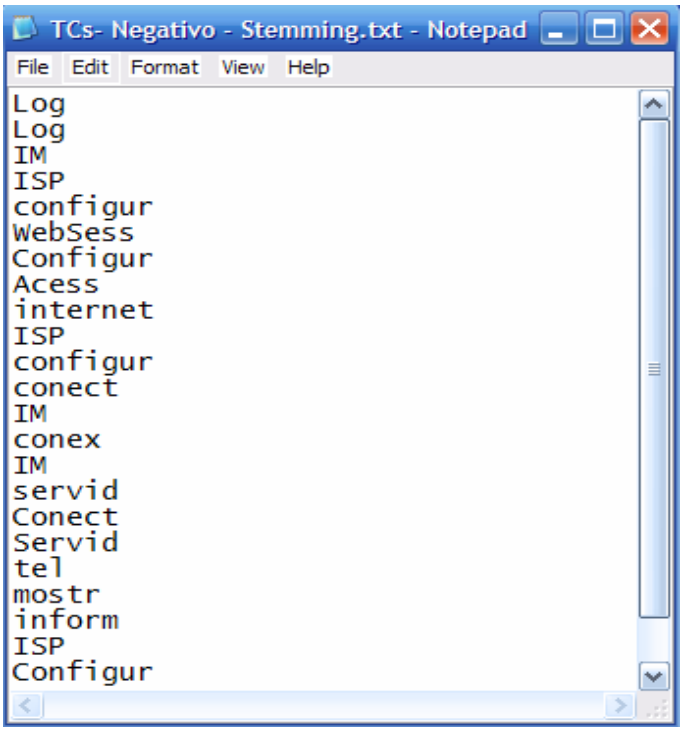

Figura 4.7 – Representação do CT da Figura 4.4 depois da redução de dimensionalidade.

Criação do Vocabulário do Corpus e Seleção de Atributos

Após o processo de redução de dimensionalidade, as listas de termos representando cada CT foram unidas, dando origem ao Vocabulário inicial do corpus. Esse vocabulário inicial contava com 2.789 termos.

Apesar das reduções feitas às listas de termos iniciais, o Vocabulário inicial ainda era muito longo, e continha termos irrelevantes. Então, esse vocabulário foi submetido à

etapa de seleção de atributos, com o objetivo de definir um conjunto de termos que melhor representasse a classe a ser aprendida, i.e., seus termos mais relevantes.

Para o cálculo de relevância, foi utilizado o escore sugerido por [Salton & Buckley, 1987]. A freqüência de cada termo no documento foi calculada, com o objetivo de identificar os termos com um alto grau de representatividade. Os termos com um valor alto de freqüência dentro da coleção foram então selecionados para compor o vocabulário final da base.

Utilizamos a ferramenta Luke do projeto Lucene Apache [Luke, 2008] para obter esses valores de freqüências automaticamente. Essa ferramenta recebeu como entrada as listas de termos que representam os CTs, e produziu uma lista geral de termos do corpus, junto com a freqüência de ocorrência de cada termo nesse conjunto. A partir dessa lista geral de termos, foram selecionados os atributos com maior freqüência de ocorrência, e com algum conteúdo semântico relevante para identificar a classe de teste sendo aprendida. Dessa forma, o Vocabulário inicial do corpus foi reduzido para um conjunto de 700 termos.

4.3.3 Indução dos Classificadores

Após a obtenção do Vocabulário final da base, é necessário criar as representações que serão usadas no treinamento dos algoritmos. Os dados de treinamento devem ser representados em matrizes binárias de Termos *versus* Documentos (CTs). A célula (*i,j*) recebe valor 1 se o termo j ocorre no CT *i*; e recebe valor 0 caso contrário. A coluna final indica se o CT pertence ou não à classe sendo aprendida. Veja como exemplo a Tabela 4.1 a seguir.

Tabela 4.1 - Matriz de Termos *vs* Casos de testes

	T1	T2	T3	T4	...	T*n*	Classe
CT1	1	1	0	0	...	1	1
CT2	0	1	1	1	...	0	1
CT3	1	0	1	1	...	0	0
...	...	...	...	...	...	...	...
CTn	1	1	1	0	...	1	0

Assim, a matriz definida para o corpus de treinamento e teste possui 879 linhas, indicando os casos de testes, e 700 colunas representando os termos selecionados na etapa de seleção de atributos.

Como foi mencionado nas seções anteriores, nosso objetivo é criar três classificadores, um para cada classe-alvo (Fronteira, Negativo e Interação). Assim, foi necessário criar três matrizes diferentes, uma vez que a coluna final teria seus valores modificados de acordo com a classe a ser aprendida.

É importante ressaltar aqui que o vocabulário usado em cada matriz é diferente, isto é, foi criado um vocabulário específico para cada classe-alvo, com base no corpus de exemplos positivos e negativos para cada classe. Os três vocabulários possuem 700 termos, porem esses termos podem variar (ainda que pouco) de classe para classe. Isto se deve ao fato de que um termo pode ser muito relevante para o aprendizado da classe Boundary, porém pode ser irrelevante para a classe Negative. Neste caso, ele não aparece no vocabulário desta última classe. Veremos mais detalhes sobre esses dados na seção a seguir.

A Ferramenta WEKA

Como dito, os classificadores do *Class-Test* foram criados através da ferramenta WEKA (*Waikato Environment for Knowledge Analysis*) [Witten & Frank, 2000]. Esta ferramenta oferece diversos algoritmos de aprendizado de máquina, tais como, Árvore de

decisão, Máquina de vetores de suporte (SVM), Naive Bayes e k-NN, que foram usados neste trabalho de mestrado. O WEKA implementa ainda outras técnicas de inteligência artificial que não foram utilizadas neste projeto de mestrado.

O WEKA é uma ferramenta de uso fácil para indução de classificadores. Para se utilizar desta ferramenta, primeiramente precisamos seguir alguns passos de preparação dos dados. A ferramenta é alimentada por arquivos em um formato padrão (extensão .arff), cuja estrutura é composta por três elementos: Relação, Atributos e Dados.

- **Relação -** a primeira linha do arquivo deve ser igual a @relation, seguida de uma palavra chave que identifique a relação ou tarefa sendo estudada.
- **Atributos -** um conjunto de linhas iniciadas com @attribute, seguido do nome do atributo e do seu tipo, que pode ser nominal ou numérico. Em uma tarefa de classificação supervisionada, conhecemos as classes das instâncias usadas para treinamento. As classes são o último atributo das instâncias.
    - **Atributos nominais** - neste caso, as alternativas devem aparecer como uma lista separada por vírgulas e cercada por chaves, e.g., {primeiro valor, segundo valor, terceiro valor}.
    - **Atributo numérico** – é um atributo contendo valores numéricos.
- **Dados** – aparecem depois de uma linha contendo @data. Cada linha deve corresponder a uma instância dos dados de entrada, e deve conter valores separados por vírgulas correspondentes (e na mesma ordem) dos atributos da seção Atributos. Se conteúdo é = 1 se o atributo (termo) ocorre no caso de teste, e 0 caso contrário.

O arquivo também pode conter linhas iniciadas com o sinal de percentagem (%). Estas linhas serão consideradas comentários e não serão processadas. A figura 4.8 ilustra um arquivo .arff.

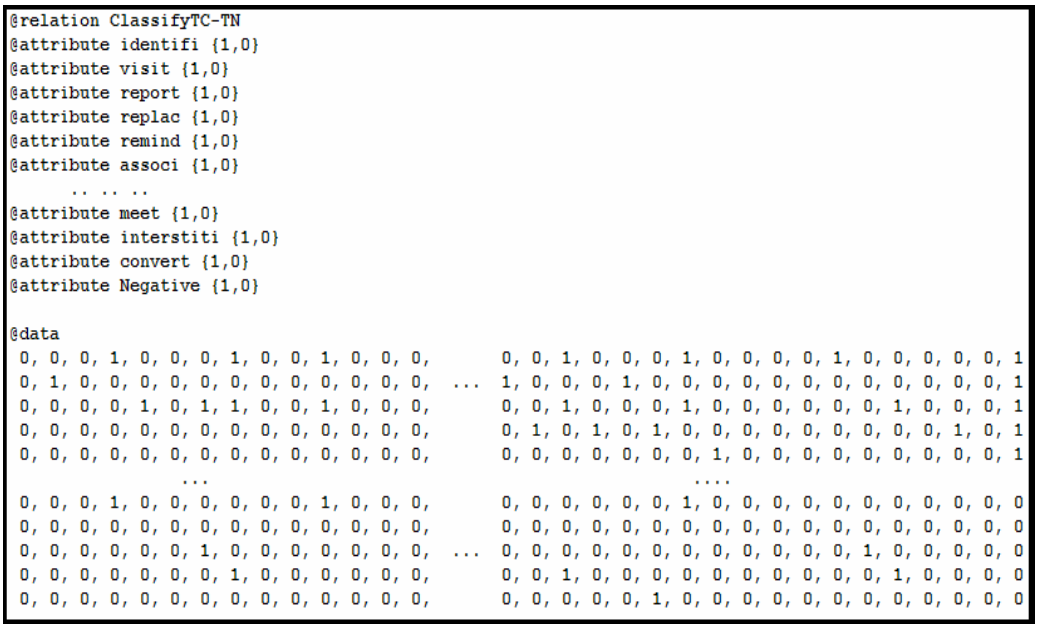

Figura 4.8 - Arquivo ARFF para experimentos

Treinamento dos Classificadores

Como dito, criamos três matrizes de relação binária Termos *vs* CT diferentes, uma para classe a ser aprendida. Após a criação do arquivo .arff correspondente a cada uma dessas matrizes, iniciou-se a indução dos classificadores desejados (Boundary, Negative e Interaction). Esses arquivos .arff contêm os dados necessários para criar os arquivos de treinamento e teste dos classificadores pela ferramenta WEKA.

Como já dito, quatro algoritmos foram usados na indução de classificadores de cada classe-alvo, com o intuito de verificarmos qual apresentava melhor desempenho para este problema. Os algoritmos utilizados foram escolhidos de famílias diferentes de algoritmos: SVM, Naive Bayes, k-NN e Árvore de decisão). No total, foram criados 12 classificadores. A partir da precisão alcançada, foram selecionados os classificadores mais eficientes para compor o *Class-Test*.

Os classificadores passaram por uma bateria de experimentos e calibração, até chegarem a um padrão aceitável de precisão na classificação. Após a finalização do treinamento, os algoritmos foram submetidos a testes com dados novos, para verificar se o aprendizado foi efetivo. A seguir, o classificador de cada classe com melhor precisão

foi selecionado para fazer parte do C*lass-Test*. O Capítulo 5 traz os detalhes sobre os testes e resultados obtidos.

4.4 Criando Planos de Teste com o Class-Test

O *Class-Test* visa contribuir para a diminuição do esforço e do tempo gasto na criação dos planos de testes. A figura 4.9 mostra a tela do *Class-Test.*

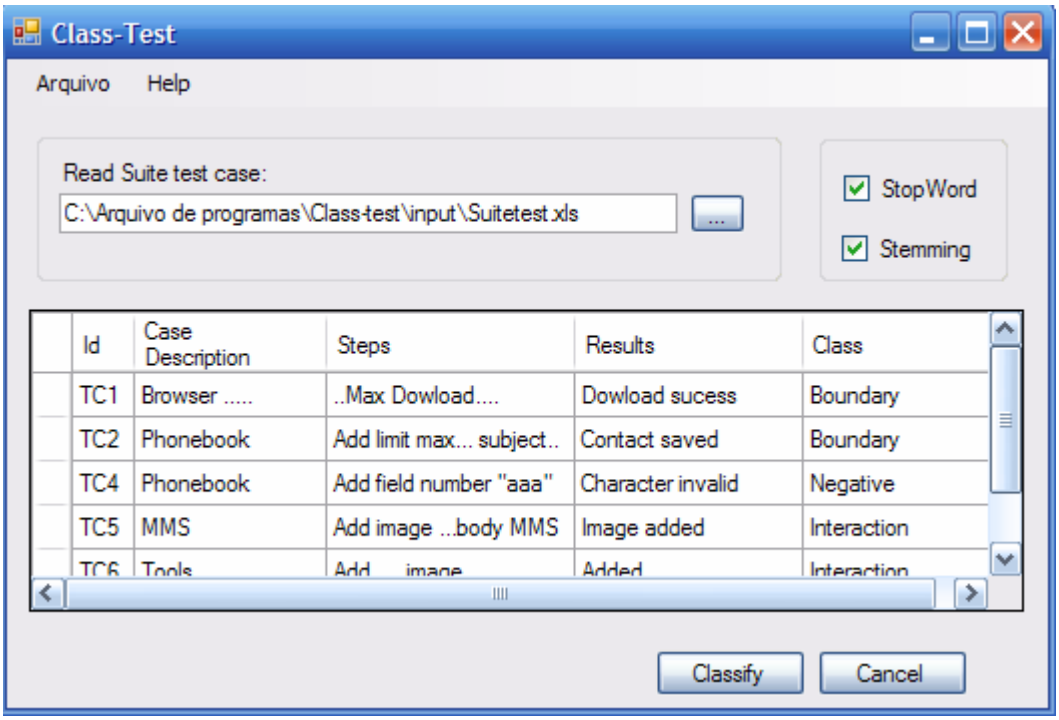

Figura 4.9 - Tela do *Class-Test*

O *Class-Test* é uma ferramenta composta por algumas funcionalidades de entrada de dados (casos de testes a serem classificados), opções de limpeza de dados (stopword e stemming), e a classificação do conjunto de entrada de casos de testes. Após a classificação, é mostrada uma tabela contendo todas as informações do caso de teste com sua respectiva categoria.

4.5 Considerações Finais

Este capítulo apresentou, em detalhes, o processo de treinamento e uso do *Class-Test*. Inicialmente discutimos a real necessidade da classificação automática dos casos de teste, e em seguida foi visto o processo geral de criação do *Class-Test*, a aquisição do corpus e as dificuldades na leitura dos casos de testes por falta de uma padronização das planilhas Excel. Vimos que a redução da dimensionalidade é etapa muito importante na eliminação do conteúdo irrelevante presente no corpus. Vimos também a necessidade da seleção de atributos, para reduzir o vocabulário geral do corpus. Por fim, vimos as fases de indução dos classificadores, e o uso do *Class-Test*.

O Capítulo 5, a seguir, traz o detalhamento dos testes e resultados obtidos nos experimentos com esses classificadores.

5 Testes e Resultados

No capítulo anterior, apresentamos todo o processo de construção do *Class-Test*, bem como seu uso para classificação de Casos de Testes. Como dito, o processo de aprendizagem inclui várias etapas de treinamento e testes, para verificação do desempenho do classificador sendo induzido. Este capítulo tem por objetivo apresentar os testes realizados durante a indução dos classificadores.

Foram realizados dois Estudos de Caso. O primeiro Estudo teve por objetivo verificar o desempenho dos 12 algoritmos induzidos, a fim de selecionar os que apresentavam melhor taxa de acerto na classificação de Casos de Teste, para então incluí-los no Class-Test. Para isto, foi utilizada a técnica de Validação Cruzada (*Cross Validation*) - ver seção 5.1.

O segundo Estudo de Caso teve por objetivo comparar o desempenho dos classificadores escolhidos para montar o *Class-Test versus* a precisão da classificação humana (por especialistas). Será feita uma discussão também sobre o esforço e o tempo gasto na classificação manual versus a classificação automática, no intuito de verificar qual é o ganho real de se adotar a abordagem automática (ver seção 5.2).

Por fim, a seção 5.3 conclui este capítulo trazendo algumas considerações finais sobre os experimentos apresentados aqui.

5.1 Estudo de Caso 1

Esta seção descreve em detalhes os testes realizados durante a criação dos classificadores de Casos de Testes descritos no Capítulo 4. O objetivo principal foi verificar, dentre alguns dos algoritmos de aprendizagem disponíveis, quais seriam os mais eficientes na sua precisão de classificação.

Os algoritmos selecionados para treinamento e testes estão mostrados na tabela 5.1 a seguir. Na 1ª coluna temos a família do algoritmo selecionado, e na 2ª coluna, a implementação do WEKA usada na indução dos classificadores. Foram escolhidos algoritmos de 4 famílias diferentes para verificar qual delas é mais adequada ao problema

sendo tratado neste trabalho de mestrado. Como veremos nesta seção, o algoritmo SMO apresentou os melhores resultados de precisão.

Tabela 5.1 - Algoritmos de Classificação.

Algoritmo de Classificação	WEKA
Árvore de Decisão	J48
Naive Bayes	Naive Bayes
SVM – Máquina de Vetor de Suporte	SMO
k-NN	IBK

A seção 5.1.1, a seguir, apresenta a metodologia dos testes, e a seção 5.1.2 mostra gráficos com os resultados obtidos no treinamento e teste dos algoritmos utlizados. Por fim, a seção 5.1.3 traz uma análise comparativa dos resultados alcançados neste estudo de caso, que serviu de base para a seleção dos algoritmos que constituem o *Class-Test*.

5.1.1 Metodologia do Experimento

Esse experimento foi realizado em 3 passos: (1) preparação do corpus de treinamento e teste por classe; (2) treinamento e testes; (3) coleta de resultados. Os resultados coletados serão vistos na seção 5.1.2.

Preparação dos corpora

Como visto no Capítulo 4, o corpus coletado consiste em 879 casos de testes de diferentes componentes, com a distribuição de 191 CTs da classe Fronteira, 338 CTs da classe Negativo e 350 CTs da classe Interação. Os dados foram manualmente etiquetados com a ajuda de especialistas na área.

Esse corpus inicial deu origem a três conjuntos de dados, um para cada classe-alvo distinta. O corpus inicial de treinamento e teste da classe Fronteira contou com 191

exemplos positivos e 688 exemplos negativos, obtidos da união dos exemplos das classes Negativo e Interação. Contudo, antes de iniciar o processo de treinamento, foi necessário "limpar" manualmente o conjunto de exemplos negativos, uma vez que alguns CTs podem ser classificados em duas ou mais classes diferentes ao mesmo tempo. Essa verificação foi manual, e consistiu em retirar do conjunto negativo todos os TCs que também pertencem à classe Fronteira. Esses TCs foram então inseridos no conjunto positivo, resultando em um corpus formado por 196 exemplos positivos e 683 exemplos negativos.

Esse mesmo procedimento foi adotado na criação dos corpora das outras duas classes-alvo, e o resultado final de distribuição dos dados é apresentado na Tabela 5.2.

Classe-alvo	Conj Positivo	Conj Negativo
Fronteira	196	683
Negativo	354	525
Interação	364	515

Tabela 5.2 – Corpora de Treinamento e Teste.

Como pode ser visto os dados para treinamento estão desbalanceados, o que não é ideal nas tarefas de indução automática de classificadores, uma vez que os algoritmos têm uma tendência de aprender melhor a classificar dados das classes majoritárias ver capítulo 3). A conseqüência desse desbalanceamento será observada nos resultados dos testes, apresentados nas seções a seguir.

Uma possível solução para balancear dados seria duplicar os dados positivos, obtendo-se assim conjuntos com quantidades mais semelhantes de dados. Contudo, isso também traz conseqüências negativas para o aprendizado, pois o classificador induzido poderá ficar *super-especializado* nos dados duplicados, e apresentar uma baixa precisão de classificação para novos dados positivos desconhecidos. Portanto, decidiu-se preservar os dados apresentado na tabela 5.2 para o treinamento dos classificadores.

Treinamento e Teste

Os arquivos .arff correspondentes aos corpora montados foram então submetidos à ferramenta WEKA. É importante relembrar que quatro algoritmos diferentes foram treinados para cada classe-alvo: SVM, Naive Bayes, k-NN e Árvore de decisão. No total, foram criados 12 classificadores.

A técnica utilizada para verificar o aprendizado dos algoritmos foi a Validação Cruzada (*K-Fold Cross Validation*) [Kohavi, 1995]. Essa técnica faz uma divisão aleatória dos documentos em *k* partições mutuamente exclusivas (chamadas de "*Folds*"), de tamanho aproximadamente igual a *n/k*, onde *n* é o tamanho do corpus de documentos. A seguir, (*k-1*) partições são usadas para treinar o algoritmo, e a partição restante é usada para testar os resultados. Esse procedimento é repetido *k* vezes, sendo que em cada rodada uma partição diferente é escolhida para o teste, e as *k-1* partições restantes são escolhidas para o treinamento. A medida de eficiência do algoritmo é dada pela média da eficiência calculada para cada iteração.

O WEKA tem essa técnica de treinamento já disponível, sendo necessário apenas preparar os arquivos .arff para cada classe, e definir o valor do parâmetro *k.* Nesse experimento, o valor de *k* escolhido foi 10. Então o WEKA dividiu cada corpora de entrada em 10 conjuntos de mesmo tamanho, e 10 realizou iterações para cada 10-*Fold* obtidos aleatoriamente. Ao final de cada uma dessas execuções, obtém-se o percentual de precisão daquela iteração.

Foram realizadas 100 *(k*k)* iterações para cada corpus de entrada (classe-alvo), e a média aritmética de precisão para cada classe foi calculada. Esta é a precisão média de classificação para cada classificador induzido (ver seção 5.1.2). A comparação entre os algoritmos usados neste trabalho é baseada na precisão média alcançada na *10-Fold Cross Validation* (ver seção 5.1.3).

Esta técnica pode ser usada para estimar a generalização do erro de um dado modelo de classificação, ou pode ser usada para a seleção de um modelo pela escolha de um em muitos modelos, o qual terá a menor generalização do erro estimado. A grande vantagem dessa técnica é que todos os documentos são usados tanto para treinamento quanto para teste. Assim, a divisão inicial do corpus influi menos no resultado final, uma

vez que qualquer documento será usado exatamente uma vez para teste e k-1 vezes para treinamento. A variação da estimativa é reduzida à medida que o parâmetro k aumenta. A desvantagem dessa modalidade é que o algoritmo de treinamento terá que ser re-executado k vezes, o que significa que levará k vezes mais computação para fazer a avaliação.

5.1.2 Resultados dos Testes

Veremos aqui os gráficos que apresentam a análise da precisão alcançada por cada classificador induzido com base nos corpora de cada classe-alvo. Como visto na seção anterior, usamos *10-Fold Cross Validation* para verificar esses valores de precisão. Assim, 100 iterações foram realizada para cada algoritmo *vs* classe-alvo, e os gráficos mostram a média aritmética de precisão dessas 100 iterações.

Na Figura 5.1, a seguir, é apresentado o gráfico da precisão alcançada pelo experimento com validação cruzada no classificador *Boundary*.

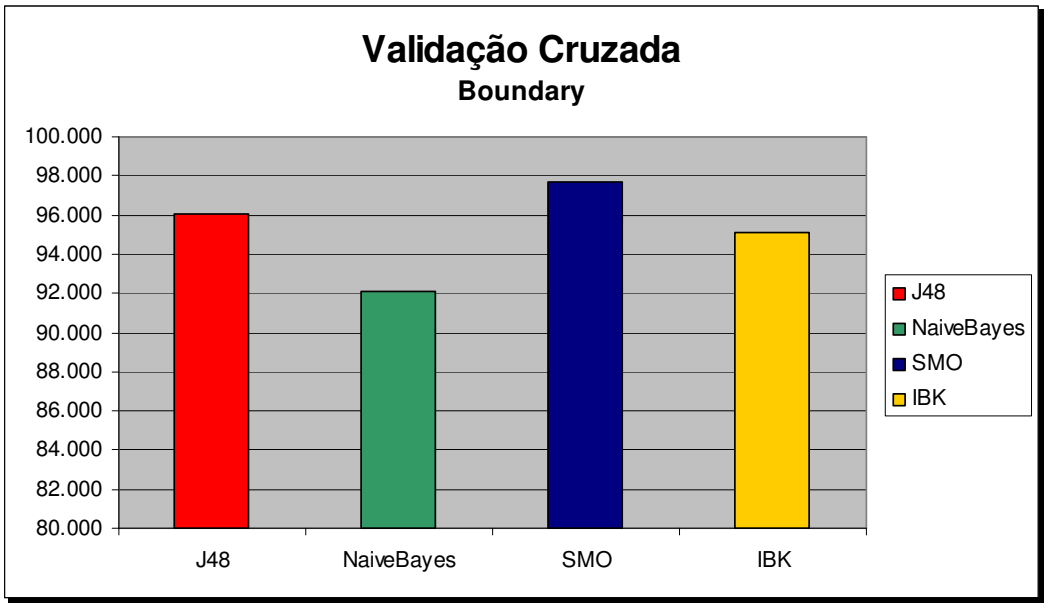

Figura 5.1 - Gráfico da Análise de Precisão do Classificador *Boundary*

Como pode ser visto na tabela 5.2, a quantidade de exemplos positivos usados para a geração do classificador *Boundary* é bem menor do que os exemplos negativos

(das outras duas classes-alvo). O número de exemplos da classe Fronteira é 196, e o da classe Não-Fronteira é 683 (ou seja, distribuição de mais ou menos 20% para 80%). Esse desbalanceamento dos dados pode trazer problemas para o aprendizado.

É sabido que, com maior desbalanceamento, os algoritmos têm uma tendência a acertar mais para a classe majoritária (que nesse caso é Não-Fronteira) do que para a classe minoritária (Fronteira). Em outras palavras, o algoritmo é melhor para dizer o que não é CT de Fronteira do que o inverso (ver tabela 5.3).

Tabela 5.3 – Desempenho dos algoritmos para a classe *Boundary*

Algoritmo de Classificação	Precisão Global	Classe	# exemplos	Precisão	Cobertura Verdadeiro positivo	Falso positivo
J48	0,960	Fronteira	196	0,951	0,893	0,013
		Não-Fronteira	683	0,970	0,987	0,107
Naive Bayes	0,921	Fronteira	196	0,895	0,781	0,026
		Não-Fronteira	683	0,939	0,974	0,219
SMO	0,976	Fronteira	196	0,979	0,944	0,006
		Não-Fronteira	683	0,984	0,994	0,056
IBK	0,950	Fronteira	196	0,915	0,878	0,023
		Não-Fronteira	683	0,965	0,977	0,122

Como pode ser visto no gráfico da Figura 5.1, o algoritmo de classificação SMO apresentou melhor taxa de precisão, seguido do algoritmo J48 e IBK, que também mostraram desempenho satisfatório. Por fim, o algoritmo Naive Bayes apresentou o pior desempenho em sua média de precisão. Contudo, ter uma taxa de precisão global boa não significa necessariamente que um algoritmo é mais adequado que outro, porque ele pode estar tendencioso para responder bem somente para a classe majoritária. Para a classe *Boundary*, apesar dos algoritmos SMO, J48 e IBK terem obtido boa taxa de precisão

global, o algoritmo SMO apresentou um melhor desempenho se consideramos, por exemplo, as diferenças entre taxa de verdadeiros positivos (cobertura) obtida para as classes negativa e positiva.

A Figura 5.2 apresenta o gráfico da precisão alcançada pelo experimento com validação cruzada no classificador *Interaction*. Aqui também o algoritmo SMO obteve a melhor taxa de precisão. O IBK foi o algoritmo que chegou mais próximo do SMO, enquanto o J48 e Naive Bayes tiveram desempenho mais baixo, não apresentando a mesma eficiência dos outros algoritmos. Ao contrário do que ocorreu em *Boundary*, no caso do classificador *Interaction*, a diferença de desempenho obtido pelos melhores algoritmos (i.e., SMO e IBK) para as classes majoritária e minoritária não é tão diferente. Isso ocorre aqui pelo fato do conjunto de dados para o classificador *Interaction* estar mais balanceado.

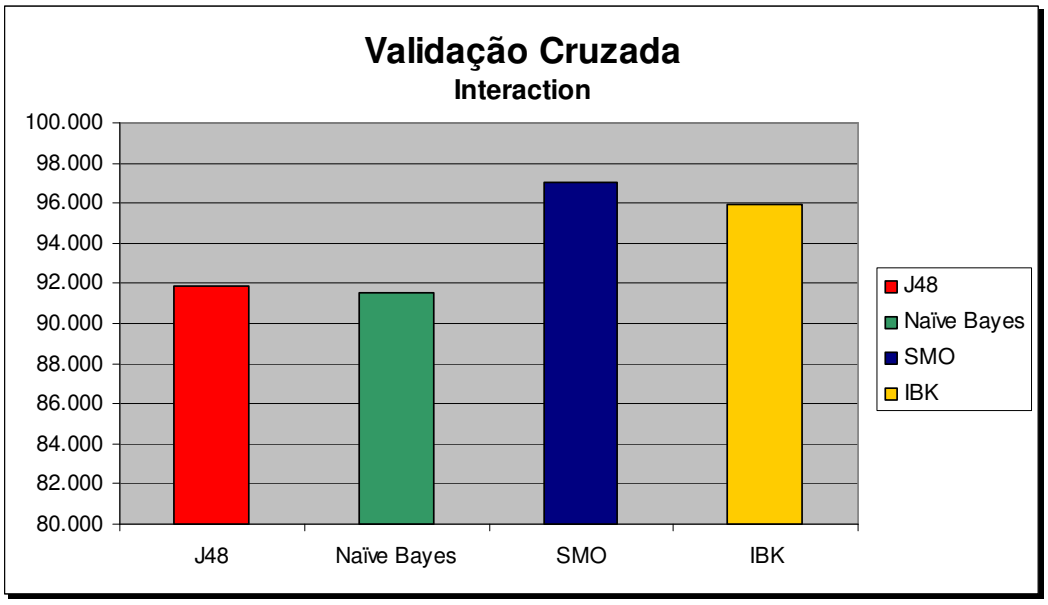

Figura 5.2 - Gráfico da Analise de Precisão do Classificador *Interaction*

Tabela 5.4 - Desempenho dos algoritmos para a classe *Interaction*

Algoritmo de Classificação	Precisão Global	Classe	# exemplos	Precisão	Cobertura Verdadeiro positivo	Falso positivo
J48	0,912	Interação	364	0,915	0,893	0,058
		Não-Interação	515	0,926	0,942	0,107
Naive Bayes	0,909	Interação	364	0,866	0,962	0,105
		Não-Interação	515	0,971	0,895	0,038
SMO	0,947	Interação	364	0,959	0,959	0,029
		Não-Interação	515	0,971	0,971	0,041
IBK	0,954	Interação	364	0,944	0,975	0,041
		Não-Interação	515	0,982	0,959	0,025

O resultado do experimento com o classificador *Negative* pode ser visto na figura 5.3. Neste caso, o algoritmo SMO não apresentou a melhor taxa de precisão, tendo sido superado pelo IBK. Contudo, a diferença de precisão foi tão pequena que pode ser considerado um empate entre os 2 algoritmos. Os algoritmos J48 e Naive Bayes apresentavam os piores resultados.

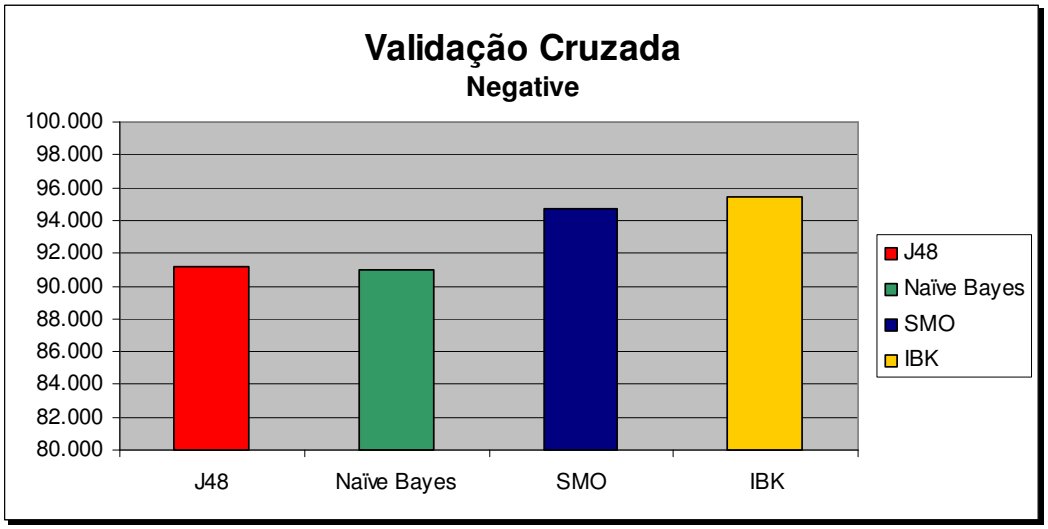

Figura 5.3 - Gráfico da Analise de Precisão do Classificador *Negative*

Tabela 5.5 - Desempenho dos algoritmos para a classe *Negative*

Algoritmo de Classificação	Precisão Global	Classe	# exemplos	Precisão	Cobertura Verdadeiro positivo	Falso positivo
J48	0,912	Negativo	354	0,913	0,887	0,057
		Não-Negativo	525	0,925	0,943	0,113
Naive Bayes	0,909	Negativo	354	0,945	0,822	0,032
		Não-Negativo	525	0,890	0,968	0,178
SMO	0,947	Negativo	354	0,928	0,946	0,050
		Não-Negativo	525	0,963	0,950	0,054
IBK	0,954	Negativo	354	0,934	0,958	0,046
		Não-Negativo	525	0,971	0,954	0,042

A seguir, veremos como foi realizado o segundo estudo de caso, e os resultados comparativos de classificação manual com automática.

5.2 Estudo de Caso 2

Este segundo Estudo de Caso teve como objetivo comparar o desempenho da classificação manual *vs* a classificação automática. Para isto, foram convidados dois profissionais de teste, um experiente e um iniciante. O objetivo era medir a precisão e o tempo gasto com a classificação manual, a fim de comparar os resultados da classificação manual com a classificação automática.

Embora a classificação manual apresente, em geral, precisão maior do que a automática, a enorme quantidade de documentos a classificar demanda um grande esforço e tempo dos profissionais responsáveis por essa tarefa.

5.2.1 Metodologia do Experimento e Resultados

Este Estudo de Caso foi realizado com o auxílio de dois profissionais de testes, um experiente (Testador 1) e um novato (Testador 2). Cada profissional recebeu uma amostra de 60 casos de testes selecionados a partir do corpus de CTs original, sendo 20 de cada classe-alvo.

Os casos de testes foram adicionados aleatoriamente em uma planilha eletrônica, no intuito de embaralhar as classes, para não facilitar a classificação manual. Foram medidos a taxa de precisão e o tempo gasto na classificação (ver tabela 5.5)

Classificação Manual			
	Precisão	Erro	Tempo
Testador 1	98%	2%	16 min.
Testador 2	90%	10%	25 min.

Tabela 5.5 - Classificação Manual

Como pode ser visto na tabela, o primeiro testador teve um melhor desempenho em sua precisão e no tempo de esforço de execução, enquanto o outro testador teve um aproveitamento inferior ao primeiro. Ambos os testadores tiveram o mesmo ambientes de execução da classificação, o único diferencial entre eles é a experiência, pois o primeiro é um testador experiente e o outro é iniciante. Portanto, podemos concluir que experiência do testador agrega um ganho alto na tarefa da classificação.

5.2.2 Classificação Manual *vs* Classificação Automática

A mesma amostra de 60 CTs foi submetida aos classificadores automáticos, a fim de possibilitar a comparação dos resultados obtidos (ver Tabela 5.6 a seguir). São apresentados os melhores e os piores resultados obtidos na classificação automática

(obtidos com o uso de classificadores diferentes) para comparar com a classificação manual.

Precisão das classificações automática e manual		
	Maior Desempenho	Menor Desempenho
Automática	97.64 %	90.99 %
Manual	98.00 %	90.00 %

Tabela 5.6 - Precisão da classificação Manual *vs* Automática

De acordo com a tabela 5.6, podemos observar que a duas técnicas de classificação obtiveram resultados equivalentes, tanto no melhor como no pior desempenho. A classificação manual teve um desempenho muito próximo ao do melhor algoritmo de classificação.

Contudo, a classificação automática apresenta um ganho significativo no tempo de categorização dos CTs (ver Tabela 5.7 a seguir).

Desempenho das classificações automática e manual			
	Precisão	Erro	Tempo
Testador 1	98%	2%	16 min.
Testador 2	90%	10%	25 min.
Melhor Classif. Automático	97.64 %	2,36%	5 min.

Tabela 5.7 – Desempenho geral da classificação Manual *vs* Automática

A partir desses resultados, podemos concluir que,quando comparada à classificação manual, a classificação automática traz mais ganhos para o processo geral

de categorização de documentos, uma vez que alcança taxas equivalentes de precisão, porém em um tempo significativamente menor.

5.3 Considerações Finais

Neste capítulo, foram apresentados os testes e resultados obtidos com os experimentos da classificação automática, bem como da classificação manual. Apresentamos dois estudos de casos, onde o primeiro teve como objetivo comparar a precisão dos quatro algoritmos de aprendizagem de máquina utilizados para induzir classificadores neste trabalho. Todos os algoritmos alcançaram boas taxas de precisa, sendo o SMO o melhor deles em duas das classes-alvo.

O segundo estudo de caso teve como objetivo realizar um experimento de classificação manual, a fim de comprara os resultados com a abordagem automática. Aqui foi verificada a superioridade da classificação automática, pois as taxas de precisão alcançadas foram equivalentes, mas esta última abordagem traz uma grande economia de tempo dos profissionais de teste, que poderão então dedicar seu tempo à criação dos planos de teste, e a outras tarefas necessárias no processo geral de teste de software.

6 Conclusão

Nesta dissertação, foi apresentado o *Class-Test*, uma ferramenta para auxílio na classificação automática de casos de testes em categorias pré-definidas.

Inicialmente foi realizado um estudo sobre a área de testes de software, relatado no Capítulo 2. Foi detalhado o processo de desenvolvimento de software, suas fases, e tipos de testes. Foram apresentadas também técnicas e abordagens de testes. A seguir, foi apresentada brevemente a área de classificação de texto no Capítulo 3. Esse capítulo abrangeu algumas etapas na indução de classificadores, como a redução da dimensionalidade, e a seleção de características para montar o vocabulário da base. Por fim, foram vistos alguns algoritmos de classificação de automática de texto.

A partir desse levantamento bibliográfico, e dos objetivos iniciais que motivaram essa pesquisa, concebemos uma ferramenta para classificar casos de testes, o *Class-Test*. Essa ferramenta tem por objetivo diminuir o esforço e o tempo gasto no processo de classificação manual. Esta ferramenta é voltada para os profissionais de teste, que através dela poderão classificar os casos de testes automaticamente. Alguns experimentos foram realizados utilizando um *corpus* de testes composto por 879 CTs com a distribuição de 191 casos de testes do tipo Fronteira (*Test Boundary*), 338 casos de testes do tipo Negativo (*Test Negative),* 350 Casos de testes do tipo interação (*Test Interaction).*

Os experimentos realizados foram vistos em dois estudos de casos. O primeiro estudo teve como objetivo avaliar, dentre os quartos algoritmos de aprendizagem selecionados, qual apresentava melhor precisão para o corpus em questão (o algoritmo SVM – Máquina de Vetores de Suporte apresentou melhor desempenho nesse estudo). Já o segundo estudo de caso teve como objetivo comparar a precisão da classificação automática versus a classificação manual. A partir desses experimentos, observou-se que aprecisão da classificação automática é equivalente à manual, contudo os classificadores trazem um ganho grande de tempo para os testadores.

Este trabalho foi desenvolvido como parte do projeto Test Research Project do CIn/BTC, em uma parceria entre o CIn-UFPE e a Motorola.

6.1 Contribuições

Listamos a seguir as principais contribuições deste trabalho, destacando sua originalidade:

- Utilização da técnica de aprendizagem de máquina para tratar problemas classificação de casos de testes em um ambiente empresarial.
- Criação do *Class-test*, uma ferramenta que auxilia o time de arquitetura de testes do CIn-BTC na criação dos planos de testes.

6.2 Trabalhos Futuros

Podemos vislumbrar diversos trabalhos futuros que estenderiam o que foi aqui apresentado. A seguir listamos algumas extensões que consideramos mais importantes.

Criação de uma interface Amigável para uso do *Class-Test* - Como sabemos, o uso do *Class-Test* como classificador de casos de teste só é possível se os dados de a serem classificados estiverem representados em arquivos .arff. Contudo, os arquitetos de software não estão familiarizados com essa representação, que de fato não é simples de construir. Assim, o principal trabalho futuro indicado aqui é a construção de um módulo de pré-processamento automático, capaz de receber uma suíte de casos de teste em uma planilha Excel, e devolver o arquivo .arff como saída.

Ampliação das categorias cobertas pelo *Class-Test* – como visto em capítulos anteriores, o CIn-BTC trabalha com 8 classes (tipos de teste): *Negative, Boundary, Interaction, Stress, Performance, Load, Primary Functionality* e *Second Functionality.* O *Class-Test* cobriu apenas 3 dessas classes, sendo necessária a sua ampliação para lidar também com os outros tipos. Um trabalho realizado por uma equipe de alunos do curso de Imersão Tecnológica do CIn-BTC - Turma 11[2] desenvolveu classificadores para as classes *Stress, Performance* e *Load*. Esses classificadores foram desenvolvidos no mesmo arcabouço do *Class-Test*, sendo fácil a sua incorporação à nossa ferramenta. Contudo,

[2] Alunas Carolina Fernandes, Polyanna Mendonça e Catuxe Varejão.

restam ainda os testes de *Primary Functionality* e *Second Functionality,* que não podem ser tratados usando Aprendizagem de máquina (ver item a seguir).

Criação manual de Classificadores para testes de *Primary* e *Second Functionality* – Os testes de *Primary Functionality e Second Functionality* não são classificados com base no texto dos seus passos, e sim com base no software a ser testado. Testes de *Primary Functionality* abordam as funcionalidades principais do software a ser testado, enquanto os testes de *Second Functionality* têm a finalidade de cobrir funcionalidade secundarias do software. Essa classificação é uma tarefa difícil, pois requer conhecimento sobre os testes e sobre o software a ser testado. Por isso, essa classificação é sempre realizada pelos arquitetos (especialistas) responsáveis pela criação do plano de teste. Este trabalho futuro tem como objetivo investigar esse problema, verificando se é possível utilizar Processamento de Linguagem Natural , juntamente com conhecimento sobre as funcionalidades do software, a fim de criar classificadores para essa classes. Neste caso, os classificadores serão criados manualmente, pelo uso de Base de Conhecimento e Inferência.

7. Referências Bibliográficas

[Aha & Kibler, 1991] AHA, D., and D. KIBLER. *"Instance-based learning algorithms",* Machine Learning, vol.6, 1991. pp. 37-66.

[Breiman et al., 1984] BREIMAN, L., FRIEDMAN, J. H., OLSHEN, R. A. e STONE, C. J. *Classification and Regression Trees.* Belmont, CA: Wadsworth, 1984.

[British ST, 1998] BRITISH STANDARDS INTITUTION. *Software testing. Vocabulary. (BS 7925-1:1998).* 15 Ago. 1998. 16p. ISBN: 0580295559.

[Burges, 1998] BURGES, C. J. C. *A Tutorial on Support Vector Machines for Pattern Recognition*. Kluwer Academic Publishers, Boston, 1998.

[Cavalcanti & Gaudel, 2007] A. CAVALCANTI and M.-C. GAUDEL. Testing for refinement in CSP. In *Formal Methods and Software Engineering, ICFEM 2007*, volume 4789 of *Lecture Notes in Computer Science*, pages 151-170. Springer Verlag, 2007.

[Cherkassky & Mulier, 1998] Cherkassky, V.e MULIER F. *Learning from data: concepts, theory, and methods*. John Wiley & Sons, Inc.1998.

[Cortes & Vapnik, 1995] CORTES, C., VAPNIK, V. *Support-vector networks*. Machine Learning, 20(3), 1995, pp.273-297.

[Cristianini & Shawe-Taylor, 2000] CRISTIANINI, N.; SHAWE-TAYLOR, J. *An Introduction to Support Vector Machines and other kernel – based learning methods*. Cambridge Unversity Press, 2000.http://www.support-vector.net.

[Delamaro, 1997] M. E. Delamaro. *Mutação de Interface: Um Critério de Adequação Inter-procedimenta para o Teste de Integração*. PhD thesis, Instituto de Física de São Carlos – Universidade de São Paulo, São Carlos, SP, June 1997.

[Demillo et all., 1978] DEMILLO, R. A et all. *Hints on Test Data Selection: Help for the Practising Programmer*. New York: Computer, v.11, n 4, p. 31-41, 1978.

[Domingos & Pazzani, 1997] DOMINGOS, P.; PAZZANI, M. *On the Optimality of the simple Bayesian Classifier under Zero-one Loss*. Machine Learmng, 29 (2/3), pp. 103, 1997.

[Fox, 1992] FOX, C. Lexical analysis and stoplists. In: FRAKES, W. B.; BAEZA-YATES, R. A. (Ed.). *Information Retrieval: Data Structures & Algorithms. Upper Saddle River*, New Jersey: Prentice Hall PTR, 1992. p. 102-130.

[Jalote, 1994] P. JALOTE. *Fault Tolerance in Distributed Systems*. Englewood Cliffs: Prentice Hall, 1994.

[Jorgensen, 2002] JORGENSEN, P. C. *Software Testing: A Craftsman's Approach. 2 ed*. CRC Press, 2002.

[Harman, 1991] HARMAN, D. *How effective is Suffixing?*. Journal of the American Society for Information Science 42, 1991, pp. 7-15.

[Hunt et al., 1966] HUNT E. B., MARIN, J., & STONE, P. J. *Experiments in induction.* New York: Academic Press, 1966.

[IEEE, 1998] The Institute of Electrical and Electronics Engineers. IEEE Std 829: Standard for Software Test Documentation. New York: IEEE Computer Society, September, 1998.

[Ingargiorla, 1996] INGARGIOLA, Giorgio. *Building Classification Models: ID3 and C4.5.* Disponível por WWW em: http://www.cis.temple.edu/~ingargio/cis587/readings/id3-c45.html, 1996.

[Kohavi, 1995] KOHAVI, R. *A study of cross-validation and bootstrap for accuracy estimation and model selection*. International Joint Conference on Artificial Intelligence (IJCAI), 1995.

[Korfhage, 1997] KORFHAGE, R. R. *Information Retrieval and Storage*. New York: John Wiley & Sons, 1997.p. 349.

[Lewis, 2000] WILLIAM E. LEWIS, *Software Testing and Continuous Quality Improvement, Auerbach CRC Press LLC*, New York, 2000, ISBN 1-8493-9833-9.

[Lewis & Ringuette, 1994] LEWIS,D.D.,RINGUETTE,M. *A Comparison of Two Learming Algorithms for Text Categorization* In: Symposium on Document analysis and IR,ISRI, Las Vegas, 1994.

[Lovins, 1968] LOVINS, J.B. *Development of a stemming algorithm.* Mechanical Translation and Computacional Linguistics 11(1-2), 1968, p. 22-31.

[Luke, 2008] LUKE - Lucene index Toolbox. Apache Lucene – Available at: http://lucene.apache.org/java/docs/ and http://www.getopt.org/luke/ Accessed in: September 2008.

[McCallum & Nigam, 1998] McCALLUM, A. K.; NIGAM, K. *A Comparison of event models for naïve Bayes text classification.* In: Processdings of the 1st AAAI Workshop on Learning for Text Categorization, Pages 41-48, Madison, USA, 1998.

[McGregor & Sykes, 2001] J. D. McGregor and D. A. Sykes. *A Practical Guide to Testing Object- Oriented Software*. Addison-Wesley, 2001.

[Mitchell, 1997] MITCHELL, T. *Machine Learning*, McGraw Hill, 1997.

[Moens, 2000] MOENS, M. F. *Automatic indexing and abstract of document texts.*

Massachusetts: Kluwer Academic Publishers, 2000.

[Moreira, 2003] MOREIRA FILHO, T.R.; RIOS, E. *Projeto & Engenharia de software: Teste de Software* . Rio de Janeiro: Alta Books, 2003.

[Myers, 1979] GLENFORD J. MYERS. *Art of Software Testing. John Wiley & Sons, Inc.,* New York, NY, USA, 1979.

[Myers, 2004] MYERS, Glenford J., John Wiley & Sons, *The Art of Software Testing*, 2, Nova Jérsei: 2004.

[Offutt & Hayes, 1996] A. JEFFERSON OFFUTT, JANE HUFFMAN HAYES: *A Semantic Model of Program Faults*. ISSTA 1996: 195-200.

[Peters, 2001] PETERS, J. *Engenharia de Software*. Rio de Janeiro: Campus, 2001.

[Pressman, 1997] R. S. Pressman. *Software Engineering - A Practitioner's Approach*. McGraw-Hill, 4 edition, 1997.

[Pressman, 2000] R. S. PRESSMAN. *Software Engineering - A Practitioner's Approach*. McGraw-Hill, Nova York, NY, 5th edition, 2000.

[Pressman, 2002] PRESSMAN, S. Róger. *Engenharia de Software*. Rio de Janeiro, RJ: Makron Books, 2002. SOMMERVILLE, Ian. Software Engineering. SP: Addison Wesley, 2003.

[Pressman, 2006] PRESSMAN, Roger S. *Engenharia de Software*. 6ª Edição São Paulo: McGraw-Hill, 2006.

[Price, 1991] PRICE, A.M.A. *Teste em Engenharia de Software, Workshop em Programação Concorrente, Sistemas Distribuídos e Engenharia de Software*. São Carlos,

02 a 12 de dezembro de 1991, Universidade de São Paulo, pp.93-104, 1991.

[Poi Apache, 2008] Api Poi Apache. http://poi.apache.org/ ultimo acesso em: Dezembro de 2008.

[Porter, 1980] PORTER, M.F *An algorithm for suffixing stripping*. Program 14(3), 130-137.

[Quinlan, 1986] Quinlan, J.R. *Induction of Decision Trees.* Machine Learning 1,1,1986, p. 81-106.

[Quinlan, 1993] Quinlan, J. R. *C4.5: Programs for Machine Learning.* San Mateo, CA: Morgan Kaufman, 1993.

[Rapps & Weyuker, 1985] S. RAPPS and E. J. WEYUKER. *Selecting software test data using data flow information*. IEEE Transactions on Software Engineering, SE-11(4):367–375, April 1985.

[Rijsbergen, 1979] RIJSBERGEN, C. V. *Information Retrieval*. 2nd ed. London: Butterworths, 1979. p.147.

[Rizzi et al., 2000] RIZZI, C.; WIVES, L.; OLIVEIRA, J., ENGEL, P. *"Fazendo uso da Categorização de Textos em Atividades Empresariais*". In: International Symposium on Knowledge Management/Document Management – ISKDM/DM, 2000, Curitiba. Anais. p.251-268.

[Rocha et al, 2001] ROCHA, A.R.C. da; MALDONADO, J.C.; KIVAL, C.W. *Qualidade de Software: Teoria e Prática*. São Paulo: Prentece Hall, 2001.

[Russel & Norvig, 2002] RUSSEL, S. & NORVIG, P. *Artificial Intelligence: A Modern Approach* (2nd Edition). Prentice Hall Series in Artificial Intelligence. 2002.

[Salton & Buckley, 1987] SALTON, G., BUCKLEY, C. *Improving Retrieval Performance by Relevance Feedback*. Ithaca, New York. 1987.

[Salton & Macgill, 1983] SALTON, G.; MACGILL, M. J. *Introduction to Modern Information Retrieval*. New York: McGRAW-Hill, 1983. p. 448.

[Scholkopf & Smola, 2002] SCHOLKOPF, B. & SMOLA, A. J. *Learning with Kernels*.

The MIT press, 2002.

[Sebastiani, 2002] SEBASTIANI, F. *Machine learning in automated text categorization*, ACM Computing Surveys, 34 (2002), 1-47.

[Tomazela & Maldonado, 1997] TOMAZELA, M. G. M; MALDONADO, J. C. *Avaliação do Custo de Aplicação dos critérios potenciais usos no teste de programa Cobol*. In: Workshop do Projeto Validação e Teste de Sistemas de Operação, Águas de Lindóia. Janeiro 1997. Anais pp. 147-159.

[UML2TP, 2003] UML2TP. UML 2.0 Testing Profile Specification. Version 2.0. Final Adopted Specification. PTC/03-08-03. OMG, ago. 2003. Disponível em: http://www.omg.org/technology/documents/profile_catalog.htm.

[Vapnik, 1998] VAPNIK, V.N. *Statistical Learning Theory*. John Wiley & sons, 1998.

[Vapnik, 1999] VAPNIK, V.N. *An Overview of Statistical Learning Theory*. In IEEE Trans. On Neural Networks, vol.10 (5), 1999, pp.988-999.

[Walton et al, 1995] G. H. WALTON, J. H. POORE, and C. J. TRAMMELL. *Statistical Testing of Software Based on a Usage Model*. Software - Practice and Experience, 25(1):97–108, 1995.

[Wiener et al., 1995] WIENER, E., PEDERSEN, L.O, WEIGEND, A.S., *A Neural Network Approach to Topic Spotting*. In: Proceedings of the symposium on Document Analysis and Information Retrieval, Las Vegas, US, 1995, pp.317-332.

[Witten & Frank, 2000] WITTEN, H.; FRANK, E. Data Mining: *Practical machine learning tools with Java implementations*. Morgan Kaufmann, San Francisco, 2000.

[Yang & Pedersen, 1997] YANG, Y. AND PEDERSEN, J. O. 1997. *A Comparative Study on Feature Selection in Text Categorization.* In Proceedings of the Fourteenth international Conference on Machine Learning. D. H. Fisher, Ed. Morgan Kaufmann Publishers, San Francisco, CA, 412-420.