



INTELIGÊNCIA COMPUTACIONAL NA AVALIAÇÃO DE CÓDIGOS EM UM
SISTEMA COMPLEXO DE DETECÇÃO COM DESENVOLVIMENTO
COLABORATIVO

Andressa Andrea Sivolella Gomes

Dissertação de Mestrado apresentada ao
Programa de Pós-graduação em Engenharia
Elétrica, COPPE, da Universidade Federal do
Rio de Janeiro, como parte dos requisitos
necessários à obtenção do título de Mestre em
Engenharia Elétrica.

Orientador: José Manoel de Seixas

Rio de Janeiro
Março de 2016

INTELIGÊNCIA COMPUTACIONAL NA AVALIAÇÃO DE CÓDIGOS EM UM
SISTEMA COMPLEXO DE DETECÇÃO COM DESENVOLVIMENTO
COLABORATIVO

Andressa Andrea Sivolella Gomes

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO
ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE
ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE
JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A
OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA
ELÉTRICA.

Examinada por:

Prof. José Manoel de Seixas, D.Sc.

Prof. Aluízio Fausto Ribeiro Araújo, D.Sc.

Prof. Afonso de Bediaga e Hickman, D.Sc.

Prof. Luis Henrique Maciel Kosmalski Costa, D.Sc.

Pesquisadora Carmen Lúcia Lodi Maidantchik, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
MARÇO DE 2016

Gomes, Andressa Andrea Sivolella

Inteligência Computacional na Avaliação de Códigos em um Sistema Complexo de Detecção com Desenvolvimento Colaborativo /Andressa Andrea Sivolella Gomes. – Rio de Janeiro: UFRJ/COPPE, 2016.

XI, 46 p.: il.; 29, 7cm.

Orientador: José Manoel de Seixas

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia Elétrica, 2016.

Referências Bibliográficas: p. 40 – 46.

1. Mineração de códigos. 2. Métodos ensemble com árvores. 3. Plataforma colaborativa. I. Seixas, José Manoel de. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia Elétrica. III. Título.

Agradecimentos

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

**INTELIGÊNCIA COMPUTACIONAL NA AVALIAÇÃO DE CÓDIGOS EM UM
SISTEMA COMPLEXO DE DETECÇÃO COM DESENVOLVIMENTO
COLABORATIVO**

Andressa Andrea Sivolella Gomes

Março/2016

Orientador: José Manoel de Seixas

Programa: Engenharia Elétrica

Apresenta-se, nesta dissertação, uma Plataforma web colaborativa, projetada e desenvolvida em um ambiente altamente complexo relacionado a área de Física de Altas Energias. A referida plataforma tem como um de seus objetivos prover um ambiente para desenvolvimento e compartilhamento de códigos fonte, que são executados em máquinas separadas do servidor que hospeda a plataforma. Este objetivo foi projetado com a finalidade de não sobrecarregar o servidor principal. Mas, códigos fonte podem conter falhas e a sua não identificação prévia, como consequência, consome recursos computacionais sem necessidade. Neste contexto, esta dissertação tem como principal objetivo avaliar a possibilidade de aplicação de algoritmos de aprendizado de máquina para classificar códigos fonte que poderiam falhar durante o processo de execução. Para atingir tal objetivo, um estudo na área de inteligência computacional aplicada a códigos é realizado, assim como a aplicação de ferramentas que garantem qualidade no desenvolvimento de software na área de Física de Altas Energias.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

COMPUTATIONAL INTELLIGENCE FOR SOURCE CODE ASSERTION IN A
COMPLEX SYSTEM WITH A COLLABORATIVE DEVELOPMENT
ENVIRONMENT

Andressa Andrea Sivolella Gomes

March/2016

Advisor: José Manoel de Seixas

Department: Electrical Engineering

In this work it is presented a collaborative web platform, designed and developed in a highly complex environment related to High Energy Physics. That platform has as one of its goals to provide an environment for development and sharing source codes, that are executed on machines separated from the server hosting this platform. This goal was designed in order to not overload the main server. But source codes may contain flaws and its no prior identification, as a result, consume computer resources unnecessarily. In this context, the main goal of this thesis is to investigate the possibility of applying machine learning algorithms to classify source codes that might fail in an attempt to run, but without running them. To achieve this goal, a study in the area of Computational Intelligence applied to source codes is performed, as well as the application of tools that guarantee quality in software development on High Energy Physics area.

Sumário

Lista de Figuras	ix
Lista de Tabelas	xi
1 Introdução	1
1.1 Motivação	1
1.2 Objetivos	2
1.3 Organização do documento	2
2 A Física de Altas Energias	4
2.1 CERN e LHC	4
2.2 Experimento ATLAS	6
2.3 Calorímetro Hadronico de Telhas (TileCal)	7
2.3.1 Análise <i>Online</i> e <i>Offline</i>	8
2.3.2 A colaboração TileCal	9
3 Plataforma web Tile-in-ONE	10
3.1 Fluxo de dados	11
3.2 Novo desenvolvimento	13
4 Mineração de códigos fonte para identificação de falhas	15
4.1 Analisadores estáticos e Mineração de códigos no CERN	17
4.2 Avaliação de atributos	18
4.2.1 Analisadores Estáticos	19
4.2.2 Medidas estatísticas e de qualidade	20
4.3 Algoritmos de aprendizado conjunto utilizando Árvores de Decisão .	21
4.3.1 Bagged Trees (<i>Bootstrap Aggregating</i>)	25
4.3.2 Random Forest	25
4.3.3 Boosted Decision Trees (BDT)	26
5 Métodos e Resultados	28
5.1 Aquisição de códigos fonte	28

5.2	Seleção de atributos	30
5.3	Treinamento	32
5.4	Discussão de Resultados	35
6	Conclusões	39
	Referências Bibliográficas	40

Listas de Figuras

2.1	Representação aérea do LHC e seus detectores no CERN. Extraído de [1].	5
2.2	Esquema do detector ATLAS. Adaptado de [2]	6
2.3	Calorímetro de Telhas do ATLAS (com suas 4 partições na cor verde) envolvendo o calorímetro de argônio líquido. O Sistema Magnético e a Câmara de Múons não estão ilustrados nesta figura. Adaptado de [3].	7
2.4	Estrutura de absorção e amostragem do TileCal. Extraído de [4]	8
2.5	Esboço das células do TileCal. Extraído de [3].	8
3.1	<i>Dashboard Web System</i> , desenvolvido em 2010 para integrar as os dados adquiridos e as análises do grupo de QD do TileCal. Mais detalhes em [5]	11
3.2	Esquemático representando a infraestrutura da plataforma Tile-in-ONE. Em vermelho, a nova camada a ser desenvolvida.	13
3.3	Fluxo de dados da plataforma web Tile-in-ONE.	14
4.1	Estimativa do percentual de erros inseridos durante fases do SDLC. (Fonte: <i>Computer Finance Magazine</i> . Dados extraídos de [6])	16
4.2	Conjunto de dados e árvore de decisão gerada, respectivamente	22
4.3	Cálculo da entropia do atributo <i>tempo</i>	23
4.4	Critérios de divisão encontrados na literatura	24
4.5	Pseudo-código para <i>Bagged Trees</i> . Retirado de [7].	26
4.6	Ilustração do método BDT. Baseado em [8].	27
5.1	Total de códigos fonte escritos (512) entre Junho/2014 e Dezembro/2015.	28
5.2	Retornos obtidos após tentativa de executar códigos fonte em máquinas <i>slave</i>	29
5.3	Percentuais de alvos do conjunto de códigos fonte.	30
5.4	Matriz de correlação dos 11 atributos extraídos inicialmente	31
5.5	Matriz de correlação dos 6 atributos selecionados. As correlações com a saída também são calculadas.	32

5.6	Pseudo-código do algoritmo <i>AdaBoost</i>	34
5.7	Matrizes de Confusão dos 4 classificadores gerados.	36
5.9	Atributos mais relevantes de acordo com erro OOB	36
5.8	Medidas de desempenho dos 4 classificadores gerados.	37

Lista de Tabelas

4.1	Lista inicial de atributos	21
5.1	Lista de atributos selecionados	31
5.2	Resumo das medidas de desempenho dos 4 classificadores gerados . .	38

Capítulo 1

Introdução

No desenvolvimento de projetos de software, quanto mais cedo uma falha em código é identificada, menos custosa é a sua manutenção. Uma das maneiras de garantir qualidade está na utilização de analisadores estáticos: tal ferramenta gera alertas sem a necessidade de execução do código que está sendo desenvolvido. Tais alertas indicam linhas em que possivelmente o código irá falhar quando for executado. Mas, com o passar do tempo, um inconveniente do analisador estático foi empiricamente identificado: percebeu-se que a maior parte dos alertas gerados não indica necessariamente falha em tempo de execução e em contra partida, algumas falhas não são identificadas pelos alertas do analisador estático.

Neste contexto, uma aplicação que surge para auxiliar a área de engenharia de software na oferta de qualidade em desenvolvimento de código fonte é a área de Inteligência Computacional aplicada em códigos. As motivações são diversas: vão desde mineração em repositórios de versionamento para identificação do que determinado projeto faz sem nem ao menos executá-lo, até a identificação de trechos de códigos em grandes projetos que são muito similares (ou até mesmo iguais) e dificultam a sua manutenção.

Este projeto é, portanto, interdisciplinar e busca aplicar técnicas de Inteligência Computacional durante a implementação de códigos fonte.

1.1 Motivação

O CERN (Organização Européia de Pesquisa Nuclear) possui em operação o LHC (*Large Hadron Colider*), que é o maior colisor de partículas já construído na história. Ao longo do LHC foram construídos experimentos, que são pontos de observação onde as partículas aceleradas em direções opostas se colidem. O ATLAS é o experimento o qual este projeto está inserido. Sua colaboração é composta por milhares de pessoas e dezenas de países membros. O calorímetro de telhas (Tile-Cal) compõe o sistema de calorimetria do ATLAS juntamente com o calorímetro de

argônio líquido, e tem como objetivo medir a energia das partículas resultantes de colisões. Um dos objetivos da colaboração do TileCal é assegurar a qualidade de dados de calibração para garantir a sua operação. Uma característica marcante de projetos no CERN é a alta rotatividade de pesquisadores dentro do experimento. Um novo entrante no projeto ou a simples troca de funções exige um tempo de treinamento. Muitas são as tarefas que precisam ser realizadas pela equipe que analisa os resultados gerados, a fim de garantir a qualidade de dados do TileCal. A maior parte dessas tarefas é realizada com o auxílio de sistemas computacionais desenvolvidos ao longo da colaboração de diferentes institutos. Para facilitar a integração de tais ferramentas, a Plataforma web Tile-in-ONE foi desenvolvida e uma de suas principais características é fornecer um ambiente via web para o desenvolvimento de análises físicas através da implementação de códigos fonte. O Tile-in-ONE foi projetado para executar os códigos desenvolvidos e compartilhados pela colaboração em um conjunto de máquinas separadas do servidor para evitar sobrecarga de processamento ou acesso a dados. Atualmente a plataforma encontra-se em produção. Se um código fonte é submetido com falhas, consome-se tempo e recursos computacionais ao enviar o código para máquinas diferentes do servidor. Neste contexto, surge uma questão: seria possível identificar códigos fonte com potencial de falha sem necessariamente executá-los? O projeto aqui desenvolvido busca responder tal pergunta.

1.2 Objetivos

Como dito anteriormente, este projeto tem como objetivo principal identificar códigos fonte com falhas, desenvolvidos em um ambiente web colaborativo, antes de sua execução, evitando que recursos computacionais sejam utilizados sem necessidade. Algorítmos de aprendizado de máquina serão verificados como uma maneira de abordar o objetivo principal deste projeto. Esta avaliação pode se dar através de medidas e ferramentas frequentemente utilizadas na área de inteligência computacional (F1-score e matrizes de confusão, principalmente). A investigação de aplicação de inteligência computacional na área de Física de Altas Energias para este fim e a aplicação de ferramentas que garantem a qualidade de códigos desenvolvidos também é um dos objetivos deste projeto.

1.3 Organização do documento

No Capítulo 2 o ambinete do CERN, o colisor de partículas LHC e o experimento ATLAS são apresentados, com ênfase no Calorímetro de Telhas (TileCal) bem como suas análises e os diferentes grupos que compõem a colaboração TileCal.

O Capítulo 3 descreve a Plataforma web Tile-in-ONE projetada e desenvolvida, desde o fluxo de dados projetado, até mesmo a infraestrutura implantada. Ao ser colocado em produção uma nova necessidade surge e tal necessidade também é descrita neste capítulo.

Uma investigação na literatura sobre a aplicação de mineração de códigos para a identificação de falhas em códigos sem a sua execução é abordada no capítulo 4, principalmente no ambiente de Física de Altas Energias, e consequentemente, no CERN. Neste capítulo também são abordados os atributos comumente utilizados na aplicação de mineração em códigos. Os algoritmos que foram testados como uma possível abordagem o finalizam.

O Capítulo 5 descreve quais códigos fonte compõem o conjunto de dados que foi utilizado para o treinamento dos algoritmos descritos no capítulo anterior. Faz-se um estudo também da necessidade em se utilizar todos os atributos citados pela literatura. E por fim, o desempenho dos classificadores gerados são discutidos.

O Capítulo 6 finaliza este documento avaliando se os objetivos descritos na seção 1.2 foram contemplados e levanta ainda algumas possíveis análises que podem ser abordadas no futuro.

Capítulo 2

A Física de Altas Energias

A Física de Partículas estuda os constituintes da matéria e as interações entre elas. Esse estudo também é referenciado como Física de Altas Energias, devido à grande quantidade de energia necessária para a observação das estruturas básicas da matéria.

Este capítulo aborda o ambiente da física de partículas no CERN, bem como o colisor de partículas atualmente em operação, o LHC, com destaque para o experimento ATLAS e um de seus calorímetros (TileCal).

2.1 CERN e LHC

Fundado em 1954, o CERN (em francês *Centre Européen pour la Recherche Nucléaire* [9]) é o maior centro de pesquisa na área de física de partículas de altas energias no mundo. Atualmente, conta com a participação de 38 países membros e outros países colaboradores, entre eles o Brasil.

O LHC (em inglês *Large Hadron Collider* [10]) é o maior colisor de partículas já construído e encontra-se atualmente em operação no CERN. Instalado a 175 metros abaixo do solo, consiste em um grande túnel em formato anelar, com 27 km de perímetro. Quando partículas são eletricamente carregadas e submetidas a pulsos eletromagnéticos no interior de tubos tem-se o processo de aceleração de partículas. Neste processo, as partículas adquirem aceleração ao serem envolvidas a campos eletromagnéticos variantes [11]. Em aceleradores circulares, partículas são injetadas e circulam no anel até atingirem a energia desejada. Diversos experimentos podem ser realizados em determinados pontos de colisão ao longo da circunferência. O perímetro do acelerador circular é diretamente proporcional a energia necessária na colisão. Ao atingir a energia desejada, os feixes de partículas aceleradas em sentidos opostos colidem e, nesse caso, detectores em formato cilíndrico são os mais utilizados. Cada colisão gera informações que devem ser analisadas por especialistas, com objetivos distintos, tais como: identificação de sub-partículas, observação de fenômenos

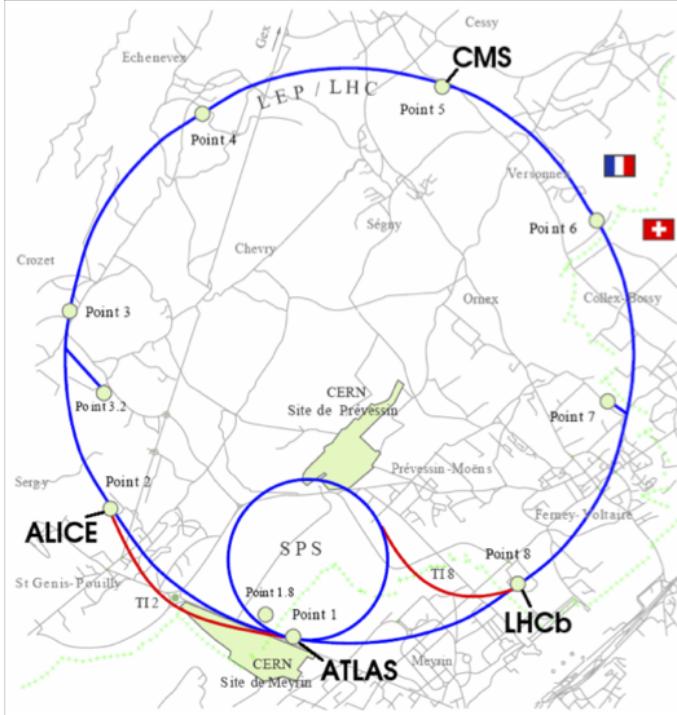


Figura 2.1: Representação aérea do LHC e seus detectores no CERN. Extraído de [1].

e a comprovação ou a eliminação de teorias físicas. Nos pontos de colisão, detectores são montados para observação e coleta de dados. Existem, por exemplo, detectores de calorimetria, para identificar a energia depositada pelas partículas colisionadas; detectores de traços, para observar trajetórias após colisões; e detectores de mísseis, responsáveis por absorver tais partículas.

Um dos objetivos do LHC (e também do ATLAS) é a comprovação da existência de uma nova partícula, o bóson de Higgs. Essa partícula, prevista no modelo padrão foi recentemente comprovada experimentalmente através do experimento em questão, laureando Peter Higgs e François Englert com o prêmio Nobel da Física na sua última edição, ocorrida em 2013 [12]. Segundo o modelo, o mecanismo de Higgs é o que dá massa a todas as partículas elementares, e o bóson de Higgs é uma partícula escalar maciça, que valida o modelo padrão. Sua detecção é difícil, uma vez que é necessária uma energia muito grande para sua observação. Ainda assim, por ser uma partícula muito energética, sua detecção se dá através de seus decaimentos, chamados de assinaturas do bóson de Higgs.

A figura 2.1 ilustra uma representação aérea do LHC. Existem quatro detectores de partículas instalados em pontos de colisão ao longo da extensão do LHC, altamente especializados: ALICE [13], ATLAS [3], CMS [14] e LHCb [15]. Os detectores ilustrados permitem obter informações detalhadas sobre a trajetória das partículas resultantes das colisões ocorridas e características energéticas, sendo possível iden-

tificar partículas. O ATLAS é o maior dos detectores do LHC.

2.2 Experimento ATLAS

O ATLAS (em inglês *A Toroidal LHC ApparatuS*) [3] é um detector em formato cilíndrico. O detector pesa aproximadamente 7.000 toneladas, com 44 metros de comprimento por 24 metros de altura. A figura 2.2 ilustra os diversos subsistemas que compõem o detector ATLAS.

Cada subsistema apresenta uma característica específica de acordo com suas funcionalidades. São eles, da parte interna para a externa:

- Detector Interno (ID), responsável por registrar as trajetórias das partículas resultantes da colisão, bem como o momento e o sinal da carga, se for o caso;
- Sistema de Calorimetria, composto pelos calorímetros eletromagnético (LAr) e hadrônico de telhas (TileCal). Os calorímetros citados são responsáveis por medirem a energia das partículas resultantes;
- Sistema Magnético, responsável por desviar partículas carregadas para medir o momento;
- o Espectrômetro de Múons, responsável por medir a trajetória de uma determinada partícula denominada Múon.

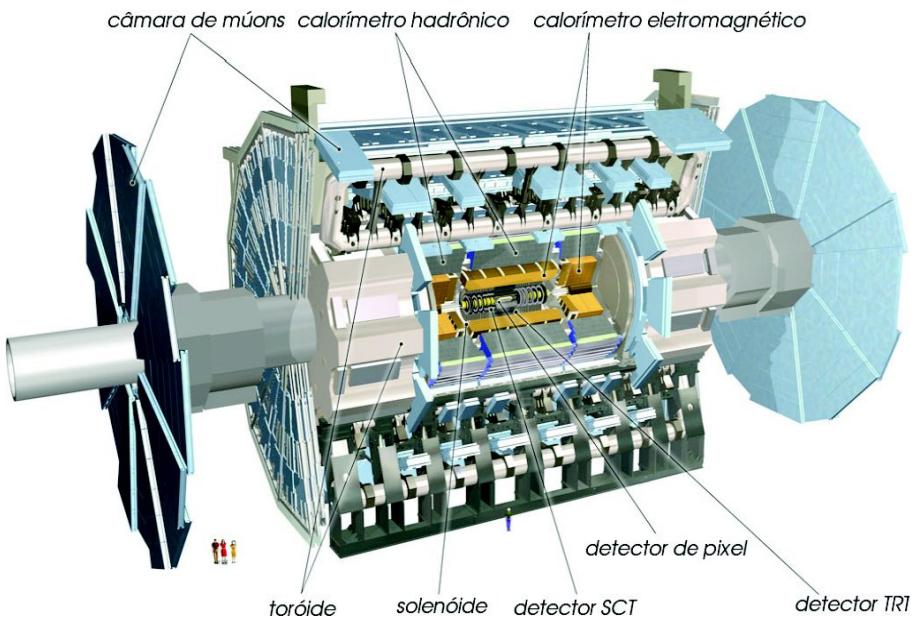


Figura 2.2: Esquema do detector ATLAS. Adaptado de [2]

O sistema de calorimetria do LHC foi projetado para absorver a energia das partículas que cruzam o detector, sendo o calorímetro hadrônico de telhas (em inglês *Tile Calorimeter* ou TileCal) é o foco deste mestrado.

2.3 Calorímetro Hadronico de Telhas (TileCal)

O TileCal [4] é composto por três barris: um central (que se divide em duas partições: LBA e LBC) e dois estendidos (cada um correspondendo a mais duas partições, EBA e EBC, respectivamente), como ilustrado na figura 2.3. Cada partição é igualmente dividida em 64 partes (módulos). Os módulos do TileCal são compostos de ferro como mediadores passivos e telhas cintilantes como material ativo. A estrutura absorvedora é coberta de placas de aço de várias dimensões, conectadas a uma estrutura macia de suporte. A principal inovação deste detector é a posição perpendicular das telhas em relação aos feixes do LHC.

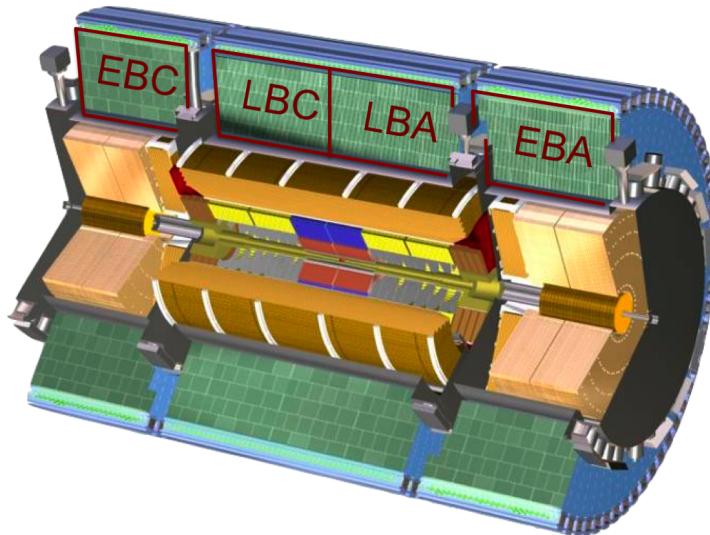


Figura 2.3: Calorímetro de Telhas do ATLAS (com suas 4 partições na cor verde) envolvendo o calorímetro de argônio líquido. O Sistema Magnético e a Câmara de Múons não estão ilustrados nesta figura. Adaptado de [3].

A passagem de partículas ionizadas pelo TileCal produz luz no espectro. Sua intensidade é proporcional à energia depositada pela partícula. A luz produzida se propaga através das telhas para suas bordas onde é absorvida por fibras óticas e deslocada, por reflexão interna total, até os canais de leitura eletrônica ou fotomultiplicadores (em inglês *PhotoMultipliers Tubes* ou PMTs), onde é convertido em um sinal de corrente. Um PMT pode ser idealizado como um pulso de corrente. A figura 2.4 ilustra a estrutura de absorção e amostragem do TileCal sendo possível identificar tubos fotomultiplicadores.

Os módulos do barril central do detector comportam 45 canais cada, enquanto os módulos dos barris estendidos comportam 32 canais. Radialmente, cada módulo do TileCal ainda é segmentado em três camadas (A, BC e D), conhecidas como células. Estas são formadas pelo agrupamento de fibras até cada PMT. Um esboço

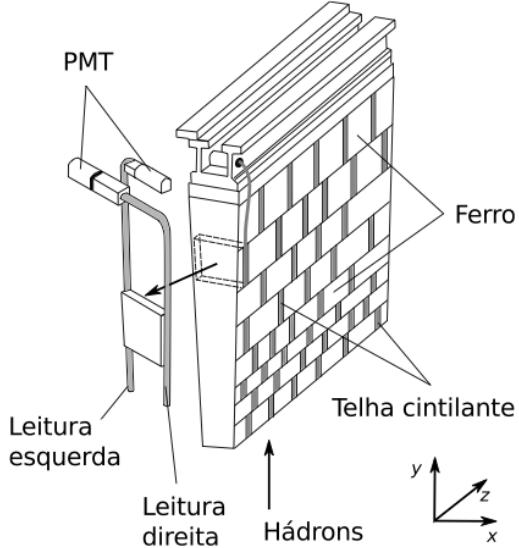


Figura 2.4: Estrutura de absorção e amostragem do TileCal. Extraído de [4]

da geometria das células é apresentado na figura 2.5. Cada módulo contém 11 linhas transversas de telhas.

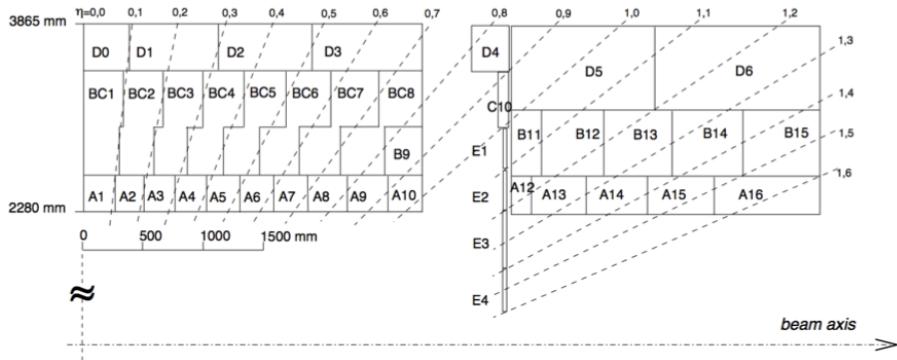


Figura 2.5: Esboço das células do TileCal. Extraído de [3].

2.3.1 Análise *Online* e *Offline*

Existem dois tipos de análises, comuns a todos os experimentos do LHC. O experimento ATLAS, sozinho, gera um volume de dados na ordem de 3,2 PetaBytes ao ano [16], que precisam ser processados e analisados. Nesse caso, é impraticável armazenar os dados adquiridos pelo experimento em sua totalidade para posterior análise. Por esse motivo, análises referentes a atividades anteriores ao armazenamento em mídia permanente são definidas como *online*. Já as análises que ocorrem a partir do acesso de informações em bases de dados são definidas como sendo *offline*.

As soluções abordadas neste mestrado estão contextualizadas no processo de análise *offline* do TileCal.

As análises *online* e *offline* serão descritas detalhadamente na seção 3.

2.3.2 A colaboração TileCal

O TileCal é composto ainda por diferentes grupos, responsáveis por atividades específicas no experimento. Cada grupo desempenha uma etapa no processo de trabalho do calorímetro. São eles:

DAQ (do inglês, *Data Acquisition* ou **aquisição de dados**) responsável pela aquisição dos sinais resultantes das interações físicas no detector;

DCS (do inglês, *Detector Control System*, ou **sistema de controle do detector**) responsável pela manutenção das fontes de alta e baixa tensão do TileCal, placas mãe e sistema de refrigeração;

DQ (do inglês, *Data Quality*, ou **qualidade de dados**) responsável por avaliar o funcionamento do detector garantindo confiabilidade nos dados adquiridos que serão analisados;

Calibração (do inglês, *Calibration*) responsável pela calibração das células do calorímetro e validação da escala eletromagnética, a fim de corrigir desvios que ocorrem ao longo do tempo devido a desgastes ocorridos pela exposição do detector a altos níveis de radiação;

Computação e Software responsável por realizar a organização e desenvolvimento da simulação, reconstrução, digitalização e desenvolvimento de filtros de alto nível;

Dados e Processamento (do inglês, *Data and Processing*) responsável por coletar informações dos grupos de qualidade de dados e calibração a fim de decidir que alterações nos dados de condições serão realizadas. Responsável também por analisar os dados físicos.

Existem, portanto, seis grupos no TileCal onde o compartilhamento de informações entre eles faz-se necessário: se um PMT não está sendo corretamente alimentado, a qualidade dos dados adquiridos pode ficar comprometida. Tal situação exige o compartilhamento de informações do grupo e DQ, por exemplo.

Capítulo 3

Plataforma web Tile-in-ONE

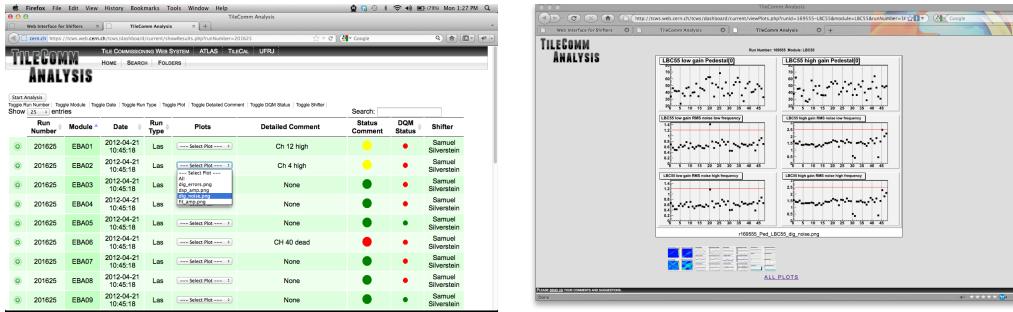
Diversos sistemas foram desenvolvidos ao longo das diferentes fases do experimento, cada um com um propósito específico [5]. A análise *online* tem início com a aquisição e decisão acerca do armazenamento dos dados adquiridos, que é responsabilidade do grupo TDAQ (do inglês, *Trigger and Data Acquisition*) [17]. A reconstrução dos dados pelo software para análise *offline* ATHENA [18] marca o início da análise *offline*. Em seguida, o DQMF (do inglês, *Data Quality Monitoring System*) [19] é aplicado e gera automaticamente estados para os PMTs, indicando a qualidade do tubo fotomultiplicador e reduzindo a quantidade de canais que precisam ser analisados. Nesta etapa, gráficos são gerados para auxiliar o grupo de qualidade de dados do TileCal em sua análise *offline*. Um sistema web foi desenvolvido para integrar os resultados gráficos gerados durante a etapa de reconstrução, guiando o grupo de qualidade de dados em suas análises. A figura 3.1 ilustra sistemas integrados por uma única interface web: a figura 3.1(a) corresponde a lista de dados reconstruídos e quais gráficos já estão disponíveis para a colaboração, indicando que a atuação do ATHENA foi concluída; a figura 3.1(b) exibe informação detalhada sobre um determinado módulo, acessado pela interface ilustrada anteriormente. Através deste sistema web é possível inserir comentários relacionados a performance do detector; e a figura 3.1(c) ilustra alguns gráficos gerados durante a etapa de reconstrução dos dados.

Para uma referência histórica, a lista de canais definidos como defeituosos pela colaboração (*Bad Channels List*) é armazenada no banco de dados de condicionamento, comum a todos os experimentos que compõem o LHC, o COOL DB [20]. Os sistemas MCWS (do inglês, *Monitoring & Calibration Web System*) [21] e DCS (do inglês, *Detector Control System*) [22] apoiam o grupo de qualidade de dados na análise dos quase 10.000 PMTs do calorímetro de telhas. O MCWS foi desenvolvido para exibir a lista de canais defeituosos de maneira gráfica e permite que comentários sobre o estado do detector sejam compartilhados. O sistema web DCS foi desenvolvido para monitorar as fontes de alta tensão que alimentam os PMTs.

The screenshot shows a Firefox browser window with the title "Web Interface for Shifters". The URL is https://rcws.web.cern.ch/rcws/dashboard/current/showRunList.php. The page has a header with "WIS" logo and navigation links for "HOME" and "TIMELINE". Below the header is a table titled "Overview" with columns: Run Number, Run Type, Date, # of Events, OK, Some Problems, Bad, Not to be Analyzed, Not Analyzed, # of Modules, Plots, and Shifter. The table lists several runs from 2012-04-21 to 2012-04-19, with various status codes and module counts.

Run Number	Run Type	Date	# of Events	OK	Some Problems	Bad	Not to be Analyzed	Not Analyzed	# of Modules	Plots	Shifter
201650	RampCIS	2012-04-21 12:01:49	85461	0.00%	0.00%	0.00%	0.00%	100.00%	120		
201645	RampCIS	2012-04-21 11:44:59	83310	0.00%	0.00%	0.00%	0.00%	100.00%	128		
201632	Ped	2012-04-21 11:13:12	100119	0.00%	0.00%	0.00%	0.00%	100.00%	256		
201626	Las	2012-04-21 10:48:03	100043	0.00%	0.00%	0.00%	0.00%	100.00%	256		
201625	Las	2012-04-21 10:45:18	10008	73.05%	7.42%	0.00%	0.00%	256	Samuel Silverstein		
201619	Ped	2012-04-21 10:33:51	10213	0.00%	0.00%	0.00%	0.00%	100.00%	256		
201617	CIS	2012-04-21 10:30:20	4932	0.00%	0.00%	0.00%	0.00%	100.00%	256		
201616	MonoCIS	2012-04-21 10:27:47	10028	0.00%	0.00%	0.00%	0.00%	100.00%	256		
201583	Las	2012-04-20 15:41:24	100168	0.00%	0.00%	0.00%	0.00%	100.00%	256		
201582	Las	2012-04-20 15:38:30	10245	0.00%	0.00%	0.00%	0.00%	100.00%	256		
201572	Ped	2012-04-20 15:28:48	10178	0.00%	0.00%	0.00%	0.00%	100.00%	256		
201571	CIS	2012-04-20 15:24:51	4932	0.00%	0.00%	0.00%	0.00%	100.00%	256		
201569	MonoCIS	2012-04-20 15:22:39	10155	0.00%	0.00%	0.00%	0.00%	100.00%	256		
201500	Las	2012-04-19 10:30:10	100027	0.00%	0.00%	0.00%	0.00%	100.00%	256		
201499	Las	2012-04-19 10:27:04	10120	0.00%	0.00%	0.00%	0.00%	100.00%	256		

(a) Lista dos dados adquiridos.



(b) Estados gerados automaticamente pelo (c) Gráficos gerados para um determinado DQMF e acesso a informações detalhadas. módulo.

Figura 3.1: *Dashboard Web System*, desenvolvido em 2010 para integrar os dados adquiridos e as análises do grupo de QD do TileCal. Mais detalhes em [5]

É evidente que o cenário descrito é composto por diversas ferramentas (web) para apoiar em diferentes aspectos as análises e operação de forma mais eficiente do TileCal. Tais sistemas foram desenvolvidos em diferentes fases do detector, envolvendo diferentes colaboradores, que não necessariamente ainda encontram-se envolvidos em atividades da colaboração. Além disso, a manutenção de tais ferramentas não é garantida pelos mesmos colaboradores contribuintes. Para a colaboração, seria mais fácil reunir todas as ferramentas existentes em uma única interface, que utilizasse preferencialmente a mesma tecnologia, permitindo inclusive, reutilização de códigos fonte. Tais requisitos foram contemplados com o projeto e desenvolvimento da Plataforma web Tile-in-ONE.

3.1 Fluxo de dados

A Plataforma Tile-in-ONE foi projetada para integrar as diferentes análises realizadas pela colaboração TileCal. Ao fornecer uma estrutura onde os colaboradores podem desenvolver códigos fonte diretamente através da web, ela integra diferen-

tes ferramentas em um único lugar. Isso garante a escolha da mesma tecnologia para todas as ferramentas desenvolvidas (linguagem Python [23]), o que torna a manutenção da plataforma mais fácil. Além disso, existe o incentivo natural para reutilização de códigos, uma vez que os desenvolvimentos encontram-se disponíveis para toda colaboração. Outra funcionalidade oferecida pela plataforma é o encapsulamento em pacotes de configurações para acessar bases de dados importantes para as análises. Isso permite que colaboradores novos não percam tempo aprendendo a acessar a informação, mas foquem na análise propriamente dita.

A figura 3.3 ilustra o fluxo de dados do Tile-in-ONE. A plataforma oferece a funcionalidade de desenvolvimento de códigos pela qual os colaboradores podem escrever seus próprios programas. Também é permitida a edição de códigos salvos ou escritos por outros usuários. Uma vez que o desenvolvimento do código é concluído, o desenvolvedor pode submeter seu código que será manipulado no lado do servidor. Este, por sua vez, irá direcionar o código em questão para uma outra máquina (nesse contexto, máquinas *slaves*), dependendo dos dados identificados pela plataforma que o código fonte a ser executado deseja acessar. Cada máquina *slave* está configurada para acessar um determinado repositório de dados utilizado para as análises da colaboração. Isso permite que os desenvolvedores se abstraiam de configurações de acesso a tais repositórios. A motivação para efetuar a execução do código fonte em outra máquina que não no servidor é evitar que ocorra sobrecarga no servidor principal e o mais importante: caso o código fonte falhe durante a execução, o servidor onde a plataforma está hospedado não fica comprometido. A figura 3.3 ilustra ainda, em vermelho, onde esta tese de mestrado irá atuar. O objetivo é criar uma etapa antes da submissão de novos códigos para máquinas *slave*, atuando, desta forma, como uma camada de segurança cujo intuito é prever quais códigos vão falhar sem a necessidade de executá-los. Esta camada consiste em um novo desenvolvimento e portanto, no momento, qualquer código pode ser submetido para ser executado na máquina *slave*.

Uma vez que o servidor web escolhe uma máquina *slave* esta tenta executar o código fonte submetido e retorna para o servidor web, de maneira estruturada, o que ocorreu durante a execução. Neste momento, o desenvolvedor do código fonte pode utilizar objetos gráficos também fornecidos pela plataforma para exibir os resultados recuperados através de seus códigos fonte. Enfim, o código fonte, a resposta obtida com a execução na máquina *slave* e os objetos gráficos são encapsulados em *Plugins* e disponibilizados em *Dashboards* para toda colaboração.

A figura 3.2 ilustra a infraestrutura atual da plataforma web Tile-in-ONE. Destaca-se em vermelho onde este mestrado pretende atuar.

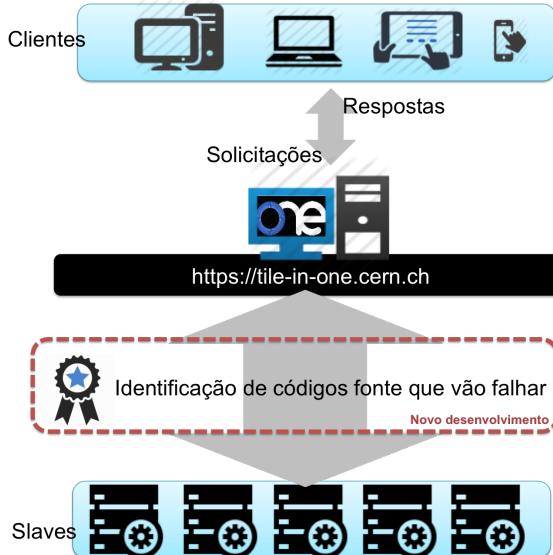


Figura 3.2: Esquemático representando a infraestrutura da plataforma Tile-in-ONE. Em vermelho, a nova camada a ser desenvolvida.

3.2 Novo desenvolvimento

A plataforma Tile-in-ONE está atualmente sendo utilizada pelos grupos de calibração e qualidade de dados do TileCal. Uma preocupação que surgiu ao longo do ano de 2014 foi a execução bem sucedida de códigos fonte sem avaliação prévia. Cada código que enfrenta falhas durante sua execução na máquina *slave* acaba consumindo recursos computacionais que podem comprometer o fluxo de dados da plataforma como um todo. A solução para tal questão levantada é estudada neste mestrado, fazendo uso de técnicas de mineração de códigos fonte, como abordado no capítulo 4.

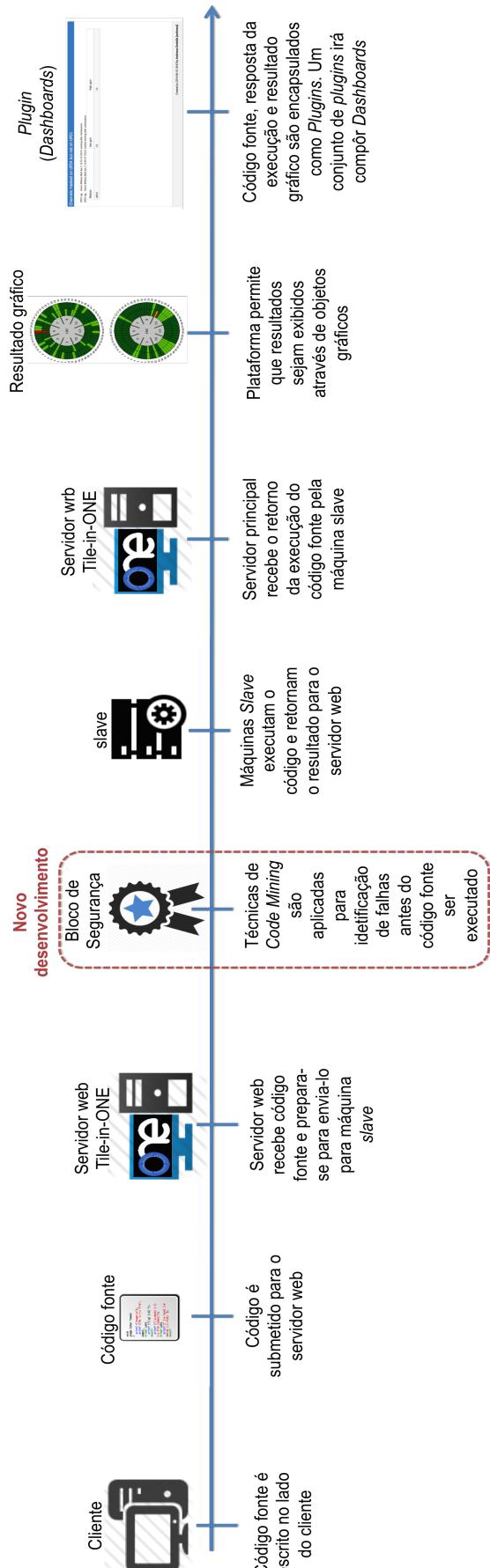


Figura 3.3: Fluxo de dados da plataforma web Tile-in-ONE.

Capítulo 4

Mineração de códigos fonte para identificação de falhas

Em desenvolvimento de software é comum observar que quanto maior for o tempo de permanência de um erro no código, mais custosa torna-se a sua correção e manutenção em um projeto [24]. Falhas em códigos fonte são definidas como “Imperfeições durante o desenvolvimento de software que, como consequência, impedem que o programa atinja as expectativas desejadas.” [25]. Entende-se ainda que muitos defeitos são introduzidos durante o processo de desenvolvimento como um todo, não somente no seu início ou fim. Assim sendo, identificar falhas antecipadamente torna-se parte essencial em desenvolvimento de software, que significa qualidade na entrega final do produto. [26].

O SDLC (do inglês, *software development life cycle* ou ciclo de vida do desenvolvimento de software) [27] indica que o desenvolvimento de software pode ser descrito em cinco etapas:

1. Levantamento de Requisitos
2. Projeto
3. Implementação
4. Teste
5. Implantação

A figura 4.1 ilustra em quais etapas do ciclo de vida de desenvolvimento do software são inseridos erros. Empresas que são referência na área de engenharia de computação (tais como a IBM [28] e HCL [29]) confirmam que cerca de 35% das falhas identificadas em códigos fonte são inseridas na fase de implementação [6].

Atualmente, diferentes técnicas têm sido utilizadas para a identificação antecipada de erros em códigos fonte. A análise estática, por exemplo, é bastante utilizada

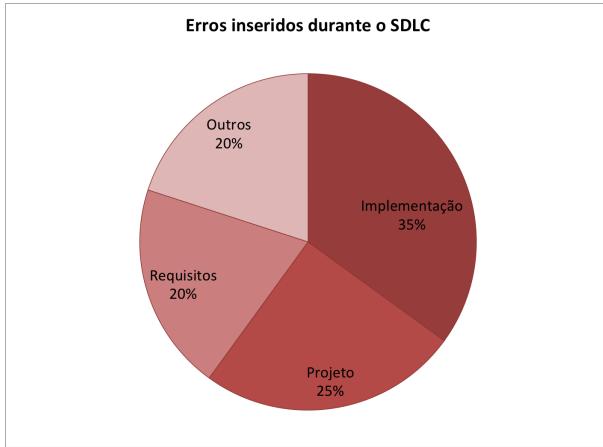


Figura 4.1: Estimativa do percentual de erros inseridos durante fases do SDLC.
(Fonte: *Computer Finance Magazine*. Dados extraídos de [6])

na área de programação desde a década de 80. As características desta análise são abordadas na seção 4.2.1. Em conjunto com analisadores estáticos, pode-se aplicar técnicas de mineração de dados, fazendo uma combinação entre a área de engenharia de software e inteligência computacional. No artigo [30] os autores abordam como a mineração de dados pode contribuir com a área de engenharia de software, a medida que técnicas de inteligência computacional, quando aplicadas, melhoram a qualidade de projetos. Nesta mesma direção, a MSR (em inglês, *Mining Software Repositories* ou mineração de repositórios de códigos) passa a ser amplamente abordada devido a disponibilidade gratuita de inúmeros repositórios públicos de controle de versionamento, rastreamento de erros inseridos e até mesmo documentações [31].

Aplicar mineração de dados em códigos fonte desperta interesse na comunidade científica, uma vez que códigos são tipicamente estruturados e semanticamente ricos em termos de construtores de programação (tais como variáveis, funções, objetos estruturados e rotinas bem definidas), diferente de textos livres, onde existe uma complexidade adicional devido ao tratamento de linguagem natural. Os objetivos são diversos: vão desde a tentativa de descobrir o que um determinado software faz (sem necessariamente executá-lo) até a manutenção de códigos ou análises de desenvolvimento de um projeto [32].

Como citado, a identificação antecipada de erros inseridos durante a implementação de códigos fonte é atraente do ponto de vista de qualidade de produto. A utilização de mineração de códigos é uma das aplicações mais ativas em engenharia de software, atualmente. Neste caso, o objetivo é obter ferramentas que não só identifiquem falhas (ou *bugs*) mas também sua localização exata nas linhas de código do projeto, de tal forma que o seu conserto seja facilitado [32]. Para alcançar esse objetivo, uma direção de bastante destaque consiste na mineração de padrões, que uma vez pré-definidos, são aplicados em diversos repositórios de projetos já existentes,

com o objetivo de encontrar anomalias comuns que violam regras ([33], [34], [35]).

Outro objetivo dominante na mineração de códigos é a tentativa de identificar fragmentos clonados. A não reutilização de códigos dificulta a manutenção do software e ainda, pode proliferar *bugs* em diferentes partes de um projeto.

A seção 4.1 investiga a aplicação de mineração de dados em códigos fonte no CERN. A seção 4.2 aborda quais são os atributos comumente utilizados quando a mineração de dados é aplicada em códigos fonte. Esta seção é composta por duas sub-seções (4.2.1 e 4.2.2), que abordam em que situações tais atributos foram utilizados na aplicação de mineração de códigos neste contexto; Por fim, a seção 4.3 descreve os algoritmos de aprendizado de máquina que foram utilizados no trabalho aqui desenvolvido, na tentativa de classificar os códigos fonte em *executáveis/não executáveis*.

4.1 Analisadores estáticos e Mineração de códigos no CERN

O CERN oferece um ambiente para desenvolvimento de software bastante particular. Estima-se que centenas de projetos de software estão sendo desenvolvidos, com propósitos específicos, desde programas que auxiliam o trabalho da equipe de recursos humanos até os que monitoram o desempenho de complexos experimentos que se desenvolvem no CERN. Apesar do setor de TI do CERN ter estipulado algumas regras para a política de segurança [36], muitos projetos de software pequenos não as seguem a risca. Isso implica em vulnerabilidades dos programas utilizados em ambientes de produção no CERN.

As regras de segurança estipuladas não são tão rígidas para estimular o meio acadêmico no avanço das pesquisas científicas. Da mesma maneira, a utilização de analisadores estáticos para assegurar a qualidade no desenvolvimento de códigos livres não são impostos para a comunidade. Existem analisadores estáticos disponíveis gratuitamente e com documentação e suporte oferecidos pela equipe de TI do CERN, mas nenhum projeto é obrigado a utilizá-los. No caso deste projeto, a linguagem computacional escolhida foi Python, sendo o *PyLint* [37] uma ferramenta para análise estática cuja utilização é indicada pelo setor de TI do CERN [38].

Recentemente, o setor de segurança em computação do CERN contratou um serviço (*Coverity* [39]) para garantir qualidade no desenvolvimento de novas funcionalidades no seu principal produto de análises físicas, o ROOT [40]. O ROOT é utilizado por cerca de 10.000 pesquisadores e a integridade do software passou a ser vista como algo valioso. Um *bug* perpetrado no ROOT pode ter um impacto muito negativo nas análises de resultados dos dados gerados pelo LHC. Assim que

incorporado, o *Coverity* já foi capaz de identificar mais de 40.000 falhas em aproximadamente 50 milhões de linhas de códigos [41]. Um ponto negativo observado por desenvolvedores do ROOT é o tempo necessário para executar o analisador estático: são 28 horas para cada 2 milhões de linhas de código. É válido observar que a ferramenta escolhida para o ROOT não atende as necessidades do projeto aqui desenvolvido: o *Coverity* atende códigos escritos em linguagem C/C++, enquanto os códigos do Tile-in-ONE (seção 3) são implementados em Python. Além disso, deseja-se aplicar mineração de dados em códigos fonte em um ambiente web, colaborativo. Portanto, não existe um repositório de versionamento.

Em física de altas energias não existe literatura sobre a aplicação de mineração de dados em códigos fonte.

4.2 Avaliação de atributos

Uma revisão feita por Heckman [42], com 21 artigos sobre mineração de dados em códigos fonte, identifica cinco categorias comumente utilizadas como atributos:

1. *Alert Characteristics* (AC): abordagem que leva em consideração os alertas gerados por analisadores estáticos (sub-seção 4.2.1);
2. *Code Characteristics* (CC): abordagem que considera medidas estatísticas e de qualidade do código fonte como atributos de entrada para algoritmos de aprendizado de máquina(sub-seção 4.2.2);
3. *Source code repository metrics* (SCR): atributos retirados de repositórios de códigos. Nesta abordagem, leva-se em consideração versionamentos (histórico de *bugs* inseridos no projeto), comentários de atualizações e documentação sobre requisitos, por exemplo;
4. *Bug database metrics* (BDM): neste caso, existe uma base de dados para o projeto contendo todas as falhas já surgidas e consertadas. O objetivo dessa abordagem é identificar padrões que podem voltar a acontecer no projeto;
5. *Dynamic analyses metric* (DA): esta abordagem armazena resultados obtidos com análise dinâmica de códigos, mediante diversas execuções (ver seção 4.2.1).

Tais categorias são descritas detalhadamente em [43].

Neste trabalho, utilizam-se as duas primeiras abordagens, as quais são descritas nas próximas sub-seções.

4.2.1 Analisadores Estáticos

Em engenharia de software, existem dois tipos de análise (complementares) que podem ser feitas para validar e testar o produto em concepção [44]:

- **Estática:** neste caso, apenas a estrutura do código é analisada, mas o código não é executado;
- **Dinâmica:** para esta análise, é preciso levantar um plano de testes, executá-lo e avaliar os resultados. Ou seja, é necessário executar o código diversas vezes;

O objetivo do trabalho aqui exposto é analisar códigos fonte sem executá-los; desta forma, utilizar analisadores estáticos torna-se atraente.

Na década de 80, diversas ferramentas que aplicam análise estática de código começaram a ser desenvolvidas para auxiliar desenvolvedores na implementação de códigos com menos *bugs* [45]. Mas, logo um inconveniente surge [46]: a quantidade de alertas gerados pelas ferramentas de análise estática de código (SCAT, do inglês, *Static Code Analysis Tools*) é grande e muitos deles não são acionáveis. Diz-se acionável um alerta que realmente vai impedir o sucesso na execução de um determinado código [42]. Essa classificação é feita manualmente pelo desenvolvedor (que, neste caso, é o especialista): se o especialista entende que determinado alerta é importante e precisa ser consertado antes da execução do programa, tal alerta é definido como *acionável*. Caso contrário, ele é definido como *não acionável* ([47], [48]).

Empiricamente, existem cerca de 40 alertas para cada mil linhas de código (KLOC, do inglês, *Kilo Lines of Code*) [48]. Desta estimativa, podem ser não acionáveis 30-100%, como observado em [49], [50], [48], [51], [52], [53]. Tais números desestimulam desenvolvedores e projetistas a aplicar ferramentas de análise estática de código no processo de desenvolvimento de software, embora entenda-se a sua importância no quesito qualidade adquirida. Por outro lado, um alerta definido pelo especialista como *acionável* nem sempre vai impedir a execução bem sucedida do código. Ou seja, se, por um lado, os analisadores estáticos geram muitos alertas (*não acionáveis*), nada garante que o alerta *acionável* indica que o programa contém *bugs* e vai falhar durante a tentativa de execução. Em outras palavras, um analisador estático pode não gerar nenhum alerta, e mesmo assim, durante a tentativa de execução, o código pode falhar.

Neste trabalho, um dos atributos utilizados corresponde à geração ou não de alertas pelo analisador estático. Adicionalmente, outras medidas estatísticas e de qualidade, descritas a seguir, compõem a lista inicial de atributos extraídos dos códigos fonte (tabela 4.1).

4.2.2 Medidas estatísticas e de qualidade

Em computação, “Halstead Metrics” [54] são medidas de qualidade e estatísticas definidas por Maurice Halstead e publicadas em 1977.

As medidas estatísticas (de Halstead) que podem ser extraídas de um código fonte são:

- η_1 : quantidade de operadores distintos no código;
- η_2 : quantidade de operandos distintos no código;
- N_1 : total de operadores;
- N_2 : total de operandos;

Destas, outras medidas estatísticas são também definidas por Halstead:

- Vocabulário: $\eta = \eta_1 + \eta_2$;
- Tamanho: $N = N_1 + N_2$;
- Volume: $V = N \log_2 \eta$;
- Dificuldade: $D = \frac{\eta_1}{2} \cdot \frac{N_2}{\eta_2}$;
- Esforço: $E = D \cdot V$;
- Estimativa de tempo de execução (em segundos): $T = \frac{E}{18}$;
- Estimativa de potenciais bugs: $B = \frac{V}{3000}$;

Outra característica que pode ser extraída de códigos fonte é a sua complexidade. Define-se como “Cyclomatic Complexity” (do inglês, complexidade ciclomática) [55] o número de decisões que um determinado bloco de código contém mais uma unidade. Este número (também conhecido como *McCabe number*) representa o número de caminhos independentes percorridos durante a execução de um código, quando se monta uma árvore abstrata de sintaxe (AST, do inglês, *abstract syntax tree*) [56].

O índice de manutenabilidade (ou MI, do inglês *Maintainability Index*) é uma medida em software que indica o quanto fácil é a manutenção de um determinado código. O MI é calculado de diferentes maneiras, dependendo da abordagem que se deseja utilizar. Entretanto, a fórmula clássica [57] (descrita na equação [I]) envolve o número de linhas totais em um código fonte (SLOC, do inglês, *source lines of code*), a complexidade ciclomática (neste contexto, CC) e o volume de Halstead (HV):

$$MI = 171 - 5.2 \ln_{HV} - 0.23_{CC} - 16.2 \ln_{SLOC} \quad [I]$$

Tabela 4.1: Lista inicial de atributos

#	Atributo	Descrição
1	Complexidade Ciclomática	Número de declarações
2	Índice de Manutenibilidade	Corresponde à organização do código
3	Vocabulário de Hasllead	# de operadores + # de operandos únicos
4	Tamanho de Halstead	# de operadores + # total de operandos
5	SLOC	Número de linhas do código fonte
6	Volume de Halstead	Combinação entre número de linhas e declarações
7	Dificuldade de Hasllead	Complexidade do código baseada no número de operadores e operandos
8	Tempo de Halstead	Tempo estimado para compilar um código
9	Bugs de Halstead	Possibilidade de bugs serem gerados
10	LLOC	Número de linhas lógicas
11	Alertas PyLint	Verdadeiro se PyLint gerou alertas. Caso contrário, falso

Os autores chegaram nesta fórmula através de um número de códigos fornecidos pela HP, escritos em linguagens de programação C e Pascal. Para cada código, um especialista (engenheiro de software) atribuiu uma nota (de 1 a 100) indicando o quanto fácil seria fazer alterações ou correções posteriores ao determinado código (quanto maior a nota, mais fácil é sua manutenção). Consequentemente, 40 faixas de valores foram identificadas e através de uma regressão estatística, a fórmula exposta foi encontrada como um índice de manutenibilidade. Existe ainda o LLOC (do inglês, *Logic Lines of Code*), que indica a quantidade de linhas com operadores lógicos no código que é outra medida para a extração como atributo.

A tabela 4.1 lista os atributos inicialmente extraídos com breve descrição. Uma análise de correlação foi realizada com o objetivo de reduzir o número de atributos utilizados. Os resultados desta análise são discutidos na seção 5.2. A existência de atributos com correlação alta é redundante para o treinamento de algoritmos de aprendizado de máquina.

4.3 Algoritmos de aprendizado conjunto utilizando Árvores de Decisão

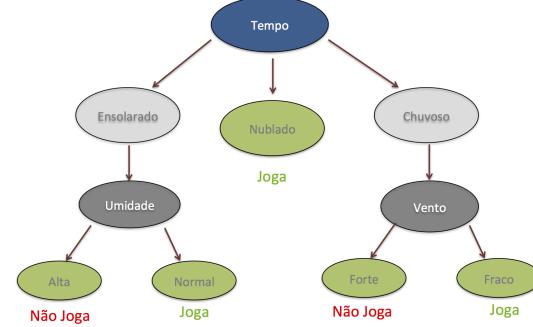
Árvore de decisão (DT, do inglês, *Decision Tree*) [7] é um algoritmo de aprendizado de máquina supervisionado. Desta maneira, o classificador aprende com casos em que se sabe a resposta (no caso deste projeto, *executável/não executável*).

Basicamente, o treinamento de uma árvore de decisão divide e agrupa o conjunto de treino em dois ou mais grupos homogêneos, sucessivamente. Para as divisões, o algoritmo se encarrega de escolher qual é o atributo mais significativo, ou seja, qual atributo irá tornar a divisão a mais homogênea (ou pura) possível. Tal processo se conhece como *splitting* e será descrito mais adiante nesta seção.

A figura 4.2(b) ilustra um exemplo clássico de árvore de decisão. Ele representa uma árvore que decide se uma pessoa vai ou não jogar tênis. Os atributos utilizados são condições do tempo (ensolarado, nublado ou chuvoso), umidade (alta ou baixa) e vento (forte ou fraco).

# Dia	Tempo	Umidade	Vento	Joga?
1	Ensolarado	Alta	Fraco	Não
2	Ensolarado	Alta	Forte	Não
3	Nublado	Alta	Fraco	Sim
4	Chuvoso	Alta	Fraco	Sim
5	Chuvoso	Normal	Fraco	Sim
6	Chuvoso	Normal	Forte	Não
7	Nublado	Normal	Forte	Sim
8	Ensolarado	Alta	Fraco	Não
9	Ensolarado	Normal	Fraco	Sim
10	Chuvoso	Normal	Fraco	Sim
11	Ensolarado	Normal	Forte	Sim
12	Nublado	Alta	Forte	Sim
13	Nublado	Normal	Fraco	Sim
14	Chuvoso	Alta	Forte	Não

(a) Conjunto de dados utilizados para gerar a árvore de decisão ilustrada ao lado



(b) Exemplo clássico de árvore de decisão

Figura 4.2: Conjunto de dados e árvore de decisão gerada, respectivamente

Ainda sobre a figura 4.2(b), o conhecimento da nomenclatura é necessário:

- O nó raiz (em azul) representa todo o conjunto de treino e é onde as divisões em grupos homogêneos começam.
- Os nós que continuam dividindo os sub-conjuntos restantes (em cinza) são denominados nós de decisão.
- Quando a divisão cessa, tem-se as folhas (em verde). As folhas indicam a decisão tomada pela árvore treinada.
- O tamanho da árvore indica quantas divisões foram efetuadas até a obtenção das folhas. No caso da figura 4.2(b) o tamanho da árvore é 3. Este parâmetro pode ser utilizado como critério de parada durante o treinamento do classificador, e assim sendo, não há garantias de obtenção de folhas puras.

Existem duas operações relacionadas ao treinamento de árvores de decisão. Elas estão descritas a seguir.

Splitting

Os critérios de divisão consideram quais atributos permitirão que a divisão do conjunto de dados obtida seja a mais pura possível. Desta forma, a escolha dos atributos que serão utilizados para a divisão em sub-conjuntos é feita através de medidas de impureza. Existem duas medidas principais utilizadas por diversos critérios de divisão, de acordo com [7]:

- Entropia (da informação), segundo Shannon [58] identifica o grau de incerteza de uma informação. No contexto deste projeto, quanto menor for a entropia da informação, mais puro é um nó. A entropia é definida de acordo com a fórmula abaixo:

$$H(X) = - \sum_{i=1}^k p_i \log_2(p_i) \text{ [II]}$$

Onde k é a quantidade de respostas de um determinado atributo (no caso do atributo *tempo*, as possíveis respostas seriam *ensolarado*, *nublado* ou *chuvisco* e $k = 3$) e p_i é a probabilidade da resposta i ocorrer dado o conjunto de dados que será dividido. A figura 4.3 ilustra o cálculo da entropia para o atributo *tempo* como uma medida de impureza. Como dito, quanto menor o valor da entropia, mais puro é um nó. A figura 4.2(a) contém o conjunto de dados utilizado.

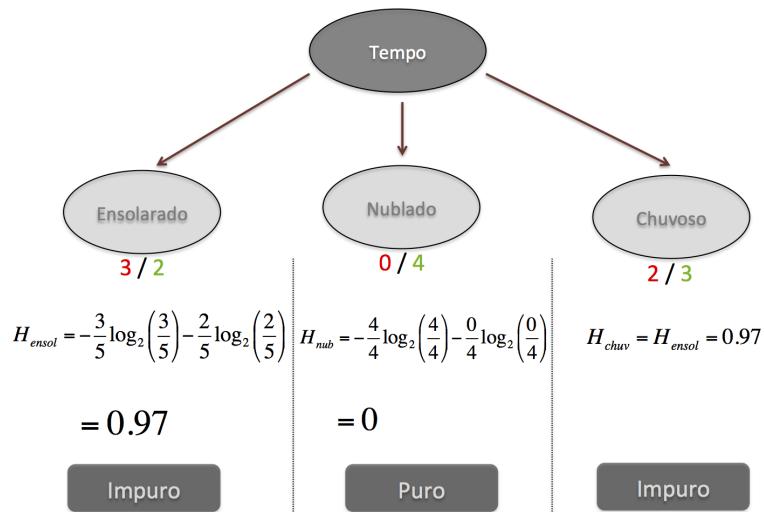


Figura 4.3: Cálculo da entropia do atributo *tempo*.

- Gini é outra medida de impureza, frequentemente utilizada por seu cálculo ser computacionalmente mais rápido que o cálculo da entropia [7]. A fórmula a seguir define o Gini do alvo Y:

$$Gini(Y, S) = 1 - \sum_{c_j \in \text{dom}(a_i)} \left(\frac{\sigma_{y=c_j} S}{|S|} \right)^2 \quad [\text{III}]$$

Lê-se, *Gini do alvo* (onde Y representa o alvo) e S é o número total de amostras no conjunto de dados (no caso, 14). O numerador do somatório ($\sigma_{y=c_j} S$) indica a frequência relativa da classe c_j no conjunto S . Para que a definição do cálculo do *Gini do alvo* fique mais clara, segue a fórmula abaixo:

$$Gini(Y, S) = 1 - [(\frac{5}{14})^2 + (\frac{9}{14})^2] = 0,46 \quad [\text{IV}]$$

O denominador igual a 14 em cada uma das frações indica o número total de amostras. No caso, das 14 amostras, 5 possuem alvo *não joga* e 9 possuem alvo *joga*, como pode ser verificado na figura 4.2(a). E assim, o *Gini do alvo* é igual a 0,46. Quanto menor o valor do Gini, mais puro é o conjunto.

A literatura, comumente utiliza dois critérios a partir das medidas de impureza descritas:

- Ganho de informação, que utiliza a entropia como medida de impureza. A figura 4.4(a) ilustra como o ganho de informação (ou, em inglês, *InfoGain*) é calculado.
- Índice Gini, que utiliza o Gini como medida de impureza. A figura 4.4(b) ilustra como o índice Gini é calculado.

$$\text{InfoGain}(X, S) = H(Y, S) - \sum_{i=1}^n H(\sigma_i) \frac{|\sigma_i|}{|S|} \quad \text{(a) Fórmula para cálculo do Ganho de Informação (ou Info Gain)}$$

$$GiniIndex = Gini(Y, S) - \sum_{v_{i,j} \in \text{dom}(a_i)} \left(\frac{|\sigma_{a_i=v_{i,j}} S|}{|S|} \right) Gini(X, |\sigma_{a_i}|) \quad \text{(b) Fórmula para o cálculo do Índice Gini (ou GiniIndex)}$$

Figura 4.4: Critérios de divisão encontrados na literatura

Este projeto utiliza o critério Índice Gini, seguindo a tendência na Física de Altas Energias.

Pruning

Esta operação é oposta a operação de divisão. Em máquina de aprendizado, utiliza-se *pruning* para se reduzir o tamanho da árvore de decisão, removendo algumas folhas ou nós após obtenção de folhas 100% puras. Como consequência é possível aumentar a eficiência da classificação a medida que ocorre redução de *overfitting*. Esta técnica não foi utilizada neste projeto, pois para evitar folhas

completamente puras, basta definir a priori o tamanho da árvore.

Uma das vantagens de se utilizar árvores de decisão para classificação é a sua fácil compreensão. Para pessoas sem conhecimentos em inteligência computacional, o treinamento de classificadores e os resultados obtidos são de fácil compreensão. Além disso, a representação gráfica é intuitiva e permite que hipóteses sejam rapidamente relacionadas.

Entretanto, existe uma desvantagem que faz com que árvores de decisão não sejam consideradas classificadores fortes: o fato de dividir conjuntos em sub-conjuntos homogêneos faz com que o classificador gerado fique especialista no conjunto de treino (*overfit*). Ou seja, a árvore gerada sempre dependerá do conjunto de treino utilizado. O método de aprendizagem de conjunto (*ensemble learning*) procura contornar essa questão levantada.

Ensemble são conjuntos de classificadores que combinam seus resultados individuais (por votação, por exemplo) para classificar novas amostras [59]. Uma das áreas mais ativas na área de aprendizado de máquina supervisionado tem sido *ensemble* [60]. Geralmente, classificadores *ensemble* apresentam maior acurácia que classificadores individuais. A razão estatística para isso é que a variância de um somatório é menor que a variância individual. Computacionalmente, o treinamento pode dar-se de maneira mais rápida, uma vez que, em alguns casos, é possível paralelizar a fase de treinamento.

Classificadores *ensemble* podem ser gerados com qualquer algoritmo base, inclusive árvores de decisão. Tais classificadores são descritos nas sub-seções a seguir.

4.3.1 Bagged Trees (*Bootstrap Aggregating*)

Neste tipo de *ensemble*, um sub-conjunto com N elementos do conjunto de treino é selecionado para o treinamento de um classificador. Em seguida, um novo sub-conjunto com outros N elementos é utilizado para gerar um outro classificador. Esse procedimento é repetido, até que um número total e pré-definido de classificadores seja gerado. A classificação final dá-se por voto majoritário. A figura 4.5 ilustra um pseudo-código que descreve como *Bagged Trees* é treinado e utilizado para classificações.

4.3.2 Random Forest

Alguns autores consideram *Random Forest* como uma variação de *Bagged Trees* [60]. Este classificador *ensemble* é apresentado por Breiman em [61].

A primeira diferença é que *Random Forest* só utiliza árvores de decisão como algoritmo base. Outra diferença entre os dois algoritmos é a quantidade de atribu-

```

BAGGING

Training phase
1. Initialize the parameters
   •  $\mathcal{D} = \emptyset$ , the ensemble.
   •  $L$ , the number of classifiers to train.

2. For  $k = 1, \dots, L$ 
   • Take a bootstrap sample  $S_k$  from  $\mathbf{Z}$ .
   • Build a classifier  $D_k$  using  $S_k$  as the training set.
   • Add the classifier to the current ensemble,  $\mathcal{D} = \mathcal{D} \cup D_k$ .

3. Return  $\mathcal{D}$ .

Classification phase
4. Run  $D_1, \dots, D_L$  on the input  $\mathbf{x}$ .
5. The class with the maximum number of votes is chosen as the label
   for  $\mathbf{x}$ .

```

Figura 4.5: Pseudo-código para *Bagged Trees*. Retirado de [7].

tos utilizados durante o treinamento dos classificadores: neste caso, nem todos os atributos são utilizados. Escolhe-se a quantidade de atributos que será utilizada e a cada nova árvore, um sub-conjunto diferente de atributos é utilizado. Isso permite analisar a contribuição de cada atributo no treinamento de cada árvore.

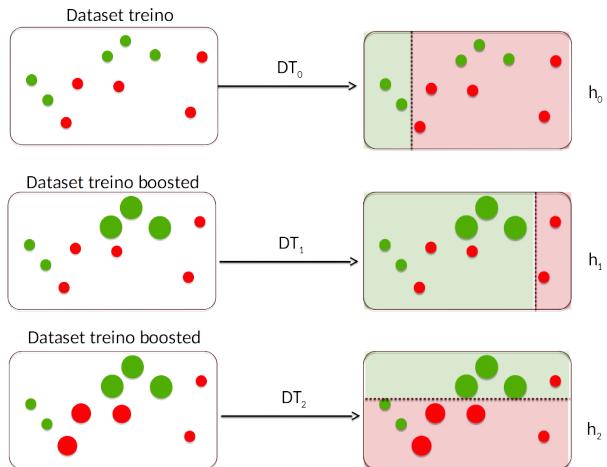
O termo *Out of Bag (OOB)* também é introduzido por Breiman. Assim, tem-se uma nova medida para erro: cerca de um terço do conjunto de treino é separada e utilizada a cada rodada como um conjunto de teste. É a partir desta medida que a análise de contribuição dos atributos torna-se possível.

Similar ao *Bagged Trees*, o resultado final dá-se por voto majoritário.

4.3.3 Boosted Decision Trees (BDT)

Diferentemente dos casos anteriores, todo o conjunto de treino é utilizado para gerar o classificador. Mas, a cada rodada, verifica-se quais amostras foram erroneamente classificadas e atribui-se um peso a elas. Desta maneira, na próxima iteração, os casos equivocados estarão evidenciados. A figura 4.6(a) ilustra o que ocorre com o conjunto de treino a cada rodada: dadas duas classes, verde e vermelha, ao gerar o primeiro classificador, temos 3 amostras erroneamente classificadas (*misclassifieds*). Em outras palavras, temos 3 amostras que pertencem à classe verde, mas o classificador gerado atribuiu a classe vermelha a elas. Em uma próxima iteração, tais amostras serão multiplicadas por um fator que influenciará o treinamento. O resultado final ocorre por voto majoritário.

É interessante observar na figura 4.6(b), que a combinação de diversos classificadores gera um classificador refinado.



(a) Exemplo do que ocorre com o conjunto de treino durante as iterações do BDT

$$\hat{y} = \sum_{n=1}^N \alpha_n h_n$$

α_0
 +
 α_1
 +
 α_2

(b) O resultado final é por voto majoritário.

Figura 4.6: Ilustração do método BDT. Baseado em [8].

Capítulo 5

Métodos e Resultados

As seções 5.1 e 5.2 descrevem, respectivamente, o conjunto de dados adquiridos e os atributos que foram previamente selecionados.

Em seguida o capítulo apresenta uma discussão do desempenho obtido pelos classificadores *ensemble*.

5.1 Aquisição de códigos fonte

O Tile-in-ONE encontra-se em produção, sendo utilizado pelos grupos de calibração e qualidade de dados do TileCal desde 2014. Em seu banco de dados, foram selecionados para este projeto um total de 512 códigos de *Plugins*, desenvolvidos pelos colaboradores através da plataforma web. A figura 5.1 ilustra a quantidade de códigos fonte implementados no período de Junho de 2014 a Dezembro de 2015.

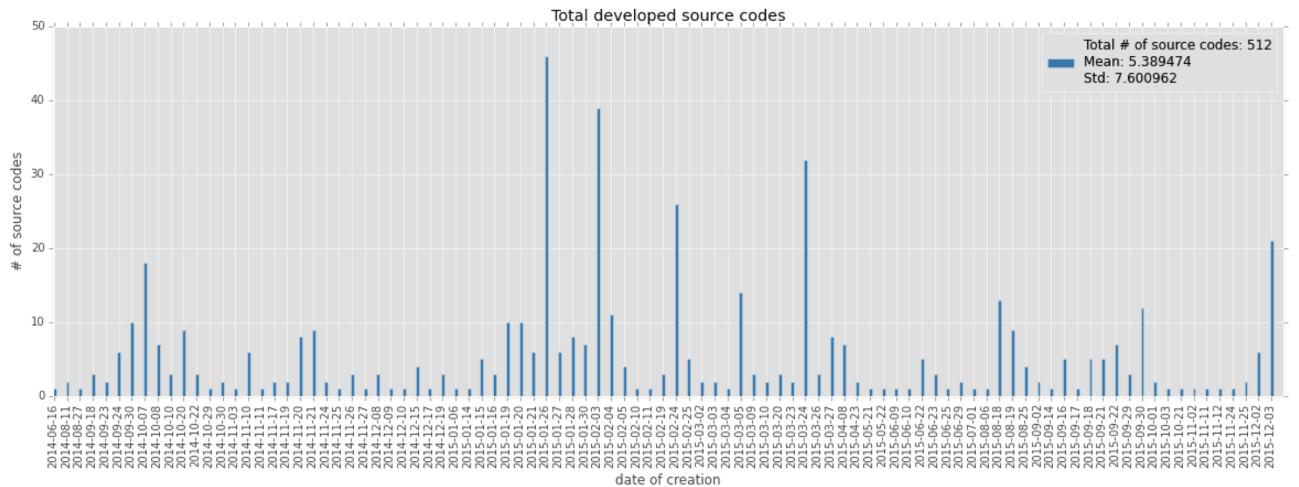


Figura 5.1: Total de códigos fonte escritos (512) entre Junho/2014 e Dezembro/2015.

Dos 512 códigos que compõem o conjunto de dados, quase 30% não obtiveram

sucesso ao serem enviados para uma máquina *slave*. De acordo com a figura 5.2, percebe-se que 8,59% não retornaram nenhum tipo de resposta para o servidor principal, o que pode significar que a máquina *slave* escolhida para executar o código fonte em questão estava sem comunicação. Isso não significa necessariamente que o código fonte é *não executável*, uma vez que ele nem sequer chegou na máquina *slave*. Os 21,09% restantes correspondem a programas que falharam durante a tentativa de execução. Mas novamente, não é possível afirmar neste momento que tais códigos são *não executáveis*. Se por ventura, algum desses tentou acessar um banco de dados (externo a plataforma) que no momento da execução estava inacessível, a máquina *slave* retornou falha e, neste caso, o problema não foi o código fonte. Talvez no futuro, esse tipo de informação possa se tornar uma entrada para o classificador: caso o código fonte esteja acessando uma fonte externa ele pode ser penalizado.

Source codes execution performance (slave machines)

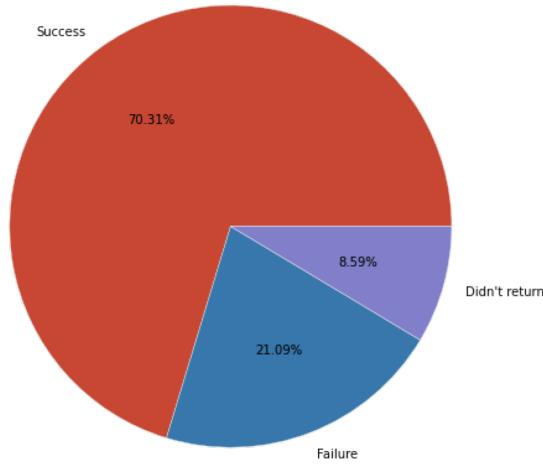


Figura 5.2: Retornos obtidos após tentativa de executar códigos fonte em máquinas *slave*.

Após avaliar os 512 códigos, e comparar com a aplicação do analisador estático (PyLint), percebe-se que 82,02% dos códigos são *executáveis*, mas em quase 20% dos casos o PyLint gera algum tipo de alerta. Do restante, menos de 1% é *não executável* devido a erros de sintaxe em Python, mas podem ser facilmente identificados ao executar o analisador estático. Dos outros quase 17% *não executáveis*, cerca de 5% não podem ser identificados apenas com a aplicação do Pylint. A figura 5.3 ilustra esses percentuais.

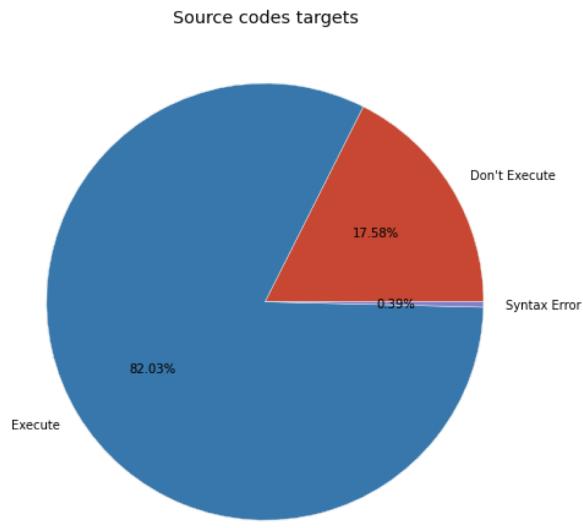


Figura 5.3: Percentuais de alvos do conjunto de códigos fonte.

Em suma, a aplicação do analisador estático não é suficiente para classificar um código como *executável* ou *não executável*. Uma ferramenta adicional faz-se necessária.

5.2 Seleção de atributos

A tabela 4.1 descreve os 11 atributos inicialmente extraídos dos códigos fonte disponíveis na base de dados da plataforma Tile-in-ONE. Ao gerar a matriz de correlação entre tais atributos (figura 5.4), percebe-se que os atributos relacionados às medidas e estatísticas de Halstead são fortemente correlacionados. Treinar um modelo de aprendizado de máquina com atributos altamente correlacionados não melhora a performance dos classificadores gerados e pode aumentar o tempo de convergência.

Tabela 5.1: Lista de atributos selecionados

#	Atributo	Descrição
1	Complexidade Ciclomática	Número de declarações
2	Índice de Manutenibilidade	Corresponde à organização do código
3	Volume de Halstead	Combinação entre número de linhas e declarações
4	Tempo de Halstead	Tempo estimado para compilar um código
5	LLOC	Número de linhas lógicas
6	Alertas PyLint	Verdadeiro se PyLint gerou alertas. Caso contrário, falso

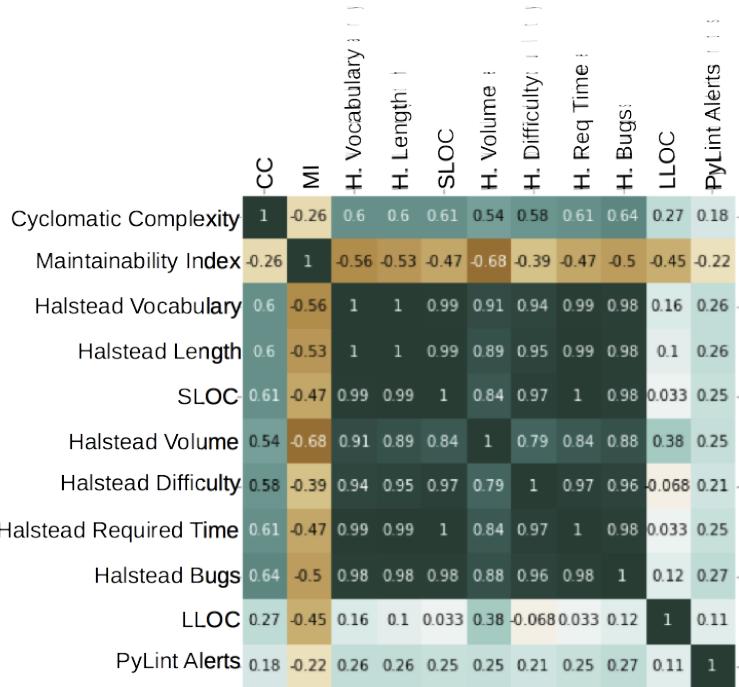


Figura 5.4: Matriz de correlação dos 11 atributos extraídos inicialmente

O teste χ^2 foi aplicado para entender, dentre os atributos relacionados a Halstead, quais são os dois mais relevantes. Para este teste, os atributos de Halstead mais relevantes são: *Halstead Volume* e *Required Time*. A nova matriz de correlação calculada (figura 5.5) demonstra que os 6 atributos selecionados (tabela 5.1) são suficientemente independentes entre si (com exceção das medidas de Halstead). Observa-se ainda a independência dos atributos selecionados com o alvo, como ilustrado na figura 5.5.

	CC	MI	H. Volume	H. Req Time	LLOC	PyLint Alerts	Target
CC	1	-0.26	0.27	0.58	0.18	0.54	-0.11
MI	-0.26	1	-0.45	-0.39	-0.22	-0.68	0.32
H. Volume	0.27	-0.45	1	-0.068	0.11	0.38	-0.036
H. Req Time	0.58	-0.39	-0.068	1	0.21	0.79	-0.21
LLOC	0.18	-0.22	0.11	0.21	1	0.25	-0.23
PyLint Alerts	0.54	-0.68	0.38	0.79	0.25	1	-0.2
Target	-0.11	0.32	-0.036	-0.21	-0.23	-0.2	1

Figura 5.5: Matriz de correlação dos 6 atributos selecionados. As correlações com a saída também são calculadas.

5.3 Treinamento

Em Física de Altas Energias, é comum encontrar a aplicação de BDT (*Boosted Decision Tree*, ver seção 4.3.3) para identificação de partículas. No Farmilab, por exemplo, as análises para busca de oscilações de neutrinos deu-se por meio de aplicação de BDT [62]. No CERN existem diversos trabalhos utilizando BDT para identificação do partículas ([63], [64] e [65] por exemplo). Portanto, existe um incentivo natural para a aplicação de métodos *ensemble* com árvores neste projeto contextualizado no ambiente do CERN. Como citado, a popularidade de árvores de decisão na comunidade científica vem da sua fácil compreensão.

Com os atributos selecionados (seção 5.2), classificadores em *ensemble* foram treinados. Para todos os casos, o algoritmo base é uma árvore de decisão, gerada previamente. Como saídas, teremos as classes *executável* indicando que provavelmente o código fonte não vai falhar ao ser executado na máquina *slave* ou, caso contrário, *não executável*.

Antes de dividir o conjunto de dados em subconjuntos de treino (70%) e teste (30%), foi necessário replicar o conjunto com alvo *não executável*, devido a desproporção em número de amostras. É importante ressaltar que os mesmos conjuntos de treino e teste foram utilizados para todos os casos. Como medida de incerteza, foi utilizado o erro quadrático médio das 100 iterações realizadas.

Como avaliação de performance, foram calculados matrizes de confusão e F1-scores. Em análises estatísticas de classificação binária, o F1-score é uma medida de acurácia. Ela considera tanto precisão quanto sensibilidade, como pode ser observado pela fórmula abaixo:

$$F1_{score} = 2 \cdot \frac{p \cdot r}{p + r} \quad (V)$$

Na fórmula (V) p é precisão e r é sensibilidade (do inglês, *recall*).

Precisão é o número de verdadeiro positivos dividido pelo número total de resultados positivos. Sensibilidade é o número de verdadeiro positivos dividido pelo número de resultados positivos que deveriam ter sido retornados.

O F1-score pode ser interpretado como uma média ponderada da precisão e sensibilidade, onde o melhor valor para avaliar a acurácia é 1 e o pior, 0.

Árvore de decisão

O critério *índice gini* para divisão da árvore de decisão foi utilizado. Estipulou-se um tamanho máximo igual a três, a fim de evitar *overfitting*. Segundo a literatura, é esperado que este classificador tenha um desempenho pior quando comparado com os métodos em *ensemble*. Esta árvore foi utilizada como classificador base nos três métodos descritos a seguir.

Bagged Trees

Utilizando o mesmo conjunto de treino utilizado para treinar o classificador base e, utilizando a árvore de decisão descrita anteriormente, um classificador em *Bagged Tree* foi gerado. No caso, o número de iterações estabelecido é igual a 100. Ou seja, ao final do treinamento, tem-se 100 árvores treinadas. O resultado dá-se por voto majoritário dessas 100 árvores. A cada rodada, 10% do conjunto de treino foi utilizado como sub-conjunto *bootstrap* (o equivalente a cerca de 60 amostras). Isso é o suficiente para garantir que as amostras não sejam repetidas inúmeras vezes, o que mantém a independência dos 100 classificadores gerados pelo método.

Random Forest

Para o classificador em *Random Forest* os mesmos critérios estabelecidos para treinar o *Bagged Tree* se aplicam. Mas, o que difere os dois métodos é a quantidade de atributos utilizados em cada iteração para treinar uma árvore. Em [61], o Breiman avalia que empiricamente, um número de atributos próximo a \sqrt{N} (sendo N o número total de atributos) é suficiente para obter melhor acurácia. No caso

abordado nesta dissertação, temos um número de atributos pequeno (e igual a 6), de tal forma que $\sqrt{6} = 2.45$. O número de atributos, então, utilizado para treinar o classificador em *Random Forest* é igual a 3. Como o *OOB error* é calculado a cada iteração, uma análise de relevância de atributos também pode ser extraída durante o treinamento deste classificador.

Boosted Decision Trees

Apesar deste método permitir treinamento por *cross-validation*, o mesmo procedimento utilizado para treinar os classificadores descritos anteriormente foi utilizado (70% do conjunto para treino e 30% para teste). O motivo dessa escolha é permitir que a análise de performance seja mais justa. Existem diversos algoritmos que determinam como os pesos que serão atribuídos às amostras erroneamente classificadas em cada iteração são definidos. No caso, o algoritmo *AdaBoost* [66] (do inglês, *Adaptive Boosting*) foi utilizado. Inicialmente, os pesos atribuídos equivalem a 1. A cada iteração o percentual de amostras classificadas de maneira equivocada é calculado e o peso que será atribuído a tais amostras é definido baseado neste percentual. O resultado final dependerá também deste peso calculado a cada iteração.

A figura 5.6 ilustra um pseudo-código extraído de [66]. Nela é possível entender como o peso β é calculado, e também como o resultado final é obtido.

```

Algorithm AdaBoost
Input: sequence of  $N$  labeled examples  $\langle(x_1, y_1), \dots, (x_N, y_N)\rangle$ 
        distribution  $D$  over the  $N$  examples
        weak learning algorithm WeakLearn
        integer  $T$  specifying number of iterations
Initialize the weight vector:  $w_i^t = D(i)$  for  $i = 1, \dots, N$ .
Do for  $t = 1, 2, \dots, T$ 
    1. Set
        
$$\mathbf{p}^t = \frac{\mathbf{w}^t}{\sum_{i=1}^N w_i^t}$$

    2. Call WeakLearn, providing it with the distribution  $\mathbf{p}^t$ ; get back a
        hypothesis  $h_t: X \rightarrow [0, 1]$ .
    3. Calculate the error of  $h_t$ :  $\varepsilon_t = \sum_{i=1}^N p_i^t |h_t(x_i) - y_i|$ .
    4. Set  $\beta_t = \varepsilon_t / (1 - \varepsilon_t)$ .
    5. Set the new weights vector to be
```

$$w_i^{t+1} = w_i^t \beta_t^{1 - |h_t(x_i) - y_i|}$$

Output the hypothesis

$$h_f(x) = \begin{cases} 1 & \text{if } \sum_{t=1}^T (\log 1/\beta_t) h_t(x) \geq \frac{1}{2} \sum_{t=1}^T \log 1/\beta_t \\ 0 & \text{otherwise.} \end{cases}$$

Figura 5.6: Pseudo-código do algoritmo *AdaBoost*

5.4 Discussão de Resultados

As figuras 5.7 e 5.8 ilustram as matrizes de confusão e algumas medidas de desempenho, respectivamente, que foram geradas utilizando o mesmo conjunto de teste para os 4 classificadores gerados.

No caso da árvore de decisão, é esperado que seu desempenho seja pior comparativamente: o tamanho fixado em três e o número máximo de folhas fixado em cinco impede que ao final, o classificador gerado possua folhas mais puras. A vantagem dessa abordagem é que desta maneira, é mais difícil obter *overfitting*. Por outro lado, uma árvore de decisão é insuficiente para se obter generalização.

Tais questões são contornadas com a aplicação dos métodos *ensemble*, como previsto na teoria. Dos classificadores gerados por tal método, obtém-se melhor desempenho com o BDT para a classe *não executável*. Percebe-se que o número de iterações estabelecido em 100 já é suficiente para atingir tal desempenho. Como ilustrado na figura 5.8(d), o classificador em BDT apresenta cerca de 90% para precisão e sensibilidade nas duas classes (*executável* e *não executável*).

O classificador *Bagged Trees* é capaz de melhorar um pouco a classificação quando comparado com a árvore de decisão gerada. Isto também está coerente com a teoria. Gerar 100 árvores de decisão com sub-conjuntos diferentes aumenta a diversidade dos 70% do conjunto de treino. Tal procedimento é comparável ao treinamento por *cross-validation*. Isso explica o melhor desempenho deste classificador quando comparado com a árvore de decisão treinada.

Além da variação ocorrida no treinamento para gerar o classificador *Bagged Trees*, o *Random Forest* varia também a quantidade de atributos utilizados no momento do treinamento das 100 árvores de decisão em cada iteração. Esse fato, além de melhorar o desempenho da classificação, permite que uma avaliação de atributos mais relevantes seja feita. Em outras palavras, a cada árvore gerada é possível obter o erro *OOB* e, avalia-lo de acordo com os atributos que participaram daquela iteração. A figura 5.9 ilustra através de um gráfico de barras quais são os atributos mais relevantes dos 6 (listados na tabela 5.1) utilizados para o treinamento da *Random Forest*. É possível perceber que o índice de manutenabilidade é o atributo que mais contribui para o treinamento do classificador. A complexidade ciclomática chega a ser menos relevante até que as medidas de Halstead.

A tabela 5.2 apresenta as medidas de desempenho dos 4 classificadores gerados. Nela é possível identificar o melhor desempenho pelo F1-score do BDT sob os demais classificadores.

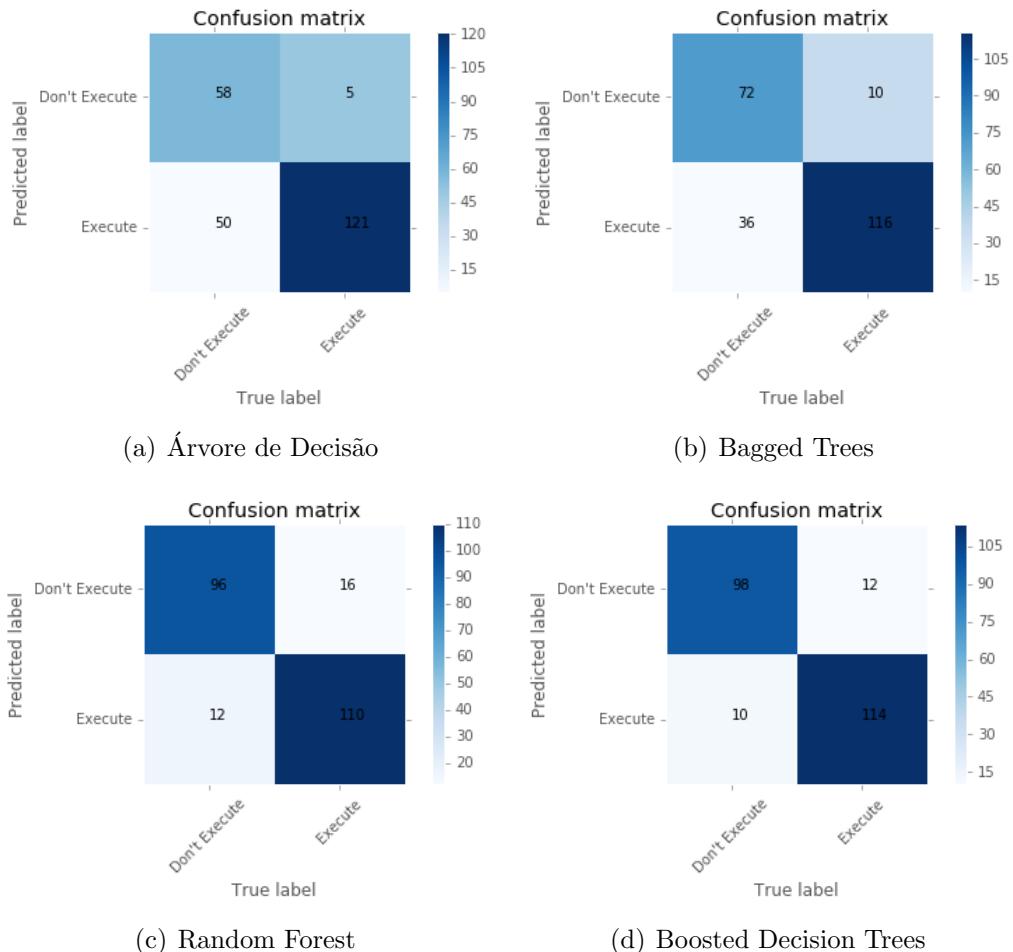


Figura 5.7: Matrizes de Confusão dos 4 classificadores gerados.

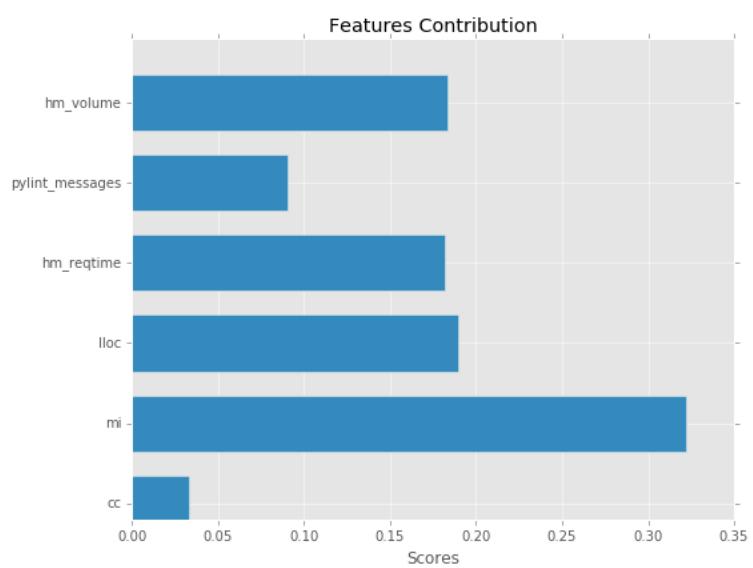
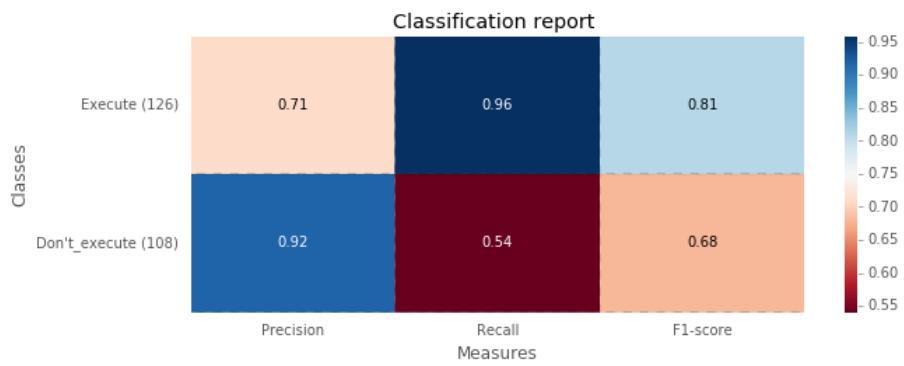
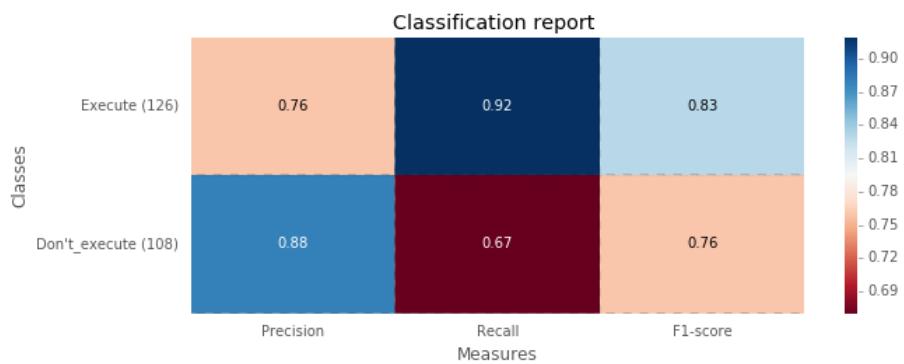


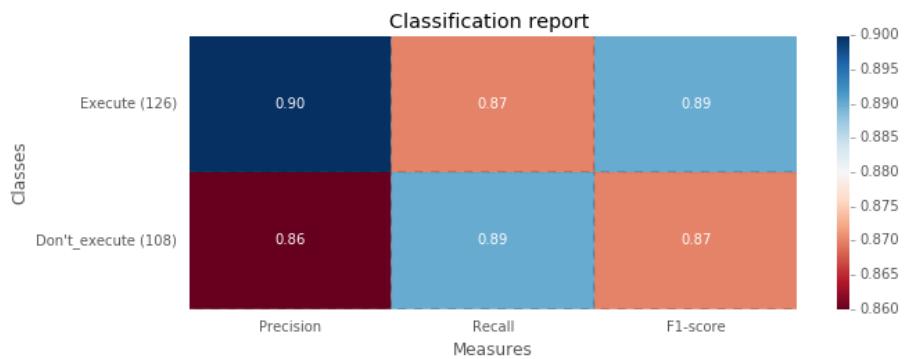
Figura 5.9: Atributos mais relevantes de acordo com erro OOB



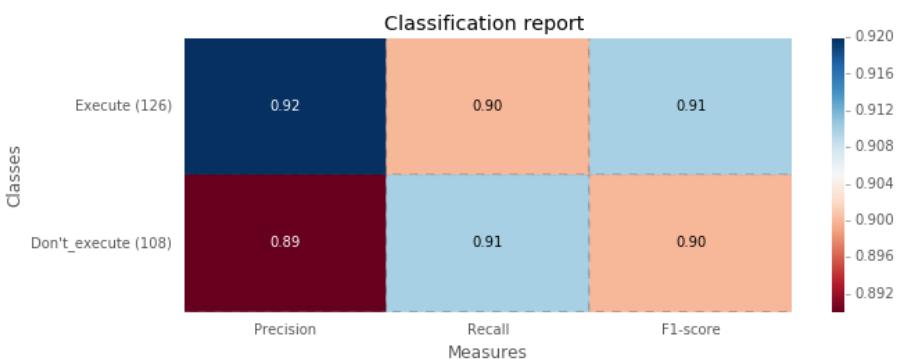
(a) Árvore de Decisão



(b) Bagged Trees



(c) Random Forest



(d) Boosted Decision Trees

Figura 5.8: Medidas de desempenho dos 4 classificadores gerados.

Tabela 5.2: Resumo das medidas de desempenho dos 4 classificadores gerados

Classificador / Classes	Precisão (Precision)		Sensibilidade (Recall)		F1-Score	
	Executável	Não executável	Executável	Não executável	Executável	Não executável
Árvore de Decisão	0,71 (0,03)	0,92 (0,09)	0,96 (0,04)	0,54 (0,13)	0,81 (0,03)	0,64 (0,11)
Bagged Trees	0,76 (0,02)	0,88 (0,05)	0,92 (0,03)	0,67 (0,12)	0,83 (0,02)	0,76 (0,07)
Random Forest	0,90 (0,01)	0,86 (0,03)	0,87 (0,01)	0,89 (0,01)	0,89 (0,01)	0,87 (0,02)
Boosted Decision Trees	0,92 (0,01)	0,89 (0,02)	0,90 (0,03)	0,91 (0,01)	0,91 (0,02)	0,90 (0,01)

Capítulo 6

Conclusões

Independente do desempenho obtido com os 4 classificadores gerados, é possível concluir que o problema de classificação de códigos fonte na plataforma web colaborativa Tile-in-ONE em *executável/não executável* pode ser abordado através de algoritmos de aprendizado de máquina em *ensemble*. Um próximo passo necessário para que tal funcionalidade seja colocada em produção seria analisar os códigos que são recorrentemente classificados de maneira equivocada e até mesmo identificar em qual parte do código exatamente encontra-se o possível *bug*.

Como analisado, existem códigos fonte, dentre os 512 utilizados para compor o conjunto de dados, que acessam fontes externas de dados (APIs e bancos de dados, por exemplo). Este fato pode ser relevante para o sucesso na execução do código e independe das variáveis de entrada que estão sendo consideradas neste projeto. Futuramente, pode-se verificar se esse atributo influencia na performance dos classificadores.

A discussão de resultados permite concluir que é possível aplicar mineração de código em um ambiente de Física de Altas Energias, extraindo-se apenas características de códigos fonte.

Através de um levantamento bibliográfico foi possível extrair 11 atributos relacionados a medidas de estatística e qualidade de códigos, dos quais, os mais independentes possíveis foram selecionados.

Para a classe *não executável* o BDT é o método *ensemble* que possui melhor desempenho, quando o F1-score (que é uma ponderação de precisão e sensibilidade) é observado. É comum a literatura eleger o BDT como melhor algoritmo dentre os quatro treinados neste projeto.

Referências Bibliográficas

- [1] “CDS - CERN Document Server”,
Acessado em janeiro de 2016. cds.cern.ch.
- [2] “ATLAS Photos”,
Acessado em janeiro de 2016. <http://www.atlas.ch/photos/>.
- [3] ATLAS COLLABORATION, *ATLAS: Technical Proposal for a General-Purpose pp Experiment at the Large Hadron Collider at CERN*, Tech. rep., CERN, 1994, CERN/LHCC 94–43.
- [4] ATLAS/TILE CALORIMETER COLLABORATION, *Tile Calorimeter Calorimeter Technical Design Report*, Tech. rep., CERN, 1996, CERN/LHCC 96–42.
- [5] BALABRAM, C. M. F. F. G. A. S. L., “Web System for Data Quality Assessment of Tile Calorimeter During the ATLAS Operation”, *Journal of Physics: Conference Series (JPCS)*, v. 331 - part 4, 2011.
- [6] PRAKASH, A., ASHOKA, D., ARADYA, V., “Application of Data Mining Techniques for Defect Detection and Classification”. In: *Proceeding of 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications. FICTA 2014. Advances in Intelligent Systems and Computing 327.*, v. 1, pp. 387–395, 2015.
- [7] ROKACH, L., MAIMON, O., *Data Mining with Decision Trees: Theory and Applications*. World Scientific Publishing Co., Inc.: River Edge, NJ, USA, 2008.
- [8] DUDA, R. O., HART, P. E., STORK, D. G., *Pattern Classification (2Nd Edition)*. Wiley-Interscience, 2000.
- [9] “The European Laboratory for Particle Physics”,
Acessado em janeiro de 2016. <http://www.cern.ch>.
- [10] “The Large Hadron Collider”,
Acessado em janeiro de 2016. <http://lhc.web.cern.ch>.

- [11] GRIFFTHS, D., *Introduction to Elementary Particles*. John Wiley & Sons, 1987.
- [12] “The Nobel Prize in Physics 2013”,
Acessado em fevereiro de 2016.
http://www.nobelprize.org/nobel_prizes/physics/laureates/2013/press.pdf.
- [13] THE ALICE COLLABORATION, “The ALICE experiment at the CERN LHC”, *Journal of Instrumentation*, v. 3, n. 08, Aug. 2008.
- [14] THE CMS COLLABORATION, “The CMS experiment at the CERN LHC”, *Journal of Instrumentation*, v. 3, n. 08, Ago 2008.
- [15] SZUMLAK, T., “The LHCb experiment”, *Acta Physica Polonica. Series B: Elementary Particle Physics, Nuclear Physics, Statistical Physics, Theory of Relativity, Field Theory*, v. 41, n. 7, pp. 1661–1668, 2010.
- [16] “ATLAS Factsheet”,
Acessado em janeiro de 2016.
http://www.atlas.ch/pdf/fact_sheet_1page.pdf.
- [17] JENNI, P., NESSI, M., NORDBERG, M., et al., “ATLAS high-level trigger, data-acquisition and controls: Technical Design Report”, *Technical Design Report ATLAS. CERN, Genebra*, v. 42, 2003.
- [18] P. CALAFIURA AND W. LAVRIJSEN AND OTHERS, “The athena control framework in production, new developments and lessons learned”, *Journal of Physics: Conference Series*, pp. 456–458, 2004.
- [19] CORSO-RADU, A., HADAVAND, H., HAUSCHILD, M., et al., “Data Quality Monitoring Framework for the ATLAS Experiment at the LHC”, *IEEE Xplore Digital Library*, v. 55, 2008.
- [20] VERDUCCI, M., “ATLAS conditions database experience with the LCG COOL conditions database project”, *Journal of Physics: Conference Series (JPCS)*, v. 119 - part 4, 2008.
- [21] SIVOLELLA, A., MAIDANTCHIK, C., “The Monitoring and Calibration Web Systems for the ATLAS Tile Calorimeter Data Quality Analysis”, *Journal of Physics: Conference Series 052037, Computing High Energy Physics, New York, USA*, v. 396, pp. 8, May 2012.

- [22] MAIDANTCHIK, C., FERREIRA, F., GRAEL, F., “DCS Web System”, *Journal of Physics: Conference Series, Computing High Energy Physics, Praga, Rep. Tcheca*, pp. 8, March 2009.
- [23] “Python Programming Language”,
Acessado em janeiro de 2016.
<http://www.python.org/>.
- [24] MCCONNEL, S., “An ounce of prevention”, *IEEE software*, May/June 2001.
- [25] KUMARESH, S., BASKARAN, R., “Defect Analysis and Prevention for Software Process Quality Improvement”, *International Journal of Computer Applications (0975 - 8887)*, v. 8, n. 7, October 2010.
- [26] CHILLAREGE, R., BHANDARI, I., CHAAR, J., et al., “Orthogonal Defect Classification - A Concept for In-Process Measurements”, *IEEE Transactions on Software Engineering*, v. 18, n. 11, November 1992.
- [27] VELMOUROUGAN, S., DHAVACHELVAN, P., BASKARAN, R., et al., “Software Development Life Cycle Model to Improve Maintainability of Software Applications”. In: *Advances in Computing and Communications (ICACC), 2014 Fourth International Conference on*, pp. 270–273, Aug 2014.
- [28] “IBM”,
Acessado em janeiro de 2016.
<http://www.ibm.com/us-en/>.
- [29] “HCL”,
Acessado em janeiro de 2016.
<http://www.hcl.com/>.
- [30] XIE, T., PEI, J., HASSAN, A., “Mining Software Engineering Data”. In: *Software Engineering - Companion, 2007. ICSE 2007 Companion. 29th International Conference on*, pp. 172–173, May 2007.
- [31] HASSAN, A., “The road ahead for Mining Software Repositories”. In: *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pp. 48–57, Sept 2008.
- [32] KHATOON, S., LI, G., MAHMOOD, A., “Comparison and evaluation of source code mining tools and techniques: A qualitative approach”. v. 17, pp. 459–484, May 2013.

- [33] LI, Z., ZHOU, Y., “PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code”. In: *In Proc. 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2005)*, pp. 306–315, ACM Press, 2005.
- [34] KRISHNA RAMANATHAN, M., GRAMA, A., JAGANNATHAN, S., “Path-Sensitive Inference of Function Precedence Protocols”. In: *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pp. 240–250, May 2007.
- [35] CHANG, R.-Y., PODGURSKI, A., YANG, J., “Finding What’s Not There: A New Approach to Revealing Neglected Conditions in Software”. In: *Proceedings of the 2007 International Symposium on Software Testing and Analysis, ISSTA ’07*, pp. 163–173, ACM: New York, NY, USA, 2007.
- [36] “CERN Security Rules”,
Acessado em fevereiro de 2016.
<https://security.web.cern.ch/security/rules/en/index.shtml>.
- [37] “PyLint - Static Analysis Tool for Python”,
Acessado em janeiro de 2016. <http://www pylint.org/>.
- [38] “CERN IT Security Recomended Tools”,
Acessado em janeiro de 2016. https://security.web.cern.ch/security/recommendations/en/code_tools.shtml.
- [39] BESSEY, A., BLOCK, K., CHELF, B., et al., “A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World”, *Commun. ACM*, v. 53, n. 2, pp. 66–75, Feb. 2010.
- [40] BRUN, R., RADEMAKERS, F., “ROOT-an object oriented data analysis framework”. In: *World Scientific Publishing*, 2005.
- [41] “Coverity efects at CERN”,
Acessado em janeiro de 2016. <http://www.coverity.com/press-releases/cern-chooses-coverity-to-ensure-accuracy-of-large-hadron-colliders>.
- [42] HECKMAN, S., WILLIAMS, L., “A Systematic Literature Review of Actionable Alert Identification Techniques for Automated Static Code Analysis”, *Inf. Softw. Technol.*, v. 53, n. 4, pp. 363–387, April 2011.

- [43] HECKMAN, S., WILLIAMS, L., “A measurement framework of alert characteristics for false positive mitigation models”, *North Carolina State University TR-2008- 23*, October 2008.
- [44] FAIRLEY, R., “Tutorial: Static Analysis and Dynamic Testing of Computer Software”, *Computer*, v. 11, n. 4, pp. 14–23, April 1978.
- [45] TISCHLER, R., SCHAUFLER, R., PAYNE, C., “Static Analysis of Programs As an Aid to Debugging”, *SIGSOFT Softw. Eng. Notes*, v. 8, n. 4, pp. 155–158, March 1983.
- [46] YUKSEL, U., SAZER, H., “Automated Classification of Static Code Analysis Alerts: A Case Study.” In: *ICSM*, pp. 532–535, IEEE Computer Society, 2013.
- [47] HECKMAN, S., WILLIAMS, L., “A Model Building Process for Identifying Actionable Static Analysis Alerts”. In: *Software Testing Verification and Validation, 2009. ICST '09. International Conference on*, pp. 161–170, April 2009.
- [48] HECKMAN, S., WILLIAMS, L., “On Establishing a Benchmark for Evaluating Static Analysis Alert Prioritization and Classification Techniques”. In: *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '08*, pp. 41–50, ACM: New York, NY, USA, 2008.
- [49] AGGARWAL, A., JALOTE, P., “Integrating Static and Dynamic Analysis for Detecting Vulnerabilities”. In: *Proceedings of the 30th Annual International Computer Software and Applications Conference - Volume 01, COMPSAC '06*, pp. 343–350, IEEE Computer Society: Washington, DC, USA, 2006.
- [50] BOOGERD, C., MOONEN, L., “Prioritizing Software Inspection Results using Static Profiling”. In: *Source Code Analysis and Manipulation, 2006. SCAM '06. Sixth IEEE International Workshop on*, pp. 149–160, Sept 2006.
- [51] KIM, S., ERNST, M., “Prioritizing Warning Categories by Analyzing Software History”. In: *Mining Software Repositories, 2007. ICSE Workshops MSR '07. Fourth International Workshop on*, pp. 27–27, May 2007.
- [52] KIM, S., ERNST, M. D., “Which Warnings Should I Fix First?” In: *Proceedings of the the 6th Joint Meeting of the European Software Engineering*

Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC-FSE '07, pp. 45–54, ACM: New York, NY, USA, 2007.

- [53] KREMENEK, T., ENGLER, D., “Z-ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations”. In: *Proceedings of the 10th International Conference on Static Analysis, SAS'03*, pp. 295–315, Springer-Verlag: Berlin, Heidelberg, 2003.
- [54] HALSTEAD, M. H., *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc.: New York, NY, USA, 1977.
- [55] MACCABE, T., MCCABE, ASSOCIATES, et al., *Structured testing. Tutorial Texts Series*, IEEE Computer Society Press, 1983.
- [56] JONES, J., “Abstract Syntax Tree Implementation Idioms”. In: *Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003)*, 2003.
- [57] OMAN, P., HAGEMEISTER, J., “Metrics for assessing a software system’s maintainability”. In: *Software Maintenance, 1992. Proceedings., Conference on*, pp. 337–344, Nov 1992.
- [58] SHANNON, C. E., “A Mathematical Theory of Communication”, *SIGMOBILE Mob. Comput. Commun. Rev.*, v. 5, n. 1, pp. 3–55, Jan. 2001.
- [59] SENI, G., ELDER, J., *Ensemble Methods in Data Mining: Improving Accuracy Through Combining Predictions*. Morgan and Claypool Publishers, 2010.
- [60] DIETTERICH, T. G., “Ensemble Methods in Machine Learning”. In: *Proceedings of the First International Workshop on Multiple Classifier Systems, MCS '00*, pp. 1–15, Springer-Verlag: London, UK, UK, 2000.
- [61] BREIMAN, L., “Random Forests”, *Mach. Learn.*, v. 45, n. 1, pp. 5–32, Oct. 2001.
- [62] ROE, B. P., YANG, H.-J., ZHU, J., et al., “Boosted decision trees as an alternative to artificial neural networks for particle identification ”, *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, v. 543, n. 2 - 3, pp. 577 – 584, 2005.
- [63] COLLABORATION, T. A., “Modelling $Z\bar{A}$ processes in ATLAS with \tilde{l} -embedded $Z\bar{A}$ $\tilde{l}_1\bar{l}_1$ data”, *Journal of Instrumentation*, v. 10, n. 09, pp. P09018, 2015.

- [64] CHATRCHYAN, S., KHACHATRYAN, V., SIRUNYAN, A., et al., “Search for the standard model Higgs boson decaying to bottom quarks in pp collisions at”, *Physics Letters B*, v. 710, n. 2, pp. 284 – 306, 2012.
- [65] GODFREY, J., *Using Boosted Decision Trees For Tau Identification in the ATLAS Experiment*, Master’s Thesis, University of the Fraser Valley, 2009.
- [66] FREUND, Y., SCHAPIRE, R. E., “A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting”, *J. Comput. Syst. Sci.*, v. 55, n. 1, pp. 119–139, Aug. 1997.