UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

INSTITUTO DE INFORMÁTICA

PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

ANDRWS AIRES VIEIRA

Uma abordagem para estimação prévia dos requisitos não funcionais em sistemas embarcados utilizando métricas de software

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientadora: Profª. Drª. Érika Cota

Porto Alegre

2015

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Aires Vieira, Andrws
Uma abordagem para estimação prévia dos requisitos não funcionais em sistemas embarcados utilizando métricas de software / Andrws Aires Vieira. -- 2015.
104 f.

Orientadora: Érika Fernandes Cota.

Dissertação (Mestrado) -- Universidade Federal do Rio Grande do Sul, Instituto de Informática, Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2015.

1. Sistemas Embarcados. 2. Métricas de Software. 3. Análise de Regressão. 4. Estimação Prévia. 5. Requisitos não Funcionais. I. Fernandes Cota, Érika, orient. II. Título.

RESUMO

O crescente aumento da complexidade dos sistemas embarcados demanda consigo a necessidade do uso de novas abordagens que acelerem o seu desenvolvimento, como por exemplo, o desenvolvimento baseado em modelos. Essas novas abordagens buscam aumentar o nível de abstração, utilizando conceitos de orientação a objetos e UML para modelar um software embarcado. Porém, com o aumento do nível de abstração, o projetista de software embarcado não possui a ideia exata do impacto de suas decisões de modelagem em questões importantes, como desempenho, consumo de energia, entre tantas outras que são de suma importância em um projeto embarcado. Dessa forma, se fazem necessárias técnicas de análise e/ou estimação de projeto que, em um ambiente de desenvolvimento mais abstrato, possam auxiliar o projetista a tomar melhores decisões nas etapas inicias de projeto, garantindo assim, as funcionalidades (requisitos funcionais) e os requisitos não funcionais do sistema embarcado. Neste trabalho, propõe-se estimar os requisitos não funcionais de um sistema embarcado a partir de informações (métricas) extraídas das etapas iniciais do projeto. Pretende-se com isso auxiliar o projetista na exploração do espaço de projeto já nos estágios iniciais do processo de desenvolvimento, através de uma rápida realimentação sobre o impacto de uma decisão de projeto no desempenho da aplicação em uma dada plataforma de execução. Os resultados experimentais mostram a aplicabilidade da abordagem, principalmente para um ambiente de evolução e manutenção de projetos de software, onde se tem um histórico de métricas de aplicações semelhantes para serem usadas como dados de treinamento. Neste cenário, a abordagem proposta possui acurácia de pelo menos 98% para as estimativas apresentadas ao projetista. Em um cenário heterogêneo, assumindo o uso da metodologia em um sistema diferente daquele usado para treinamento, a acurácia do método de estimação cai para pelo menos 80%.

Palavras-chave: Sistemas Embarcados. Métricas de Software. Análise de Regressão. Estimação Prévia. Requisitos não Funcionais.

An approach to early estimation of non-functional requirements for embedded systems using software metrics

ABSTRACT

The increasing complexity of embedded systems demands the use of new approaches to accelerate their development, such as model-driven engineering. Such approaches aim at increasing the level of abstraction using concepts such as object-orientation and UML for modeling the embedded software. However, with the increase of the abstraction level, the embedded software developer looses controllability and predictability over important issues such as performance, power dissipation and memory usage for a specific embedded platform. Thus, new design estimation techniques that can be used in the early development stages become necessary. Such a strategy may help the designer to make better decisions in the early stages of the project, thus ensuring the final system meets both functional and non-functional requirements. In this work, we propose an estimation technique of non-functional requirements for embedded systems, based on data (metrics) extracted from early stages of the project. The proposed methodology allows to better explore different design options in the early steps of software development process and can therefore provide a fast and yet accurate feedback to the developer. Experimental results show the applicability of the approach, particularly for software evolution and maintenance, which has a history of similar applications metrics to be used as training data. In this scenario, the accuracy of the estimation is at least of 98%. In a heterogeneous scenario, where the estimation is performed for a system that is different from the one used during training, the accuracy drops to 80%.

Keywords: Embedded Systems. Software Metrics. Regression Analysis. Early Estimation. Non-functional Requirements.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AMLOC	Average Method LOC
AOSD	Aspect-Oriented Software Development
CBO	Coupling Between Objects
CC	Complexidade Ciclomática
CC2	Strict Cyclomatic Complexity
CC3	Modified Cyclomatic Complexity
CPU	Central Processing Unit
DCBD	Descoberta de Conhecimento em Banco de Dados
DIT	Depth of the Inheritance Tree
DLOC	Declarative Lines of Code
DSE	Design Space Exploration
ELOC	Executable Lines of Code
ev(G)	Essential Complexity
FDL	Formal Description Language
FP	Fault-Prone
GLS	Generalized least squares
IDE	Integrated Development Environmen
IRLS	Iteratively Reweighted Least Squares
ISA	Instruction Set Architecture
KDD	Knowledge-Discovery in Databases
KLOC	Kilo Lines of Code
LAD	Least Absolute Deviation
LCOM	Lack of Cohesion in Methods
LOC	Lines of Code
MDE	Model Driven Engineering
MLOC	Million Lines of Code
MLP	Multilayer Perceptron
MMLOC	Max Method LOC
MN	Max Nesting
NC	Number of Classes
NFP	Non-Fault-Prone

NI	Number of Interfaces
NL	Number of Lines
NOA	Number of Attributes
NOC	Number of Children
NOM	Number of Methods
NOP	Number of Parameters
NOPA	Number of Public Attributes
NOPK	Number of Packages
NRMSE	Normalized Root-Mean-Square Error
OLS	Ordinary Least Squares
OO	Orientação a Objetos
QMO	Quadrados Mínimos Ordinários
RAM	Random Access Memory
RF	Random Forest
RFC	Response for a Class
RNA	Redes Neurais Artificiais (RNA)
RSME	Root-Mean-Square Error
SOC	System on a Chip
SVM	Support Vector Machine
UML	Unified Modeling Language
WMC	Weighted Methods pes Class

SUMÁRIO

1 INTRODUÇÃO

Sistemas embarcados são sistemas de processamento de informação que são incorporados em um produto maior e que normalmente não são diretamente visíveis para o usuário (WOLF, 2012). Exemplos de sistemas embarcados incluem sistemas de processamento de informação em equipamentos de telecomunicações, em sistemas de transporte, em equipamentos de fabricação, em eletrônicos de consumo, dentre tantos outros equipamentos (MARWEDEL, 2006).

A possibilidade de utilização de dispositivos embarcados nas mais diversas aplicações requer a definição de funcionalidades bastante distintas e um tempo de desenvolvimento bastante reduzido para um produto, principalmente no que tange à eletrônica de consumo. Porém, o incremento de funcionalidade não está necessariamente associado a um aumento de valor de venda do produto ou margem de lucro. Pelo contrário, agregar tecnologia e funcionalidade a seus produtos é uma questão de sobrevivência das empresas.

Características tais como a especificidade ou a multiplicidade de funcionalidades, o uso da eletrônica e do software como meio e parte de um sistema maior e mais complexo, as restrições de hardware em termos de quantidade de memória, consumo de energia e requisitos de processamento, a necessidade de reduzir ao máximo o tempo de projeto, e a sensibilidade ao custo, são algumas das características que definem os sistemas embarcados, os quais, atualmente, são responsáveis pela maior parte do mercado eletroeletrônico mundial (EBERT and JONES, 2009).

Diversas classes de sistemas embarcados podem ser identificadas: sistemas de automação industrial, eletrônica de consumo, sistemas de comunicação, automotivos, sistemas médicos, instrumentação eletroeletrônica, sistemas para aviação e militares, entre outros (UBM TECH ELECTRONICS, 2013). Embora compartilhem características gerais que as diferenciam de aplicações não embarcadas, cada classe de aplicação possui requisitos funcionais e não funcionais bastante distintos e exigem diferentes níveis de garantia da qualidade.

Em sistemas embarcados, o atendimento dos requisitos não funcionais é especialmente importante, pois normalmente o software está sendo executado em uma plataforma de hardware com recursos bem limitados. Requisitos tais como desempenho, consumo de energia, tamanho de software e/ou hardware, uso de memória, atendimento a prazos de execução (*deadlines*), entre outros definem se determinado sistema embarcado servirá ou não ao seu propósito.

Além disso, em sistemas embarcados normalmente também é preciso lidar com a competividade do mercado (*time-to-market*). Atrasos de poucas semanas no lançamento de um produto no mercado podem comprometer seriamente os ganhos esperados (CARRO and WAGNER, 2003) (EBERT and JONES, 2009).Por exemplo, pensando no mercado de celulares, existem diversas empresas que competem para lançar um novo modelo antes da concorrente. Segundo Smith (SMITH, 2004), o prejuízo de apenas um dia de atraso em grandes empresas pode chegar a 1 milhão de dólares.

Conforme destacado, os sistemas embarcados necessitam atender algumas exigências específicas, por exemplo, baixo consumo de energia, invulnerabilidade, desempenho, etc. Todavia, para analisar esses requisitos é necessário definir ao menos uma plataforma de hardware para simulação. No entanto, usualmente também avalia-se mais de uma plataforma visando encontrar o melhor *trade-off* entre todos os requisitos não funcionais que devem ser considerados e a plataforma de hardware, o que torna esta tarefa de Exploração de Espaço de Projeto (DSE – do inglês *Design Space Exploration*) ainda mais complexa.

Outro fator que deve ser levado em consideração é qualidade interna do software. Estudos internacionais (CURTIS, 2009) mostram que a baixa qualidade interna do software gera uma série de sobrecustos para o negócio, embora tais custos nem sempre sejam explicitamente medidos. Por exemplo, defeitos no produto são muitas vezes causados não por problemas nos requisitos funcionais, mas frequentemente são resultado da baixa qualidade interna do software que dificulta, por exemplo, operações de manutenção. Baixa qualidade interna gera riscos de indisponibilidade do sistema, corrupção de dados, não atendimento de normas técnicas, entre outros. Tais riscos dificilmente são detectados pela bateria de testes funcionais do sistema. O próprio teste, por sua vez, se torna lento e pouco efetivo em um sistema de menor qualidade interna. Por fim, quando há problemas de qualidade interna, nota-se uma degradação no desempenho do software ao longo do tempo.

Todos estes riscos podem ser traduzidos em perdas financeiras para o negócio. Embora não se tenha conhecimento de estatísticas brasileiras sobre esta questão, sabe-se, por números internacionais (CURTIS, 2009) que, por exemplo, 10% de queda de desempenho na aplicação pode levar a uma perda de produtividade da ordem de um milhão de dólares em um trimestre. Por outro lado, o mesmo estudo (CURTIS, 2009) mostrou que melhorias da qualidade interna do software reduziram em 25% a quantidade de retrabalho em 1 ano e reduziram em 60% o esforço de manutenção, permitindo uma economia de $75.000 dólares por aplicação, em mudanças para incremento de funcionalidades. Por fim, sabe-se que qualidade interna está

diretamente relacionada à flexibilidade do software e, portanto, à sua capacidade de adequar-se às frequentes mudanças de requisitos. Dessa forma, qualidade interna do software é essencial para aumentar a agilidade e competitividade do negócio.

Resumindo, é necessária uma boa estratégia nas fases inicias do projeto que tenha como objetivos: 1) buscar (ou estimar) que os requisitos não funcionais do sistema embarcado sejam respeitados ao fim do projeto; 2) garantir um nível de qualidade interna aceitável. Pois, uma vez que o objetivo 1 não for respeitado o software terá de ser refatorado, e quando isso for necessário, se o software tiver uma baixa qualidade, o resultado será um processo muito lento e custoso.

Por outro lado, nota-se um crescente aumento na complexidade dos sistemas embarcados atualmente, conforme demonstra a Figura 1.1. Para lidar com esse crescente aumento na complexidade, técnicas que possam acelerar o desenvolvimento de software abstraindo essa complexidade se fazem necessárias, para que o *time-to-market* continue sendo satisfeito.

Logo, linguagens mais abstratas (C++, Java) e com recursos mais avançados, como orientação a objetos, precisam ser usadas para abstrair tal complexidade e lidar com demanda das inúmeras funcionalidades das aplicações. Ainda nesse contexto, o Desenvolvimento Baseado em Modelo (Model-Driven Development – MDD), surge como uma excelente solução, pois centra as atenções diretamente na modelagem da aplicação. Uma linguagem que vem ganhando grande espaço ao longo do tempo entre os projetistas de sistemas embarcados para modelagem é a Unified Modeling Language (UML) (OMG, 2014) e suas diversas extensões/adaptações.

Muitos trabalhos (KNORRECK, APVRILLE and PACALET, 2010) (VIDAL, DE LAMOTTE, *et al.*, 2009) (WANG, 2009) (RICCOBENE, SCANDURRA, *et al.*, 2005) (MARTIN, 2002) (DE JONG, 2002) (MARTIN, LAVAGNO and LOUIS-GUERIN, 2001), apresentam o potencial da UML como uma linguagem para a especificação e projeto de sistemas embarcados, devido à sua rica notação gráfica e à capacidade de modelagem, a qual habilita a captura da estrutura e do comportamento do sistema em múltiplos níveis de abstração. Além da UML, a Arquitetura Orientada a Modelo (*Model Driven Architecture* – MDA) está sendo promovida como uma nova abordagem para desenvolver sistemas embarcados através da especificação de modelos. A MDA provê mecanismos para aumentar a portabilidade, a interoperabilidade, a manutenibilidade e o reuso de modelos.

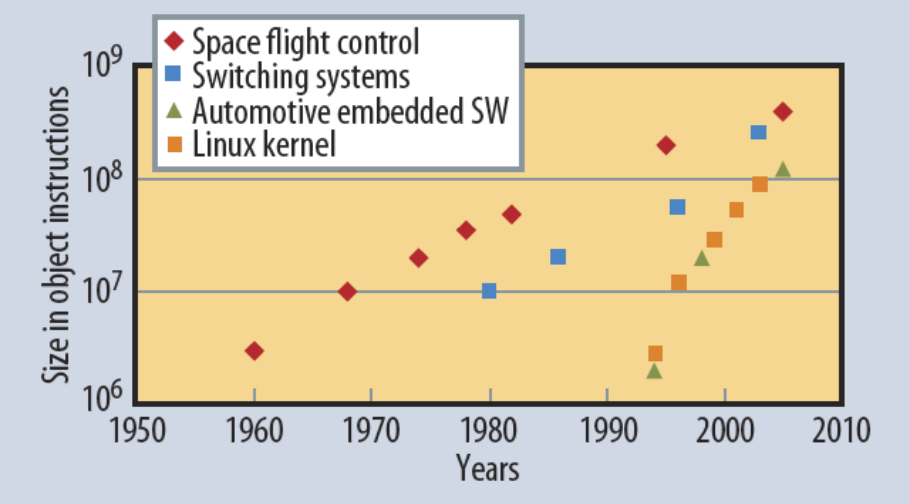
Porém, maior abstração significa menor previsibilidade e controle sobre os requisitos não funcionais e normalmente os engenheiros de software não têm como medir o impacto de suas decisões em questões essenciais durante o desenvolvimento de sistemas embarcados, como por exemplo, consumo de energia e desempenho em uma plataforma de hardware específica. Para que a exploração do espaço de projeto seja realizada nos estágios iniciais do desenvolvimento, é desejável que o projetista possa avaliar as soluções candidatas de forma antecipada, utilizando o mesmo nível de abstração em que o sistema está sendo especificado.

Entretanto, a etapa de exploração de espaço de projeto ainda se mostra ser um grande desafio e gargalo ao projeto de sistemas embarcados (GRIES, 2004), principalmente por ela estar amarrada à presença e conhecimento da plataforma de execução, o que retarda sua execução e o próprio desenvolvimento. Por outro lado, técnicas conhecidas da engenharia de software que poderiam antecipar este passo ainda não estão completamente definidas para este domínio.

Dessa forma, se ainda fazem necessárias técnicas de análise e/ou estimação de projeto que em um ambiente de desenvolvimento mais abstrato possam auxiliar o projetista a tomar melhores decisões nas etapas inicias de projeto, garantindo assim, as funcionalidades (requisitos funcionais) e os requisitos não funcionais do sistema embarcado em um ambiente de desenvolvimento mais abstrato e ainda, que o produto final seja concluído em um curto tempo de projeto para atender o *time-to-market* .

Entretanto, a predição do comportamento do software (produto final) em execução, nas fases iniciais do processo de desenvolvimento é um tópico pouco explorado. Pode-se atribuir isso ao fato de que no domínio da engenharia de software, normalmente tais atributos não são levados em consideração, como por exemplo, consumo de energia e desempenho. Entretanto, no domínio dos sistemas embarcados esses atributos tornam-se de extrema importância, ainda mais com o grande aumento da complexidade do software embarcado, pois se for possível melhorar o processo de tomada de decisões nas etapas iniciais do projeto será possível economizar tempo, esforço, recursos financeiros, entre tantos outros, comparados aos casos em que um software não atenda as restrições de hardware depois de “pronto” e necessite ser refatorado.

Figura 1.1 – Aumento da complexidade do Software Embarcado



Fonte: (EBERT and JONES, 2009)

1.1 OBJETIVOS

O objetivo deste trabalho é permitir uma exploração do espaço de projeto antecipada através da estimação dos requisitos não funcionais desde as fases iniciais do ciclo de desenvolvimento de software. A técnica apresentada neste trabalho propõe a criação de modelos preditivos capazes de estimar os requisitos não funcionais antecipadamente, com o intuito de guiar o projetista de software embarcado a fazer melhores escolhas, principalmente durante a etapa de modelagem do sistema.

1.2 METODOLOGIA

Para tal, será proposto o uso de métricas de software estáticas para predição/estimação dos requisitos não funcionais (métricas estáticas e dinâmicas). As métricas de software serão extraídas em três diferentes níveis do projeto:

- Modelagem – Métricas de Diagrama de Classes
- Codificação – Métricas de Código Fonte

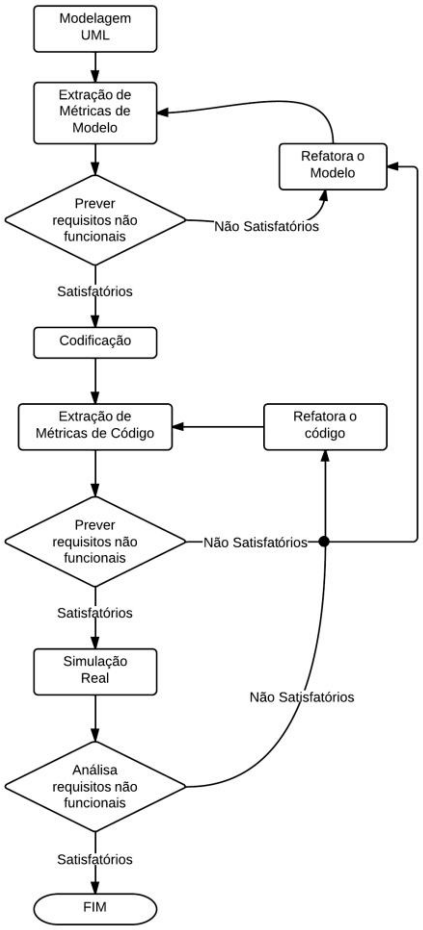
- Simulação – Métricas de Execução no Hardware

Então, após ter uma base de dados sobre diferentes etapas do desenvolvimento de software, será aplicado um processo de descoberta de conhecimento sobre esse conjunto de dados que permita relacionar as métricas da etapa de modelagem com as de codificação e principalmente com as de execução.

Após o momento em que tiverem sido encontradas tais relações, será possível estimar os requisitos não funcionais já nas etapas inicias do projeto. Com isso o projetista do software embarcado será capaz de fazer melhores escolhas ou realizar algumas alterações no projeto de forma antecipada, o que implicara em inúmeras vantagens como, por exemplo, diminuir o preço do desenvolvimento de software, acelerar o *time-to-market*, entre tantas outras.

Um exemplo de como essa abordagem poderá ajudar o projetista de software é demostrado na Figura 1.2.

Figura 1.2 – Fluxograma de análises dos requisitos não funcionais



Fonte: Próprio autor

1.3 ORGANIZAÇÃO DO TEXTO

O texto desta dissertação encontra-se organizado como indicado nos parágrafos a seguir.

No capítulo 2 será apresentado o embasamento teórico necessário para entendimento deste trabalho, onde serão abordados os seguintes tópicos: métricas de software; descoberta de conhecimento em base de dados; análise de regressão; e trabalhos relacionados.

No capítulo 3 serão apresentadas as duas abordagens aplicadas durante esse trabalho de mestrado: correlação de Pearson e KDD (*Knowledge-Discovery in Databases*).

Já o capitulo 4 apresentará os resultados experimentais para abordagem escolhida.

Finalmente, no capitulo 5 são apresentadas as conclusões juntamente com os trabalhos futuros que podem ser realizados para dar continuidade a este trabalho.

2 REFERENCIAL TEÓRICO

2.1 MÉTRICAS DE SOFTWARE

Uma métrica de software é a medição de um atributo (propriedade) de uma determinada entidade (produto, processo ou recursos) do software. Métricas de software podem ser usadas para estimar os custos do projeto, para gerenciar ou estimar recursos, para avaliar a eficácia dos métodos de programação e a confiabilidade de um sistema, avaliar a complexidade do programa e esforço de programação, correção, testabilidade, manutenção, flexibilidade, precisão, custo, e assim por diante (COOK, 1982).

Segundo Sommerville (2007), uma métrica de software é qualquer tipo de medição que se refira a um sistema de software, processo ou documentação relacionada. Desta forma, as métricas podem ser aplicadas em qualquer parte do ciclo de vida de desenvolvimento de software, bem como a seus artefatos, como por exemplo, modelos, código fonte, documentação, casos de teste, etc.

Em outras palavras, uma métrica de software é uma medição do software, isto é, uma medida do nível ou grau para o qual o produto possui ou apresenta certa qualidade, propriedade ou atributo. A grande importância da extração das métricas está no fato de fornecerem uma visão quantitativa do software e seu desenvolvimento, pois deste modo elas podem ser usadas para melhorar e refinar o produto (BOEHM, BROWN, *et al.*, 1978).

Presman (2006), propõe um processo de medição dividido em 5 etapas a serem executadas em sequência: formulação, coleta, análise, interpretação e realimentação. Na etapa de formulação as medidas e métricas são definidas para a representação do que está sendo considerado (PRESMAN, 2006). Durante a coleta são utilizados mecanismos para acumular dados necessários para derivação das métricas formuladas. Na análise é feito o cálculo de métricas através de ferramentas matemáticas. Na interpretação devem ser avaliados os resultados em um esforço para ganhar profundidade na visão da qualidade de representação. Por fim a realimentação ou *feedback* faz recomendações derivadas da interpretação das métricas de produto. Características semelhantes de processo de medição também podem ser encontradas em (SOMMERVILLE, 2007).

Algumas medidas podem ser obtidas automaticamente por ferramentas enquanto outras precisam ser coletadas através da observação e análise do software e seu ambiente. Na

sequência, as métricas são validadas para ver como elas refletem a qualidade do que está sendo medido. Por fim, as métricas podem ser rejeitadas, melhoradas, ou aceitas (COOK, 1982).

Toda métrica deve possuir algumas propriedades que são interessantes para que haja uma boa avaliação dos resultados da medição, a saber:

- Facilmente calculada, entendida e testada;
- Passível de estudos estatísticos;
- Expressa em alguma unidade;
- Obtida o mais cedo possível no ciclo de vida do software;
- Passível de automação;
- Repetível e independente do observador;
- Sugere uma estratégia de melhoria;

É de suma importância observar que estas propriedades são a essência das métricas. Medir por medir é uma grande perda de tempo, toda métrica deve ser completamente compreendida antes de ser aplicada, sob pena de se tornar inútil (PRESSMAN, 2011).Uma característica interessante é sugerir uma estratégia de melhoria, o analista que criar uma métrica deve pensar em como os resultados devem ser compilados a fim de aprimorar a qualidade do software.

As métricas disponíveis atualmente devem ser usadas como guia para o desenvolvimento e manutenção de software (FENTON and PFLEEGER, 1998). Elas não devem ser usadas como medidas rígidas e indiscutíveis que substituam por total o julgamento humano pois, segundo Cook (1882), não existe um número único e "mágico" que pode ser associado com o produto de software. As necessidades do usuário irão determinar quais as características do software são consideradas importantes. Assim, um dado produto pode ser visto de forma diferente por diferentes utilizadores.

Métricas de software podem ser classificadas quanto ao âmbito da sua aplicação, quanto ao critério utilizado na sua determinação e quanto ao método de obtenção da medida (MILLS, 1988).

Quanto aos critérios, as métricas são diferenciadas em métricas *objetivas* e métricas *subjetivas*. As objetivas são obtidas através de regras bem definidas, sendo a melhor forma de possibilitar comparações posteriores consistentes. Assim, os valores obtidos por elas deveriam ser sempre os mesmos, independentemente do instante, condições ou indivíduo que os determinam. A obtenção dessas métricas é passível de automatização. Já as métricas subjetivas

podem partir de valores, mas dependem de um julgamento humano e, portanto, dificultam as comparações e a reprodutibilidade das medidas. Por exemplo, contar defeitos é objetivo, classificar o nível do defeito é subjetivo.

Outra maneira de classificar as métricas de software é quanto ao método de obtenção, dividindo-as em *primitivas* ou *compostas*. As métricas primitivas são aquelas que podem ser diretamente observadas em uma única medida, como o número de linhas de código, erros indicados em um teste de unidade ou ainda o tempo total de desenvolvimento de um projeto. Por outro lado, as métricas compostas são as combinações de uma ou mais medidas como o número de erros encontrados a cada mil linhas de código ou ainda o número de linhas de teste por linha de código.

Entretanto, a classificação mais ampla é baseada no objeto da métrica, subdividindo-as em *métricas de produto*, que medem a complexidade e tamanho final do programa ou sua qualidade (confiabilidade, manutenibilidade etc.), e em *métricas de processo*, que se referem ao processo de concepção e desenvolvimento do software, medindo, por exemplo, o processo de desenvolvimento, tipo de metodologia usada ou tempo de desenvolvimento, entre outras (MILLS, 1988).

As métricas de processo são dados quantitativos sobre o processo de desenvolvimento do software, tais como o tempo necessário para realizar alguma atividade (por exemplo, o tempo necessário para desenvolver casos de teste de um programa). Humphrey (1989) argumenta que a medição de processos e atributos do produto é essencial para a melhoria do processo. Ele também sugere que a medida tem um importante papel a desempenhar na melhoria do desempenho individual dos membros de uma equipe, onde as pessoas tentam se tornar mais produtivas (HUMPHREY, 1989).

Métricas de processo podem ser utilizadas para avaliar se a eficiência de um processo foi melhorada. Por exemplo, o tempo e esforço dedicado ao teste de software podem ser monitorados. Melhorias efetivas para o processo de teste devem reduzir o esforço e/ou tempo de teste. No entanto, as medições do processo por si só não podem ser usadas para determinar se a qualidade do produto tem melhorado (MILLS, 1988).

Segundo Sommerville, três tipos de métricas de processo podem ser extraídas (SOMMERVILLE, 2007):

1. *O tempo necessário para um processo específico ser finalizado*. Este pode ser o tempo total atribuído ao processo, o tempo de calendário, o tempo gasto no processo por engenheiros particulares, e assim por diante.
2. *Os recursos necessários para um determinado processo*. Esses recursos podem ser o esforço total em pessoas por dia, custos de viagem ou recursos computacionais.
3. *O número de ocorrências de um determinado evento*. Esses eventos podem ser o número de defeitos descobertos durante inspeção de código, o número de mudanças de requisitos solicitados, a média do número de linhas de código modificadas em resposta a uma mudança de requisitos.

Por outro lado, as *métricas de produto* são métricas preditivas que são utilizadas para medir os atributos internos de um sistema de software em qualquer etapa de seu desenvolvimento e podem medir o tamanho, complexidade, abstração, desempenho, portabilidade, entre tantos outros fatores do produto. Logo, como neste trabalho buscamos avaliar um produto de software tanto em relação à sua qualidade interna quanto em relação à sua adequação a uma determinada plataforma de hardware, este trabalhou acabou por utilizar este tipo de métrica. As métricas de produtos ainda podem ser divididas em duas classes: *métricas dinâmicas* e *métricas estáticas*.

2.1.1 Métricas de Produto Dinâmicas

As métricas de produto dinâmicas são medições feitas a partir do programa (produto) em execução em uma determinada plataforma de hardware. Essas métricas podem ser coletadas desde os testes iniciais de um sistema, até após o sistema ter entrado em uso.

Diversas métricas dinâmicas foram propostas na literatura (CHHABRA and GUPTA, 2010) (DUFOUR, DRIESEN, *et al.*, 2002) (DUFOUR, DRIESEN, *et al.*, 2003) (YACOUB, AMMAR and ROBINSON, 1999) e estão relacionadas ao consumo de recursos e/ou custo de execução do programa. Assim, por exemplo, tamanho total da aplicação, número de objetos criados, número de chamadas de métodos, quantidade de memória dinâmica atribuída pelo programa, entre tantas outras.

Pensando no domínio dos sistemas embarcados, outras métricas dinâmicas de interesse estão relacionadas ao consumo dos recursos de hardware: consumo de memória, uso da unidade central de processamento (CPU - Central Processing Unit), comunicação entre CPU e memória,

energia, etc. Neste trabalho, estamos interessados nas métricas dinâmicas diretamente relacionadas ao uso eficiente da plataforma de hardware, conforme detalhado abaixo.

O consumo de memória é dividido na utilização da memória principal (Random Access Memory – RAM), e caches. Memórias cache aceleraram o acesso a um determinado dado ou código, mas consomem mais energia quando comparadas com memórias RAM, entretanto compensam pelo grande ganho de desempenho (HENNESSY and PATTERSON, 2011). Caches podem ser divididas em níveis (por exemplo, L1, L2 e L3), e ainda podem ser divididas entre dados e instruções, de acordo com seu tamanho e proximidade com a CPU. Quando a informação não é encontrada na cache mais próxima ocorre um *cache miss* e as informações devem ser buscadas no próximo nível de cache ou na memória principal. Entretanto, este processo de buscar a informação no próximo nível de memória aumenta muito o tempo de execução e o consumo de energia. Logo, quanto menor o número de *cache miss*, melhor. Uma métrica comumente usada é o número de *misses* que ocorrem durante em uma execução, esse número pode ser mensurado em qualquer nível da cache, por exemplo, nesse trabalho foram mensurados o número misses na cache L2 e na cache L1 dividida em dados e instruções, conforme a Tabela 2.1.

A CPU, por sua vez, implementa algumas estratégias para aumentar a sua velocidade de operação e evitar comunicação extra. Uma dessas estratégias é despachar o maior número de instruções possíveis em *pipeline*, de tal forma que, em cada ciclo, uma instrução seja completada. Se o fluxo do *pipeline* é mantido, o tempo de execução total da aplicação é reduzido. Entretanto, manter o fluxo de uma instrução por ciclo não é uma tarefa trivial quando existem instruções de desvios condicionais ou outras condições de dependência de dados. Uma maneira de manter o fluxo do *pipeline* consiste em tentar prever o resultado da condição em um desvio condicional e buscar a instrução que tem maior probabilidade de suceder a instrução de desvio. Isto é implementado no processador por um circuito digital chamado b*ranch predictor*. Se a predição estiver correta, o fluxo do pipeline é mantido e o tempo de execução é reduzido. Caso contrário, ignoram-se os resultados já computados por engano e o pipeline é reiniciado com a instrução pós-desvio correta. Logo, quanto menor o número de previsões de desvios incorretos, menor será o tempo de execução de uma aplicação e menor será seu consumo energético. Também é possível mensurar a quantidade de acertos (*Predicted Branches*) e erros *Missed Branches*) da previsão, conforme a Tabela 2.1.

Neste trabalho de mestrado, outra métrica de hardware que foi considerada é o número de instruções por ciclo (IPC do inglês, *Instructions per Cycles*) que tem uma relação direta com o

desempenho da aplicação, pois quanto mais instruções for possível executar por ciclo, considerando a mesma frequência, melhor será o desempenho. A última métrica que será considerada nesse trabalho é o consumo de energia, que no âmbito dos sistemas embarcados é de suma importância, pois muitos sistemas embarcados são alimentados através da energia de baterias, logo os projetistas devem visar sempre o menor consumo de energia possível.

.

Tabela 2.1 – Métricas Dinâmicas

Métrica	Descrição
Predicted Branches (PB)	Número de branches preditos corretamente.
Missed Branches (MB)	Número de branches não preditos corretamente.
Instructions per Cycles (IPC)	Razão entre o total de instruções executadas pelo total de ciclos em uma aplicação.
Instruction Cache Misses (ICM)	Número de misses na cache de instruções L1.
Data Cache Misses (DCM)	Número de misses na cache de dados L1.
L2 Cache Misses (L2M)	Número total de misses na cache L2
Energy (E)	Total de energia consumida pela aplicação

Fonte: Autor

2.1.2 Métricas de Produto Estáticas

Métricas de produto estáticas são coletadas através de medições efetuadas sobre representações do sistema, tais como, códigos fonte ou modelos UML, documentação, etc. Elas ainda podem ser classificadas como métricas de complexidade, tamanho, orientação a objetos, entre tantas outras, dependendo de cada autor. A seguir são apresentadas algumas métricas de produto estáticas.

2.1.2.1 Métricas de Tamanho

Algumas métricas foram desenvolvidas para tentar quantificar o tamanho e auxiliar as medições na fase de concepção do software. Alguns exemplos de métricas que medem o tamanho do software são apresentadas a seguir. Devido ao fato de muitas métricas serem

amplamente conhecidas por suas abreviaturas, que derivam de seus nomes originais em inglês, os nomes das métricas que são apresentadas a seguir foram mantidos em inglês.

2.1.2.1.1 Lines of *Code (LOC)*

O fato mais importante sobre a métrica de *Linhas de Código* é entender que embora seu cálculo aparentemente seja simples, na verdade ela exige a interpretação dos dados com um certo nível de semântica.

Quando a programação era feita em *assembly* e cada linha representava uma instrução do processador, a contagem de linhas era trivial. Já nos tempos atuais, com a presença massiva de linguagens de alto nível de abstração nos projetos de software, essa antiga correspondência um para um se perdeu, e uma linha de código pode resultar em várias instruções. Diferentes linguagens de programação, ou mesmo diferentes comandos da mesma linguagem de programação podem levar a grandes variações nessa métrica.

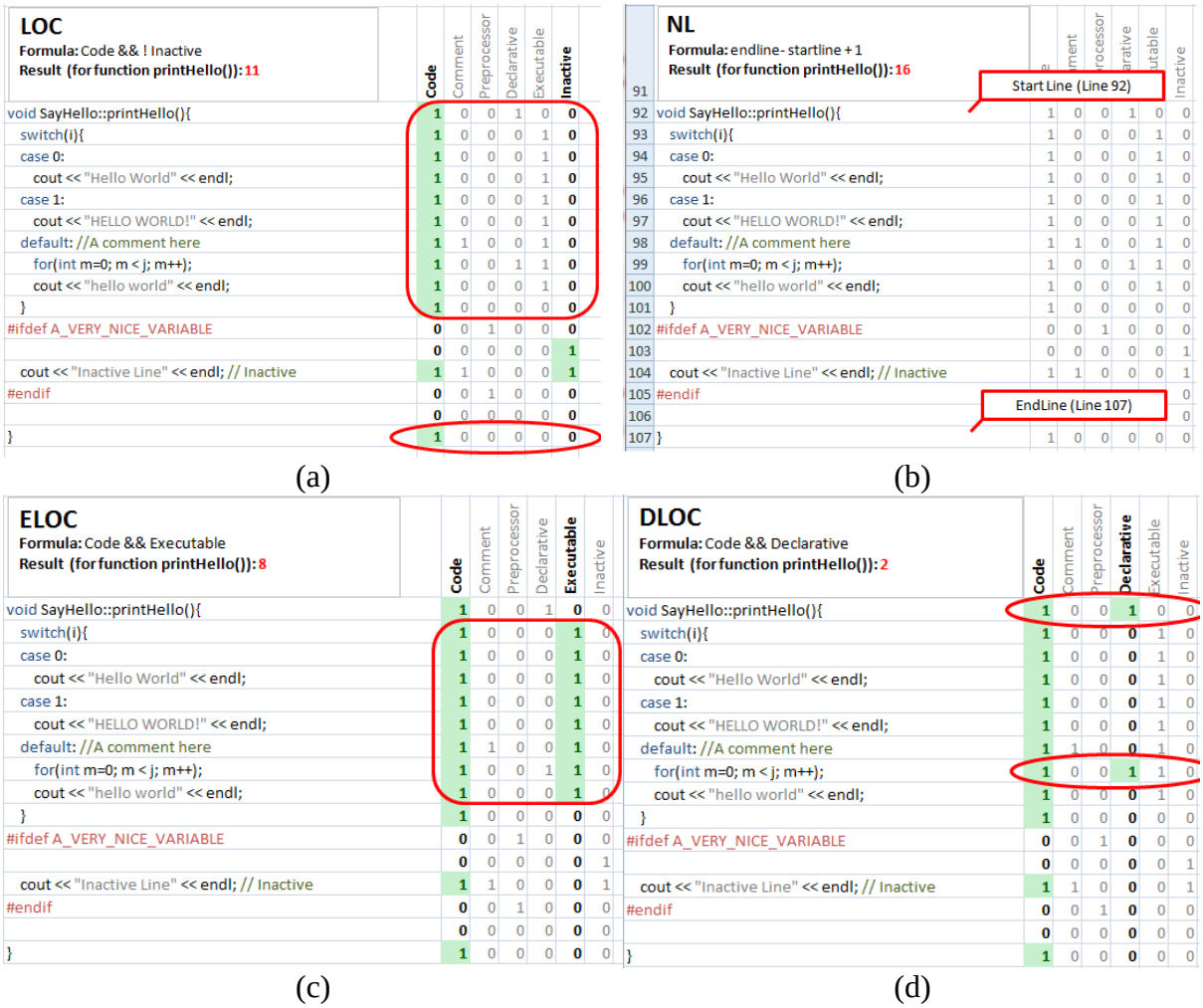
Por defnição são contadas apenas as linhas executáveis (ver Figura 2.1(a)), ou seja, são excluidas linhas em branco e comentários. Entretanto, existem diversas variações no modo de realização dessa contagem (JONES, 1986), entre elas:

- *Number of Lines (NL)*, conta todas as linhas, conforme a Figura 2.1 (b);
- *Executable Lines of Code (ELOC)*, conta as linhas contendo código fonte executável, conforme a Figura 2.1 (c);
- *Declarative Lines of Code (DLOC)*, conta as linhas contendo declaração de dados, conforme ar Figura 2.1(d);
- Contar as linhas executáveis juntamente com as linhas de declaração de dados;
- Contar as linhas executáveis juntamente com as linhas de declaração de dados e linhas de comentários;
- Contar as linhas executáveis juntamente com as linhas de declaração de dados, linhas de comentários e comandos de controle da linguagem;

Atualmente, devido ao crescente tamanho dos projetos de software, essa métrica tem aparecido renomeada como KLOC (*Kilo Lines of Code*) ou MLOC (*Million Lines of Code*) indicando, respectivamente, a contagem em milhares ou milhões de linhas de código. No entanto, é importante ressaltar que as mesmas metodologias de contagem são aplicadas a essas variações da métrica, tendo como única diferença o multiplicador indicativo de uma maior ordem de grandeza nos dados levantados

Figura 2.1 – Exemplo de Métricas de Tamanho

(a) LOC; (b) NL; (c) ELOC; (d) DLOC



Fonte: (TOOLWORKS, 2014)

2.1.2.1.2 Number of semicolons

Esta métrica apenas mensura o número total de pontos e vírgulas em um código fonte. Entretanto, dependendo da linguagem de programação, pode ser uma métrica mais representativa que LOC. Por exemplo, na linguagem de programação C, onde todas instruções terminam com pontos e virgulas, essa métricas representaria o número de instruções executadas em C.

2.1.2.2 Métricas de Complexidade

Métricas de complexidade são uma forma objetiva de medir a complexidade de uma parte do software. A alta complexidade deve ser evitada pois torna o sistema mais suscetível a erros e prejudica a compreensão do código, dificultando assim a manutenção. Dentre as principais métricas de complexidade, algumas são citadas a seguir.

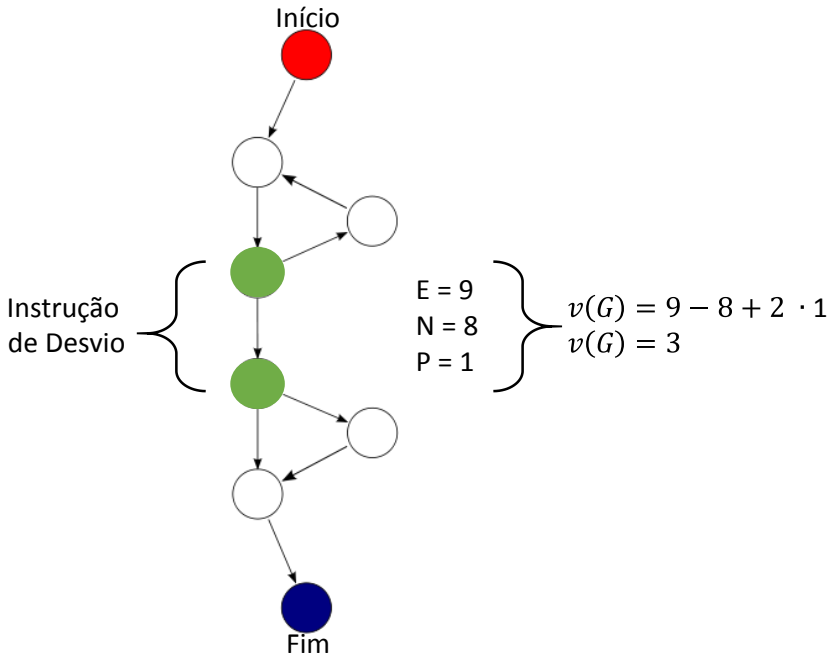
2.1.2.2.1 Cyclomatic Complexity (CC)

A complexidade ciclomática, também conhecida como V(G), é provavelmente a métrica de complexidade mais amplamente utilizada na engenharia de software. Definida por Thomas McCabe (MCCABE, 1976), esta métrica considera a lógica de controle em um procedimento, com o propósito de expressar as capacidades de teste e manutebilidade do código fonte de um dado programa.

Essa métrica pode ser representada através de um grafo de fluxo de controle, onde os nós representam uma ou mais instruções sequenciais e os arcos orientados indicam o sentido do fluxo de controle entre várias instruções. A complexidade ciclomática *v* de um determinado grafo *G* pode ser calculada através de uma fórmula da teoria dos grafos, conforme mostrado na Equação 2.1, onde e é o número de arestas, n é o número de nós do grafo e p o número de componentes conectados no grafo. Um exemplo de aplicação desta equação pode ser visto na Figura 2.2.

$$v(G) = e - n + 2 \cdot p \qquad \text{Equação 2.1}$$

Figura 2.2 – Complexidade Ciclomática



Fonte: Autor

Alternativamente, uma outra maneira de calcular a complexidade ciclomática (apenas de método) é contar o número de decisões causados por desvios condicionais (If..ElseIf..Else, Case, While, For, ...) e somar um, conforme a Equação 2.2. A Figura 2.3 ilustra esse processo, onde as colunas em negrito são todas as instruções de desvios que devemos levar em consideração no cálculo da complexidade ciclomática.

$$v(G) = Numero\ de\ Decisões + 1 \qquad \text{Equação 2.2}$$

McCabe sugere que a complexidade ciclomática de um módulo não deva passar de 10. Ele salienta, entretanto, que o este valor deve ficar idealmente entre 3 e 7 para que os métodos fiquem bem estruturados, legíveis e de fácil manutenção e teste (MCCABE, 1976). A Tabela 2.2 apresenta todos os intervalos dos valores possíveis para CC e suas classificações.

Figura 2.3 – Complexidade Ciclomática (CC)

Cyclomatic **Formula:** case,catch,do,for,if,?,while+1 **Result (for function cyclomaticDemo()):** 8	AND	OR	CATCH	DO	FOR	IF	?	WHILE	SWITCH	CASE		
void cyclomaticDemo(){	0	0	0	0	0	0	0	0	0	0		
bool a=true,b=true,c=true;	0	0	0	0	0	0	0	0	0	0		
	0	0	0	0	0	0	0	0	0	0		
if(a || (b && c)){	1	1	0	0	0	1	0	0	0	0		
while(a? b : c){	0	0	0	0	0	0	1	1	0	0		
for(int i=0; i < 10; i ++){	0	0	0	0	1	0	0	0	0	0		
switch(i){	0	0	0	0	0	0	0	0	1	0		
case 1:	0	0	0	0	0	0	0	0	0	1		
case 2:	0	0	0	0	0	0	0	0	0	1		
cout<<i<<endl;	0	0	0	0	0	0	0	0	0	0		
break;	0	0	0	0	0	0	0	0	0	0		
case 5:	0	0	0	0	0	0	0	0	0	1		
break;	0	0	0	0	0	0	0	0	0	0		
default:	0	0	0	0	0	0	0	0	0	0		
cout<<i<<endl;	0	0	0	0	0	0	0	0	0	0		
} } } } }	0	0	0	0	0	0	0	0	0	0		
	1	1	0	0	1	1	1	1	1	3	+ 1 =	8

Fonte: (TOOLWORKS, 2014)

Tabela 2.2 – Valores de complexidade ciclomática

CC	Tipo de método	Risco
1-4	Um método simples	Baixo
5-10	Um procedimento bem estruturado e estável	Baixo
11-20	Um procedimento mais complexo	Moderado
21-50	Um processo complexo, alarmante	Alto
>50	Um método propenso a erros, extremamente problemático e instável	Muito alto

Fonte: (MCCABE, 1976)

2.1.2.2.2 Strict Cyclomatic Complexity (CC2)

A complexidade ciclomática estrita estende a complexidade ciclomática (CC) incluindo na contagem do número de decisões o número de operadores booleanos presentes em cada decisão. Sempre que um operador booleano (AND, OR, XOR, ...) é encontrado dentro de uma declaração condicional, CC2 é incrementada, de forma a somar mais um a sua complexidade. A Figura 2.4 apresenta um exemplo de como é feito o cálculo da CC2.

O raciocínio por trás da CC2 é que um operador booleano aumenta a complexidade interna de uma decisão. Logo, CC2 conta o número "real" de decisões, independentemente de elas aparecerem como uma única instrução condicional ou divididas em várias declarações. Usar operadores booleanos para combinar decisões pode ser um artifício para diminuir o valor de CC. Entretanto, a complexidade em si não estará diminuindo, mas apenas sendo mascarada. Por outro lado, a CC2 se torna imune a este tipo de reestruturação.

Figura 2.4 – Complexidade Ciclomática Estrita

CyclomaticStrict **Formula:** and,or,case,catch,do,for,if,?,while+1 **Result (for function cyclomaticDemo()):** 10	AND	OR	CATCH	DO	FOR	IF	?	WHILE	SWITCH	CASE	
void cyclomaticDemo(){	0	0	0	0	0	0	0	0	0	0	
bool a=true,b=true,c=true;	0	0	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	0	0	
if(a || (b && c)){	1	1	0	0	0	1	0	0	0	0	
while(a? b : c){	0	0	0	0	0	0	1	1	0	0	
for(int i=0; i < 10; i ++){	0	0	0	0	1	0	0	0	0	0	
switch(i){	0	0	0	0	0	0	0	0	1	0	
case 1:	0	0	0	0	0	0	0	0	0	1	
case 2:	0	0	0	0	0	0	0	0	0	1	
cout<<i<<endl;	0	0	0	0	0	0	0	0	0	0	
break;	0	0	0	0	0	0	0	0	0	0	
case 5:	0	0	0	0	0	0	0	0	0	1	
break;	0	0	0	0	0	0	0	0	0	0	
default:	0	0	0	0	0	0	0	0	0	0	
cout<<i<<endl;	0	0	0	0	0	0	0	0	0	0	
} } } } }	0	0	0	0	0	0	0	0	0	0	
	1	1	0	0	1	1	1	1	1	3	+1= 10

Fonte: (TOOLWORKS, 2014)

2.1.2.2.3 Modified Cyclomatic Complexity (CC3)

A complexidade ciclomática modificada é praticamente igual à métrica CC tradicional. A única diferença é que cada bloco *Switch* (ou *Select*) é contado apenas como um ramo (uma decisão), e não como várias decisões. Nesta variação, a instrução *Switch Case* é tratada como se fosse uma única grande decisão. Isto leva a valores consideravelmente menores de complexidade para métodos com grandes instruções Switch Case. Entretanto, em muitos casos, as instruções *Switch Case* são simples o suficiente para serem consideradas apenas como uma decisão, o que justifica o uso de CC3 nesses casos. Um exemplo de como é feito o cálculo da CC3 é apresentado na Figura 2.5.

Figura 2.5 – Complexidade Ciclomática Modificada (CC3)

CyclomaticModified
Formula: catch,do,for,if,?,while,switch+1
Result (for function cyclomaticDemo()): 6

	AND	OR	CATCH	DO	FOR	IF	?	WHILE	SWITCH	CASE		
void cyclomaticDemo(){	0	0	0	0	0	0	0	0	0	0		
bool a=true,b=true,c=true;	0	0	0	0	0	0	0	0	0	0		
	0	0	0	0	0	0	0	0	0	0		
if(a || (b && c)){	1	1	0	0	0	1	0	0	0	0		
while(a? b : c){	0	0	0	0	0	0	1	1	0	0		
for(int i=0; i < 10; i ++){	0	0	0	0	1	0	0	0	0	0		
switch(i){	0	0	0	0	0	0	0	0	1	0		
case 1:	0	0	0	0	0	0	0	0	0	1		
case 2:	0	0	0	0	0	0	0	0	0	1		
cout<<i<<endl;	0	0	0	0	0	0	0	0	0	0		
break;	0	0	0	0	0	0	0	0	0	0		
case 5:	0	0	0	0	0	0	0	0	0	1		
break;	0	0	0	0	0	0	0	0	0	0		
default:	0	0	0	0	0	0	0	0	0	0		
cout<<i<<endl;	0	0	0	0	0	0	0	0	0	0		
} } } } }	0	0	0	0	0	0	0	0	0	0		
	1	1	0	0	1	1	1	1	1	3	+1=	6

Fonte: (TOOLWORKS, 2014)

2.1.2.2.4 Essential Complexity (ev(G))

O cálculo da complexidade essencial é feito removendo-se todos os subgrafos estruturados (aqueles que têm um único ponto de entrada e um único ponto de saída) do grafo de controle e então calculando-se a complexidade ciclomática do grafo resultante. Um grafo que tem um único ponto de entrada e um único ponto de saída é reduzido então para um subgrafo com complexidade igual a 1. Qualquer ramificação para dentro ou para fora de um laço ou de uma decisão fará com que o grafo seja não redutível e terá complexidade essencial maior que 2 (MCCABE and BUTLER, 1989).

2.1.2.2.5 Max Nesting (MN)

Esta métrica avalia a profundidade de blocos aninhados dentro de um método e apresenta, como resultado, a maior profundidade encontrada. Em outras palavras, essa métrica também é um indicador de complexidade, pois quando inúmeros comandos de controle (e.g. If, For, While, ...) são aninhados, a execução se torna mais complexa, assim como a compreensão e a legibilidade do código, o que também irá afetar a manutenibilidade. Métodos que apresentam

MN acima de seis necessitam de refatoração para facilitar a sua manutenibilidade (SOMMERVILLE, 2010).

2.1.2.3 Métricas de Orientação a Objetos

Como o paradigma da orientação a objetos usa entidades e não algoritmos como componentes fundamentais, a abordagem das métricas de código-fonte para programas orientados a objetos deve ser diferente. Além de tamanho e complexidade, em um software orientado a objetos é desejável medir o uso da herança e o grau de interdependência entre as entidades, por exemplo. Segundo (PRESMAN, 2006), Chidamber e Kemerer propuseram um dos conjuntos mais amplamente conhecidos de métricas para software orientado a objeto, o conjunto CK (CHIDAMBER and KEMERER, 1994), composto por seis métricas. A seguir serão apresentadas essas seis métricas do conjunto CK bem como outras métricas de orientação a objetos.

2.1.2.3.1 Weighted Methods pes Class (WMC)

A métrica WMC, ou métodos ponderados por classe, indica a complexidade individual de uma classe. Esta métrica representa o somatório das complexidades dos métodos da classe. Normalmente, assume-se a complexidade ciclomática (CC) como métrica de complexidade dos métodos. A Equação 2.3 apresenta a fórmula de WMC proposta por (CHIDAMBER and KEMERER, 1994), onde *n* é o número de métodos da classe e *C* é a complexidade ciclomática dos métodos.

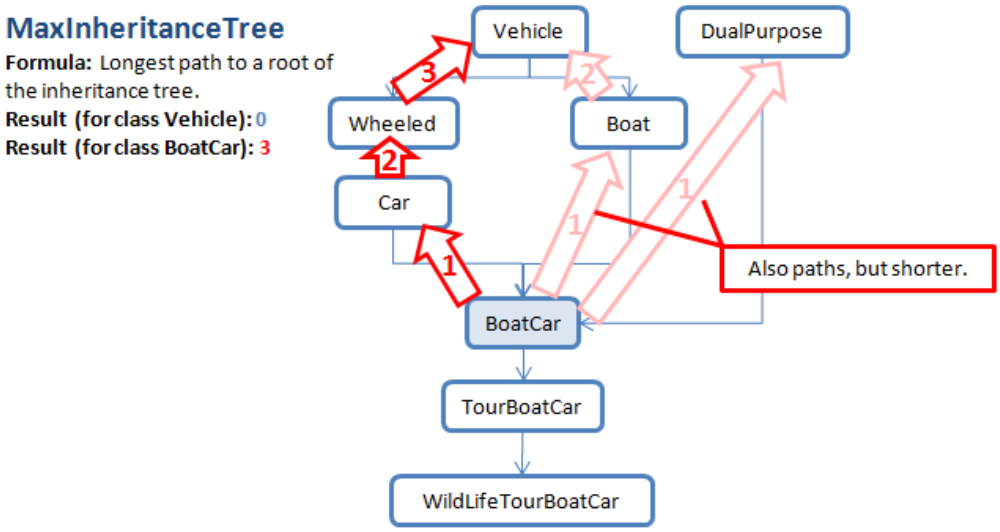
$$WMC = \sum_{i=1}^{n} C_i \qquad \text{Equação 2.3}$$

O número de métodos de uma classe, bem como suas complexidades, pode ser um indicador de tempo e esforço necessário para o desenvolvimento e manutenção. Além disso, classes com muitos métodos tendem a causar grande impacto em suas subclasses, uma vez que seus descendentes herdam seus métodos. Por fim, classes com métodos muito complexos tendem a ser muito especializadas, dificultando, assim, seu reúso em projetos futuros. Por todas essas razões, o WMC deverá ser mantido o mais baixo possível (PRESSMAN, 2011).

2.1.2.3.2 Depth of the Inheritance Tree (DIT)

Profundidade da árvore de herança, também proposta por (CHIDAMBER and KEMERER, 1994), é a métrica que indica a distância máxima entre uma classe até a raiz da árvore de herança à qual ela pertence. A Figura 2.6 ilustra o como essa métrica aplicar-se-ia à classe "BoatCar" em uma dada hierarquia. À medida que a DIT cresce, é possível que classes de nível inferior herdem muitos métodos. Isso causa dificuldades potenciais quando se tenta prever o comportamento de uma classe. Uma hierarquia de classe profunda também leva a uma complexidade maior no projeto. Entretanto, analisando o lado positivo, grandes valores de DIT implicam que muitos métodos podem ser reutilizados.

Figura 2.6 – Exemplo da Métrica DIT



Fonte: (TOOLWORKS, 2014)

2.1.2.3.3 Number of Children (NOC)

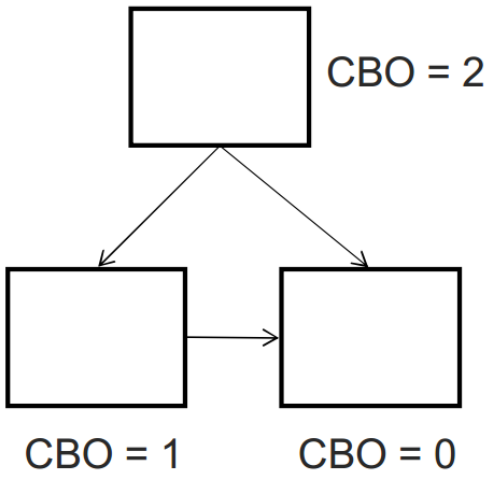
O número de classes filhas está relacionado às subclasses que estão imediatamente subordinadas a uma classe na hierarquia. Por exemplo, de acordo com a Figura 2.6, a classe "BoatCar" tem uma filha (NOC=1), enquanto a classe "Vehicle" tem duas filhas, ou seja, seu valor de NOC é igual a dois. Conforme cresce o número de classes herdeiras diretas, a reutilização aumenta, o que é um bom sinal. Por outro lado, à medida que o NOC aumenta, a

abstração representada pela classe pai pode ser diluída se algumas das classes filhas não forem herdeiros apropriados da classe pai (a necessidade de muitas especializações pode indicar, na verdade, que a classe pai não possui uma boa abstração). Conforme o NOC aumenta, a quantidade de testes também aumentará pois será necessário exercitar cada classe herdeira em seu contexto operacional. Essa métrica também foi proposta por (CHIDAMBER and KEMERER, 1994).

2.1.2.3.4 Coupling Between Objects *(CBO)*

Essa métrica mede o acoplamento entre objetos ou, em outras palavras, a conectividade de uma classe (CHIDAMBER and KEMERER, 1994). Uma classe A está acoplada a uma classe B quando a classe A utiliza métodos ou atributos da classe B. Um exemplo de como esta métrica se aplica pode ser visto na Figura 2.7. Segundo Pressman (2011), em geral, os valores de CBO para cada classe deverão ser mantidos os mais baixos possível.

Figura 2.7 – Exemplo da métrica CBO



Fonte: Autor

2.1.2.3.5 Response for a Class (RFC)

O conjunto de respostas de uma classe é um conjunto de métodos que podem potencialmente ser executados em resposta a uma mensagem recebida por um objeto daquela classe (CHIDAMBER and KEMERER, 1994). O RFC é número de métodos no conjunto de respostas. Segundo Pressman (2011), conforme RFC aumenta, o esforço de teste também aumenta porque

a sequência de testes cresce e adicionalmente, à medida que RFC aumenta, a complexidade geral do projeto da classe aumenta (PRESSMAN, 2011).

2.1.2.3.6 Lack of Cohesion in Methods (LCOM)

Definir uma métrica objetiva que indique a coesão de um módulo é complicado, pois implica a identificação da semântica de todas as operações e dados do módulo e do seu entorno. Por outro lado, identificar a presença de operações e/ou dados definidos fora do módulo em questão pode ser possível. Então, a métrica LCOM, como o próprio nome diz, é usada para medir a falta de coesão em métodos (CHIDAMBER and KEMERER, 1994). Valores grandes para LCOM indicam uma baixa coesão, enquanto valores baixos indicam uma alta coesão.

Em outras palavras, cada método dentro de uma classe acessa um ou mais atributos (também conhecidos como variáveis de instância). LCOM será o número de métodos que acessam um ou mais dos mesmos atributos. Segundo (CHIDAMBER and KEMERER, 1994), as classes que apresentam baixos valores para as métricas de avaliação de coesão devem ser divididas em classes menores e mais coesas, pois coesão entre os métodos de uma classe é uma propriedade desejável.

2.1.2.3.7 Métricas de Tamanho para OO

A seguir são apresentadas algumas métricas OO quantificam os projetos quanto ao número de classes, número de métodos, etc. São métricas simples referentes ao tamanho, onde seu cálculo consiste em contar o número de vezes que o artefato em questão aparece no projeto ou na classse ou nos métodos.

- *Number of Attributes (NOA)*

Essa métrica calcula o número de atributos de uma classe. O valor mínimo para esta métrica é zero e não existe um valor máximo para seu resultado. Porém, uma classe com muitos atributos pode indicar que ela lida com vários assuntos diferentes e tem muitas responsabilidades, o que é um indicativo de baixa coesão (LORENZ and KIDD, 1994).

- *Number of Classes (NC)*

A métrica número total de classes ou, simplesmente, número de classes, representa a contagem de classes, abstratas e concretas, em um escopo selecionado. Não existem intervalos ou limites sugeridos para os valores desta métrica. O número de classes de um projeto de software dependerá das necessidades específicas de cada projeto e da aplicação adequada, ou não, dos conceitos de desenvolvimento orientado a objetos (MARTIN, 2002).

- *Number of Interfaces (NI)*

Esta métrica indica o número de interfaces no projeto OO. Uma interface é um mecanismo capaz de criar uma camada extra de abstração em projetos orientados a objetos. Classes relacionadas à interface deverão implementar os métodos (serviços) definidos na interface (MARTIN, 2002).

- *Number of Packages (NOPK)*

Um pacote é um grupo de classes relacionadas e possivelmente cooperantes. Os pacotes facilitam o reúso e ajudam a resolver problemas de ambiguidade para classes com o mesmo nome. A métrica número de pacotes é definida como a contagem do número de pacotes existentes no escopo selecionado para análise, incluindo eventuais subpacotes (MARTIN, 2002).

- *Number of Methods (NOM)*

Em projetos onde o paradigma orientado a objetos é utilizado, os métodos, ou operações, são os componentes de uma classe responsáveis por descrever seu comportamento. A contagem de métodos de uma classe é uma forma simples de medida de complexidade. As classes com muitos métodos tendem a causar grande impacto em suas subclasses, uma vez que seus descendentes herdam seus métodos. Porém, classes com poucos métodos podem indicar um problema de projeto, ou seja, talvez seus atributos e métodos não sejam suficientes para constituir um novo tipo ou conceito (PRESMAN, 2006).

- *Number of Public Attributes (NOPA)*

Essa métrica, número de atributos públicos, mede o encapsulamento, pois os atributos de uma classe devem servir apenas às funcionalidades da própria classe. Portanto, boas práticas de programação recomendam que os atributos de uma classe devem ser privados e manipulados através dos métodos de acesso. Portanto, o número ideal para essa métrica é zero (LANZA and MARINESCU, 2006).

2.2 DESCOBERTA DE CONHECIMENTO EM BASE DE DADOS

A todo o momento, dados vêm sendo armazenados, formando grandes volumes de informação. Estes dados armazenados podem conter informações ocultas de grande relevância e, devido ao grande volume de dados, a extração destas informações não é uma tarefa trivial. Por isso, técnicas e ferramentas de auxílio aos analistas para a extração e análise de informações úteis de um grande repositório de dados são essenciais. O conjunto destas técnicas e ferramentas pertence a um domínio conhecido como Descoberta de Conhecimento em Banco de Dados (DCBD) ou ainda, na língua inglesa, como *Knowledge Discovery in Databases* (KDD).

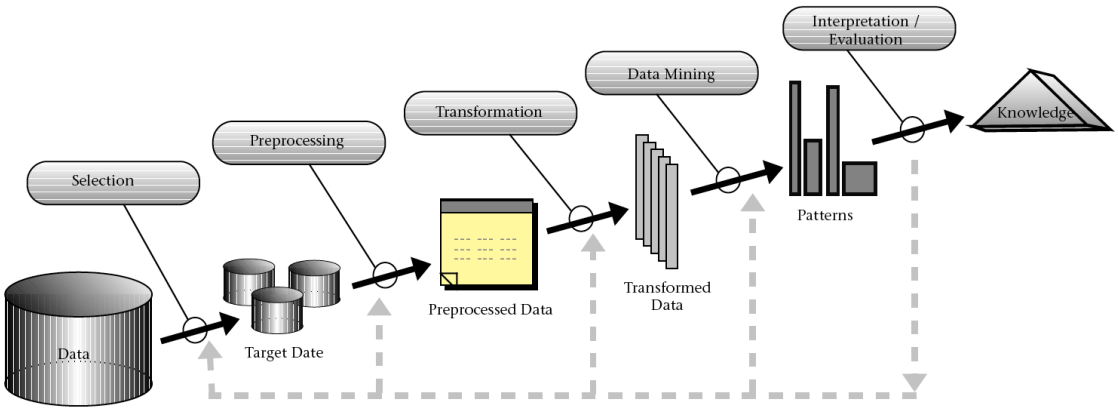
O KDD é um processo não trivial de identificação de padrões novos, válidos, potencialmente úteis e compreensíveis em dados (FAYYAD, PIATETSKY-SHAPIRO and SMYTH, 1996). Em outras palavras, é um processo que preocupa-se com o desenvolvimento de métodos e técnicas para que os dados façam sentido, concentrando-se no mapeamento dos dados brutos em modelos mais compactos (relatórios, gráficos), genéricos (identificação de modelos que descrevem os dados) ou úteis (modelos preditivos que estimem valores futuros) que os dados originais.

No núcleo deste processo, encontra-se a aplicação de técnicas para a identificação e extração de informações relevantes, ocultas nos dados. Estas técnicas são conhecidas como mineração de dados por identificarem padrões e conhecimentos relevantes para o negócio, ou seja, minerar os dados à procura de informações relevantes. Os mais diferentes tipos de dados podem ser minerados e, dependendo de suas caraterísticas, diversos algoritmos podem ser utilizados para mineração, como as regras de associação, classificação, *clustering*, entre outras.

O KDD é um processo iterativo composto de diversas etapas que envolvem a preparação dos dados, procura por padrões, avaliação e refinamento. O processo é iterativo, pois todas as

etapas estão conectadas, e também interativo, pois todas as etapas contêm tarefas e decisões a serem feitas pelos usuários, Caso a etapa de avaliação não seja satisfatória, de acordo com o processo, deve-se voltar para a etapa de preparação dos dados. A Figura 2.8 representa este fluxo.

Figura 2.8 – Etapas do processo de KDD



Fonte: (FAYYAD, PIATETSKY-SHAPIRO and SMYTH, 1996)

2.2.1 Etapas do KDD

Conforme ilustra a Figura 2.8 o processo de KDD é basicamente divido em cinco etapas: i) seleção; ii) pré-processamento; iii) transformação; iv) mineração de dados; v) avaliação/interpretação (pós-processamento).

A fase de seleção de dados é a primeira no processo de descobrimento de informação e possui impacto significativo sobre a qualidade do resultado final, uma vez que nesta fase é escolhido o conjunto de dados contendo todas as possíveis variáveis (atributos) e registros (observações) que farão parte da análise. O processo de seleção é bastante complexo, uma vez que os dados podem vir de uma série de fontes diferentes (*data warehouses*, planilhas, sistemas legados, etc.) e podem possuir os mais diversos formatos.

O pré-processamento ou limpeza dos dados é outra parte crucial no processo de KDD, pois a qualidade dos dados vai determinar a eficiência dos algoritmos de mineração. Nesta etapa

devem ser realizadas tarefas que eliminem dados redundantes e inconsistentes, recuperem dados incompletos e avaliem possíveis dados discrepantes ao conjunto, chamados de *outliers*.

A transformação dos dados é a fase do KDD que antecede a fase de mineração de dados. Após serem selecionados, limpos e pré-processados, os dados necessitam ser armazenados e formatados adequadamente para que os algoritmos possam ser aplicados. Além disto, nesta fase, se necessário, é possível obter dados faltantes através da transformação ou combinação de outros, são os chamados "dados derivados". Um exemplo de um dado que poderia ser calculado a partir de outro é uma métrica composta.

Após a realização das fases anteriores, a mineração dos dados é iniciada. Esta fase é a mais importante do KDD, sendo realizada através da escolha da(s) técnica(s) e do(s) algoritmo(s) mais compatível(eis) com o objetivo da extração, a fim de encontrar padrões nos dados que sirvam de subsídios para descobrir conhecimentos ocultos. Na subseção 2.2.2, um breve aprofundamento desta fase será realizado.

Por fim, a última etapa é a fase que identifica se os padrões extraídos na etapa de *data mining*, são interessantes ao critério estabelecido pelo negócio. Caso não seja encontrado nenhum padrão, deve-se voltar às fases anteriores para novas iterações, até que realmente seja extraído algum conhecimento útil. Então, este conhecimento extraído é validado, sendo incorporado a um sistema inteligente, que é utilizado pelo usuário final como apoio a algum processo de tomada de decisão.

2.2.2 Data Mining

Mineração de Dados (do inglês, *Data Mining*) é o processo de pesquisa em grandes quantidades de dados para extração de conhecimento, utilizando técnicas de Inteligência Computacional (ENGELBRECHT, 2007) para procurar relações de similaridade ou discordância entre dados. Este processo tem como objetivo encontrar padrões, irregularidades e regras, com o intuito de transformar dados, aparentemente ocultos, em informações relevantes para a tomada de decisão e/ou avaliação de resultados (FAYYAD, PIATETSKY-SHAPIRO and SMYTH, 1996).

Esta etapa do KDD consiste na aplicação de técnicas e algoritmos específicos, que irão extrair os padrões a partir dos dados. Nesse sentido, de acordo com o objetivo da extração de

conhecimento, deve-se escolher a(s) técnica(s) e algoritmo(s) apropriado(s) para o processo de KDD.

De acordo com FAYYAD et al. (1996), existem diversas técnicas de Data Mining para encontrar respostas ou extrair conhecimento em repositórios de dados, sendo alguns dos mais importantes para o KDD: Classificação, Regressão, Clusterização, Sumarização e Regras de Associação. Estes métodos são brevemente descritos a seguir.

- Regras de Associação – Tem por objetivo encontrar relacionamentos ou padrões frequentes que ocorrem em um conjunto de dados, ou seja, é a busca por itens que ocorram de forma simultânea frequentemente em transações do banco de dados. Algoritmos tais como o Apriori (AGRAWAL and SRIKANT, 1994), GSP (SRIKANT and AGRAWAL, 1996) , entre outros, são exemplos de algoritmos para tarefa de descoberta de associações.
- Clusterização – tem como objetivo associar um item a uma ou várias classes categóricas (clusters), determinando as classes pelos dados, independentemente da classificação pré-definida. Os clusters são definidos por meio do agrupamento de dados baseados em medidas de similaridade ou modelos probabilísticos, visando detectar a existência de diferentes grupos dentro de um determinado conjunto de dados e, em caso de sua existência, determinar quais são eles. Pode ser estabelecido previamente um número de grupos a ser formado, ou então pode-se admitir ao algoritmo de agrupamento uma livre associação de unidades, de forma que a quantidade de grupos resultante seja conhecida somente ao final do processo. Para implantar esse tipo de tarefa são utilizados algoritmos tais como K-Means, DBSCAN, SOM, entre tantos outros (TAN, STEINBACH and KUMAR, 2005).
- Sumarização – É uma tarefa descritiva, que consiste em definir um conjunto mínimo de características que seja capaz de identificar um subconjunto de objetos. As técnicas de sumarização são comumente aplicadas para análise exploratória de dados e geração de relatórios automatizados (FAYYAD, PIATETSKY-SHAPIRO and SMYTH, 1996).
- Classificação – É uma tarefa preditiva, que consiste em determinar uma função que mapeie cada item de uma base dados a uma das classes previamente definidas. Por exemplo, classificar transações de cartões de crédito como legítimas ou

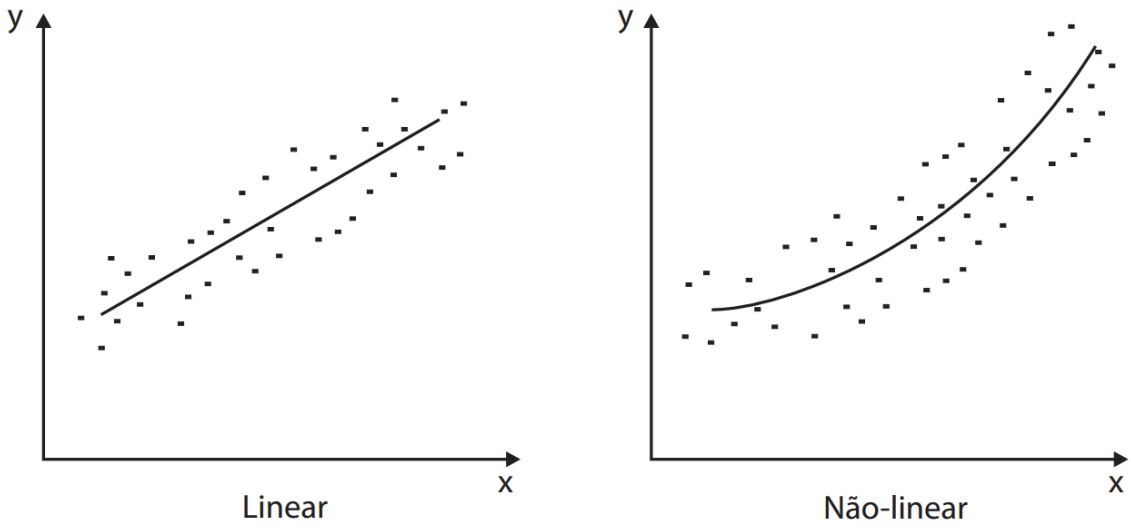
fraudulentas. Algumas das técnicas mais utilizadas para classificação são: Árvores de Decisão, Classificação Bayesiana, Redes Neurais, Algoritmos Genéticos, entre outras (FAYYAD, PIATETSKY-SHAPIRO and SMYTH, 1996).

- Regressão – O principal objetivo desta técnica é obter uma equação que explique satisfatoriamente a relação entre uma variável dependente e uma ou mais variáveis independentes, possibilitando fazer predição de valores da variável alvo de interesse. Este relacionamento pode ser por uma função linear ou uma função não-linear e ainda pode ser dividida em regressão simples ou regressão múltipla. Esta técnica, por ser a escolhida para o desenvolvimento deste trabalho, será descrita com mais detalhes na próxima subseção (GRAYBILL and IYE, 1994).

2.2.3 Análise de Regressão

Os modelos de regressão são largamente utilizados em diversas áreas do conhecimento, tais como: computação, administração, engenharias, biologia, agronomia, saúde, sociologia etc. O principal objetivo desta técnica é obter uma equação que explique satisfatoriamente a relação entre uma variável dependente (alvo, resposta) e uma ou mais variáveis independentes (preditoras, explicativas), possibilitando fazer predição de valores da variável de interesse (GUIMARÃES, 2008). Adicionalmente, existem dois tipos de análise de regressão: regressão múltipla (mais de uma variável independente) e regressão simples (apenas uma variável independente) e ambas podem se dar tanto por um relacionamento linear, como não-linear, conforme a Figura 2.9.

Figura 2.9 – Formas lineares e não lineares entre pares de variáveis



Fonte: (GUIMARÃES, 2008)

2.2.3.1 Regressão Linear

O primeiro passo, na análise de regressão, é obter as estimativas para os parâmetros do modelo, logo é necessário um método de estimação. Um dos métodos estatísticos mais utilizado e recomendado pela sua precisão é o método dos mínimos quadrados, também conhecido como Quadrados Mínimos Ordinários (QMO) ou OLS (do inglês, *Ordinary Least Squares*), o qual busca encontrar o melhor ajuste (equação) para um conjunto de dados tentando minimizar a soma dos quadrados das diferenças entre o valor estimado e os dados observados (HOFFMANN and VIEIRA, 1977).

Entretanto, o método dos Quadrados Mínimos é apenas um, dentre os inúmeros métodos de regressão que existem na literatura. Alguns outros exemplos de métodos de regressão são: *Generalized least squares* (GLS), *Iteratively reweighted least squares* (IRLS), *Least Absolute Deviation* (LAD), *Partial Least Squares* (PLS), *Bayesian Linear Regression*, *Ridge regression*, *Lasso Regression*, entre tantos outros (HASTIE, TIBSHIRANI and FRIEDMAN, 2009) (FILZMOSER, 2008).

Logo, diferentes métodos de regressão geram modelos em diferentes formatos de representação. Como esses modelos expressam o conhecimento obtido durante o processo de mineração de dados, cada método irá expressar o conhecimento de uma forma diferenciada.

Com isso, depois de gerados, esses modelos devem ser avaliados segundo alguns fatores, como por exemplo, a acurácia. A acurácia tem como objetivo avaliar o desempenho do modelo de regressão na predição do valor alvo de novas observações. Por isso, a maioria das medidas de acurácia utilizadas em problemas de regressão são baseadas na diferença entre o valor predito pelo modelo e o valor real do atributo.

A Raiz do Erro Quadrático Médio (RMSE do inglês, *Root-Mean-Square Error*), indica o ajuste absoluto do modelo aos dados, ou, em outras palavras, o quão perto os valores observados (reais) estão dos valores previstos. A Equação 2.4 apresenta a fórmula do RMSE, onde $X_{obs[i]}$ representa o valor observado e $X_{pred[i]}$ o valor previsto na iteração/observação i. Se o principal objetivo do modelo é a previsão, o RMSE é o critério mais importante para a medir a acurácia entre diferentes modelos preditivos para uma mesma variável dependente, pois ele apresenta os valores do erro nas mesmas dimensões da variável analisada (HARRELL, 2002).

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(X_{obs[i]} - X_{pred[i]})^2}{n}} \qquad \text{Equação 2.4}$$

Adicionalmente, é possível utilizar mais de uma variável independente como atributo preditor, neste caso, temos uma regressão múltipla.

2.2.3.2 Regressão Não-linear

Enquanto funções de regressão linear simples e múltipla, são adequadas para a modelagem de uma grande variedade de relações entre as variáveis respostas (dependentes) e variáveis preditoras (independentes), muitas situações exigem uma função não-linear.

Por definição, um modelo de regressão é não-linear se pelo menos um dos seus parâmetros aparecem de forma não-linear. Por exemplo, os seguintes modelos na Equação 2.5, Equação 2.6, Equação 2.7 e Equação 2.8 são todos não-lineares.

$$Y = \exp(\alpha + \beta X) \qquad \text{Equação 2.5}$$

$$Y = \alpha + \beta \exp(-\gamma X) \qquad \text{Equação 2.6}$$

$$Y = (\alpha + \beta x)^{-1} \qquad \text{Equação 2.7}$$

$$Y = (\alpha + \beta)^{-1}[\exp(-\alpha) + \exp(-\beta)] \qquad \text{Equação 2.8}$$

No modelo da Equação 2.5, os parâmetros α e β são não-lineares. Já no modelo da Equação 2.6 os paramentos α e β são lineares enquanto que γ é não-linear. Adicionalmente, nos modelos da Equação 2.7 e Equação 2.8, ambos os parâmetros são não-lineares.

De maneira semelhante à regressão linear, também existem diversos métodos na literatura que buscam encontrar relações não-lineares entre variáveis. Muitos dos métodos que se aplicam à regressão linear, também se aplicam/adaptam à regressão não-linear, como por exemplo, o método dos Quadrados Mínimos. Outros métodos comumente utilizados são: Árvores de Regressão, Redes Neurais, *Support Vector Machine*, *Random Forest*, entre tantas outras (HASTIE, TIBSHIRANI and FRIEDMAN, 2009).

2.3 TRABALHOS RELACIONADOS

2.3.1 Métricas de Software para Avaliar a Qualidade de Software e Detecção de Falhas

Dois dos maiores objetivos de gestão de qualquer empresa de software são: diminuir o tempo e custo dos testes e aumentar a qualidade do código. Segundo Nagappan (2006), tais objetivos podem ser satisfeitos através da medição do código e análise e construção de modelos de previsão de defeitos (NAGAPPAN, BALL and MURPHY, 2006). Detecção de falhas fundamentada em mineração de métricas de software tem sido uma área de pesquisa ativa por muitos anos (BRIAND, 2005). A previsão de defeitos em software é a tarefa de classificar os módulos do software como propensos a falhas (do inglês, *fault-prone* - FP) e não propensos (do inglês, *non-fault-prone* - NFP), por meio de uma classificação baseada em métricas (BRIAND, MELO and WÜST, 2002) (MENZIES, DISTEFANO, *et al.*, 2004).

Foi mostrado que a detecção de módulos propensos a falhas utilizando técnicas de mineração de dados tem 70% de acurácia (MENZIES, GREENWALD and FRANK, 2007) enquanto detecção de falhas feita por revisão/análise manual tem 60% de acurácia (SHULL, BASILI, *et al.*, 2002). Considerando que um revisor experiente consegue inspecionar de 8 a 20 LOC/minuto, torna-se claro que preditores automáticos de defeitos baseados em métricas de modelo e código tentem ajudar muito na dectação de falhas.

Após os estudos seminais de Porter e Selby (1990), um grande interesse na previsão de falhas de software a partir de métricas surgiu (PORTER and SELBY, 1990). Alguns exemplos de trabalhos relacionados são (MENZIES, GREENWALD and FRANK, 2007) (JIANG, CUKI, *et al.*, 2008) (TURHAN, MENZIES, *et al.*, 2009) (KANMANI, UTHARIARAJ, *et al.*, 2004; MIHANCEA and MARINESCU, 2013) (NAKAMURA, 2012). Turhan (2009), mostra que é possível construir classificadores com base em métricas de software de um fabricante Turco de geladeiras e prever falhas em módulos do software de um ônibus espacial da NASA (TURHAN, MENZIES, *et al.*, 2009), neste caso demostrando que tal relação não depende do tipo de aplicação.

Um dos primeiros trabalhos a utilizar métricas de modelo foi realizado por Ohlsson e Alberg (1996), a motivação do estudo foi ter o conhecimento de qual módulo está mais propenso a falhas antes de codificar o software. Para isso foram analisados 130 módulos de um sistema telefônico da Ericsson. As métricas de modelo foram derivadas de modelos FDL (*Formal Description Language*) e como técnincas de predição foram utilizados os modelos de Alberg e regressão linear. Os resultados sugerem que é possível prever a maior parte dos módulos propensos a falhas a partir dos modelos antes de começar a codificação (OHLSSON and ALBERG, 1996).

Jiang et al. (JIANG, CUKI, *et al.*, 2008) mostram que a previsão de módulos propensos a falhas baseada somente em métricas de modelo é menos precisa do que a previsão baseada em métricas de código. O trabalho mostra ainda que os classificadores treinados com base em atributos de nível de código apenas não são tão precisos quanto um classificador baseado em métricas de código e modelo combinadas. Embora as métricas no nível de modelo sejam mais difíceis de se obter, o uso de ambas as métricas combinadas leva a uma melhor precisão e desempenho.

Istehad e Mohammad conduziram um estudo experimental ao longo de várias versões do Mozilla Firefox e puderam constatar em seus experimentos que as métricas de complexidade, acoplamento e coesão tinham um certo nível de correlação com as vulnerabilidades apresentadas pelo software (CHOWDHURY and ZULKERNINE, 2010). Já Yadav e Singh (2013) avaliaram três softwares com o objetivo de estabelecer uma correspondência entre a qualidade do projeto e a qualidade do produto final (YADAV and SINGH, 2013).

2.3.2 Exploração de Espaço de Projetos em Sistemas Embarcados

Alguns trabalhos têm como objetivo permitir a exploração do espaço de projeto através de estimativas extraídas de modelos UML. Oliveira (2009), explora os aspectos da Engenharia Dirigida por Modelos (MDE – do inglês Model Driven Engineering) visando extrair informações dos modelos para explorar o espaço de projeto de forma antecipada. É especificado um conjunto de regras para geração de modelos UML que permite a extração de informações relevantes à avaliação quantitativa dos modelos permitindo, assim, podar o número de possíveis soluções de projeto. Além disso, os requisitos não-funcionais são especificados em UML como estereótipos visando também remover soluções inválidas do espaço de exploração (OLIVEIRA, FERREIRA, *et al.*, 2009) (OLIVEIRA, NASCIMENTO, *et al.*, 2010).

Nascimento (2012) propõe um *framework* onde baseia-se em também em MDE e modelos UML para extração de informação prévia, com o objetivo de melhorar o processo de exploração do espaço de projeto. Neste trabalho, são usados os conceitos de MDE para a geração automática de uma representação interna dos fluxos de controle e dados a partir da especificação de aplicações embarcadas descritas em diagramas UML de classe e sequência. Depois, por meio de regras de transformações aplicadas aos modelos UML uma representação baseada em MOF (*Meta Object Facility*) é obtida e com essa representação eles conseguem capturar os aspectos estruturais e comportamentais de um modelo (NASCIMENTO, OLIVEIRA and WAGNER, 2012).

De um modo mais tradicional, Mariani (2010) propõe um método iterativo para exploração de espaço de projeto, com relação às configurações possíveis de hardware, que explora as propriedades de correlação dessas configurações de sistemas multi-processados para então escolher as configurações mais promissoras a serem analisados em uma simulação de baixo nível (MARIANI, BRANKOVIC, *et al.*, 2010).

2.3.3 Estimação Prévia de Requisitos não Funcionais para Sistemas Embarcados

Poucos trabalhos têm tentado estabelecer uma relação entre a qualidade do software e seu desempenho em plataformas embarcadas. Chatzigeorgiou e Stephanides (CHATZIGEORGIOU and STEPHANIDES, 2002) mostram que a programação OO pode levar a um maior tempo de execução e maior consumo de energia em processadores

embarcados. Mattos (2005) propõe a transformação automática de código OO para torná-lo mais eficiente em uma plataforma embarcada de tal forma que o desenvolvedor possa trabalhar em um nível mais alto de abstração com menor penalidade no desempenho final. No entanto, a ferramenta proposta em (MATTOS, SPECHT, *et al.*, 2005) apenas transforma objetos dinâmicos em estáticos, sem uma análise mais aprofundada da organização dos módulos. Redin (2008) avalia a aplicabilidade de métricas de software tradicionais em software embarcado. Em seus experimentos foi verificado que decisões tomadas nas fases de modelagem do software têm um grande impacto nas métricas físicas. Entretanto, não foi definida nenhuma relação específica entre o código e as métricas físicas (REDIN, OLIVEIRA, *et al.*, 2008).

Corrêa et al. (CORRÊA, LAMB, *et al.*, 2010) propõem a utilização de uma rede neural para encontrar correlações entre as métricas de código e as métricas físicas. A rede neural é treinada, primeiramente, com algumas métricas de código (e.g., CC, NC, LCOM, NOA, NOC, etc) e métricas físicas (uso de memória, ciclos e energia) de algumas versões de uma única aplicação. Assim, para uma nova versão da mesma aplicação, a rede é alimentada com as métricas de código e fornece, como resultado, uma estimava das métricas físicas (ciclos e energia). Entretanto, as diferentes versões de código utilizadas durante o treinamento eram resultantes de uma operação de refatoração simples (*method inline* (FOWLER, 1999)) aplicada sequencialmente sobre o código. Assim, a organização geral dos módulos da aplicação não se altera de uma versão para outra e o erro de predição para outros tipos de modificações não foi avaliado, o que torna o método, de certa forma, limitado em relação à sua capacidade de predição. Assim, uma relação mais precisa entre as decisões de projeto de alto nível e métricas físicas ainda não está bem estabelecida.

WEHRMEISTER (2012) utiliza uma abordagem de verificação automática de especificações de alto nível (modelos UML) para simular o comportamento do sistema ainda nas fases iniciais do projeto. Mais especificamente, seu framework simula o comportamento dentro de modelos UML, gerando um rastro das ações executadas. Os resultados obtidos mostram que a simulação precoce de modelos UML é possível, abrindo espaço para a sua utilização em diferentes ferramentas CASE para a verificação, nas fases iniciais do projeto, dos requisitos não funcionais em sistemas embarcados (WEHRMEISTER, PACKER and CERON, 2012).

3 ABORDAGEM PROPOSTA

Do ponto de vista do desenvolvedor de software embarcado, um método de estimação ideal deve prover uma informação precisa sobre o desempenho de um software, usando apenas informações de projeto, ou seja, antes da codificação e sem a necessidade de se executar o sistema em uma determinada plataforma de hardware. Além disso, tal método deve prover informação suficiente para que o desenvolvedor possa alterar o projeto de forma satisfatória. Dentro deste contexto, o objetivo inicial desta dissertação era encontrar correlações entre métricas de software estáticas (por exemplo, métricas extraídas de diagramas ou código fonte) e métricas dinâmicas (ou seja, métricas do software em execução em uma plataforma). Se tal correlação existir, ela permitirá não apenas predizer o desempenho de um projeto em uma dada plataforma, mas também indicar de que forma o projeto deve ser alterado para atingir o desempenho desejado. Na Seção 3.1 são apresentados alguns resultados da tentativa de se encontrar a correlação (de Pearson) entre métricas de software estáticas e dinâmicas. Embora os resultados da correlação não tenham sido satisfatórios, eles serviram de motivação para o uso de uma técnica/metodologia alternativa que se mostrou, no fim, bastante promissora, apresentada na Seção 3.2.

3.1 CORRELAÇÃO DE PEARSON (MOTIVAÇÃO)

Na análise de correlação, procura-se determinar o grau de relacionamento entre duas variáveis, ou seja, procura-se medir a covariabilidade entre elas, implicando que quando os valores em uma variável mudam, valores na outra variável também mudam, de maneira previsível. Em outras palavras, as duas variáveis não são independentes, logo, diferente da análise de regressão, não é necessário distinguir a variável dependente e a variável independente.

Desse modo, se for possível estabelecer uma relação direta (nas duas direções) entre atributos do modelo de alta abstração, com atributos do software em execução no hardware, será possível usar uma métrica de modelo para predizer uma métrica de execução. Por exemplo, se houver uma correlação positiva entre as métricas de acoplamento e energia, pode-se dizer que sempre que o acoplamento aumenta o consumo de energia também aumenta, ou vice-versa.

Um método usualmente conhecido para medir a correlação entre duas variáveis é o Coeficiente de Correção Linear de Pearson (ρ) (DANCEY and REIDY, 2011). O coeficiente Pearson varia entre -1 e +1, onde o valor sugere a força do relacionamento entre as duas variáveis e o sinal indica se a correlação é direta ou inversa. Um valor de $\rho = 1$ indica uma correlação perfeita e direta, enquanto $\rho = 0$ indica nenhuma correlação entre as variáveis, ou ainda $\rho = -1$ indica que existe uma correlação perfeita de ordem inversa.

Entretanto, os valores de ρ na prática raramente chegam a valores perfeitos (DANCEY and REIDY, 2011), o que não implica na inexistência de correlação. Muitos autores definem intervalos de confiança para os valores de ρ e classificam basicamente a magnitude da correlação entre fraca, moderada, forte. Por exemplo, Dancey (2011), apresenta os intervalos de confiança conforme a Tabela 3.1. A Tabela 3.1 apresenta os valores em módulo, ou seja, dependendo do sinal de ρ o grau de relacionamento ainda pode ser classificado com direto ou inverso.

Tabela 3.1 – Grau de Força dos Valores do Coeficiente de Correção de Pearson

Valor de ρ	**Grau de Magnitude do Relacionamento**
$\rho < \|0.1\|$	Nula
$\|0.1\| \geq \rho < \|0.5\|$	Fraca
$\|0.5\| \geq \rho < \|0.7\|$	Moderada
$\|0.7\| \geq \rho < \|1\|$	Forte
$\rho = \|1\|$	Perfeita

Fonte: (DANCEY and REIDY, 2011)

A primeira abordagem feita nesse trabalho foi então tentar encontrar correlações entre as métricas utilizando a correlação de Pearson. Mais especificamente, tentar encontrar uma correlação que seja independente do tipo da aplicação.

Para isso, foram extraídas métricas de 3 conjuntos de aplicações, com objetivo de tentar encontrar alguma correlação equivalente entre diferentes tipos de aplicações. O primeiro conjunto de aplicações consiste em 21 implementações distintas de um sistema bancário. O segundo conjunto de aplicações é formado por 12 implementações distintas de um sistema de votação. Por fim, o terceiro conjunto de aplicações é constituído por 11 implementações também distintas de um sistema de comércio eletrônico. Maiores informações sobre estas aplicações são apresentadas na Seção 4.1.

Após a coleta das métricas dessas aplicações (processo apresentado nas Seções 3.2.1 e 4.1), foram calculados os coeficientes de correlação entre métricas de diagramas de classes (Tabela A 1) e as métricas de hardware (Tabela 2.1) para cada conjunto de aplicação. Os coeficientes de correlação para cada um desses cenários são apresentados na Tabela 3.2, Tabela 3.3 e Tabela 3.4, respectivamente referentes ao sistema bancário, sistema de comércio eletrônico e sistema de votação. Em todas as tabelas, as setas verdes destacam um valor de $\rho \geq 0.5$ e as setas vermelhas destacam um valor de $\rho \leq -0.5$.

Analisando apenas a Tabela 3.2 é possível observar uma correlação moderada entre algumas métricas de hardware e algumas de modelo. O interessante é que todas as métricas de modelo que apresentam correlação moderada com métricas de hardware são referentes aos aspectos relacionados à herança, ou seja, para a aplicação do sistema bancário, o grau de herança definido no modelo do software possui uma correlação moderada com a performance da aplicação durante a execução.

Tabela 3.2 – Correlação de Pearson para Sistema Bancário

Hardware / Modelo	L2 Misses	Predicted Branches	Missed Branches	IPC	Icache Misses	Dcache misses	Energy
NumAttr	0.216	0.326	0.337	0.011	0.388	0.287	0.286
NumOps	0.336	0.257	0.287	0.066	-0.013	0.262	0.281
NumPubOps	0.368	0.274	0.289	0.107	0.006	0.303	0.223
Setters	0.082	0.201	0.072	0.310	-0.147	0.154	0.227
Getters	-0.122	-0.071	-0.056	-0.053	-0.209	-0.141	0.128
NumDesc	↑0.565	↑0.607	↑0.648	0.134	↑0.559	↑0.609	0.279
NOC	↑0.565	↑0.607	↑0.648	0.134	↑0.559	↑0.609	0.279
DIT	↑0.537	↑0.551	↑0.618	0.130	↑0.514	↑0.554	0.182
CLD	↑0.543	↑0.539	↑0.606	0.059	↑0.501	↑0.550	0.309
OpsInh	0.408	0.254	0.300	0.101	-0.007	0.305	0.252
AttrInh	0.453	0.390	↑0.523	-0.101	0.462	0.411	↑0.583
Dep_Out	0.294	0.247	0.292	0.021	-0.075	0.221	0.278
Dep_In	0.248	0.130	0.225	-0.085	-0.118	0.124	0.329
NumAssEl_ssc	0.309	0.255	0.329	-0.038	0.146	0.245	0.303
NumAssEl_sb	0.258	0.204	0.277	-0.055	0.096	0.192	0.298
NumAssEl_nsb	-0.025	-0.028	-0.035	-0.061	-0.133	-0.039	0.185
EC_Par	-0.029	-0.294	-0.123	-0.300	-0.349	-0.249	0.076
IC_Par	0.138	0.070	0.026	0.248	-0.354	0.071	-0.072

Fonte: Autor

Tabela 3.3 – Correlação de Pearson para Sistema de Comércio Eletrônico

Hardware / Modelo	L2 Misses	Predicted Branches	Missed Branches	IPC	Icache Misses	Dcache misses	Energy
NumAttr	-0.320	-0.214	-0.258	0.113	-0.059	-0.216	-0.215
NumOps	-0.302	-0.238	-0.244	-0.077	-0.115	-0.268	-0.225
NumPubOps	0.005	-0.010	-0.015	-0.063	0.033	-0.032	-0.007
Setters	-0.003	0.017	-0.003	0.278	-0.020	-0.017	0.008
Getters	-0.264	-0.228	-0.216	-0.089	-0.136	-0.274	-0.215
NumDesc	0.530	0.571	0.581	0.560	0.519	0.531	0.553
NOC	0.620	0.637	0.649	0.423	0.543	0.601	0.621
DIT	0.254	0.502	0.453	0.225	0.633	0.429	0.499
CLD	0.137	0.012	0.044	0.237	-0.093	0.026	0.006
OpsInh	-0.162	-0.323	-0.253	0.432	-0.395	-0.322	-0.327
AttrInh	-0.494	-0.481	-0.470	0.613	-0.361	-0.510	-0.490
Dep_Out	-0.657	-0.572	-0.594	0.023	-0.366	-0.592	-0.563
Dep_In	-0.563	-0.576	-0.587	0.323	-0.441	-0.587	-0.584
NumAssEl_ssc	-0.406	-0.496	-0.462	-0.244	-0.517	-0.464	-0.480
NumAssEl_sb	-0.512	-0.424	-0.418	0.491	-0.238	-0.455	-0.426
NumAssEl_nsb	-0.210	-0.262	-0.202	0.359	-0.275	-0.278	-0.258
EC_Par	0.427	0.451	0.512	0.216	0.427	0.385	0.449
IC_Par	0.081	0.093	0.149	0.551	0.128	0.005	0.082

Fonte: Autor

Avaliando a Tabela 3.3 é possível pensar que a correlação entre o grau de herança de um projeto UML e o desempenho da aplicação final no hardware realmente tem alguma correlação, pois mesmo mudando o tipo de aplicação em análise, a correlação mesmo que moderada se manteve entre algumas métricas de herança e hardware. Mas ainda neste cenário apareceram algumas correlações moderadas também com as métricas de acoplamento, que antes na Tabela 3.2 não tinham aparecido. Logo, essas correlações entre as métricas de acoplamento e hardware podem ser vistas como uma característica deste tipo de aplicação, ou seja, uma correlação que não deve se manter para diferentes tipos de aplicações.

Entretanto, analisando os resultados do último conjunto de métricas na Tabela 3.4, é notório que grande parte das correlações antes vistas na Tabela 3.2 e na Tabela 3.3 não permaneceram neste estudo de caso que possui um conjunto de aplicações diferentes, o que de certa forma, rejeita nossa premissa de uma correlação moderada entre o grau de herança e performance apresentada até o momento que pudesse ser independente do tipo de aplicação. A métrica DIT até continua com algumas correlações moderadas, com um pequeno destaque para correlação com número de misses na cache de instruções, que teve um coeficiente de correlação maior que 0.5 em todos casos, conforme pode ser visto destacado em verde nas três tabelas.

.

Tabela 3.4 – Correlação de Pearson para Sistema de Votação

Hardware / Modelo	L2 Misses	Predicted Branches	Missed Branches	IPC	Icache Misses	Dcache misses	Energy
NumAttr	0.056	-0.075	-0.104	0.235	-0.241	0.050	-0.166
NumOps	-0.201	-0.412	-0.248	-0.108	-0.442	-0.365	-0.456
NumPubOps	-0.303	↓-0.520	-0.359	-0.022	↓-0.533	-0.435	↓-0.538
Setters	-0.102	-0.249	-0.108	-0.049	-0.261	-0.226	-0.288
Getters	-0.199	-0.412	-0.245	-0.100	-0.459	-0.326	-0.488
NumDesc	0.041	0.241	0.205	0.299	0.208	0.267	0.244
NOC	0.083	0.256	0.190	0.316	0.213	0.307	0.298
DIT	0.215	0.498	0.385	0.135	↑0.522	0.408	↑0.541
CLD	0.085	0.357	0.338	0.254	0.406	0.306	0.373
OpsInh	0.075	0.249	0.208	0.281	0.217	0.278	0.296
AttrInh	0.318	0.327	0.202	0.187	0.208	0.440	0.363
Dep_Out	-0.124	-0.399	-0.269	-0.173	-0.412	-0.306	-0.460
Dep_In	-0.139	-0.395	-0.260	-0.104	-0.426	-0.290	-0.464
NumAssEl_ssc	-0.124	-0.381	-0.317	-0.054	-0.370	-0.237	-0.363
NumAssEl_sb	-0.102	-0.362	-0.277	-0.185	-0.352	-0.277	-0.382
NumAssEl_nsb	-0.371	↓-0.584	-0.446	-0.046	↓-0.630	-0.494	↓-0.608
EC_Par	-0.065	-0.220	-0.060	-0.088	-0.255	-0.212	-0.363
IC_Par	-0.142	-0.441	-0.309	-0.049	↓-0.531	-0.312	-0.455

Fonte: Autor

Na tentativa de encontrar uma correlação entre métricas que capturasse as características das diversas aplicações, foi feito mais um experimento onde todas as métricas de todas as aplicações foram avaliadas de forma conjunta. O resultado desta análise pode ser visto na Tabela 3.5. Como resultado tivemos apenas uma correlação moderada entre DIT e L2 Misses.

Tabela 3.5 – Correlação de Pearson para Todas Aplicações

Hardware / Modelo	L2 Misses	Predicted Branches	Missed Branches	IPC	Icache Misses	Dcache misses	Energy
NumAttr	-0.238	-0.265	-0.245	0.349	-0.258	-0.265	-0.288
NumOps	-0.104	-0.156	-0.131	0.237	-0.203	-0.156	-0.175
NumPubOps	-0.086	-0.216	-0.190	0.320	-0.271	-0.198	-0.246
Setters	-0.082	-0.042	-0.057	0.314	-0.104	-0.055	-0.054
Getters	-0.163	-0.132	-0.123	0.021	-0.160	-0.154	-0.102
NumDesc	0.249	-0.046	0.003	0.059	-0.098	0.012	-0.096
NOC	0.179	-0.170	-0.118	0.137	-0.227	-0.107	-0.221
DIT	0.551	0.414	0.442	-0.252	0.387	0.441	0.384
CLD	0.203	0.259	0.283	-0.055	0.284	0.261	0.240
OpsInh	0.357	0.242	0.263	-0.101	0.198	0.276	0.243
AttrInh	0.140	-0.099	-0.037	0.064	-0.124	-0.054	-0.073
Dep_Out	0.060	0.026	0.049	0.087	-0.042	0.022	0.011
Dep_In	-0.046	-0.019	0.005	0.092	-0.050	-0.032	-0.003
NumAssEl_ssc	-0.065	-0.028	-0.012	0.147	-0.035	-0.037	-0.030
NumAssEl_sb	-0.025	0.011	0.028	0.084	0.000	-0.001	0.015
NumAssEl_nsb	-0.183	-0.208	-0.208	0.056	-0.248	-0.205	-0.180
EC_Par	-0.257	-0.228	-0.189	0.092	-0.200	-0.252	-0.176
IC_Par	-0.157	-0.176	-0.178	0.361	-0.246	-0.180	-0.211

Fonte: Autor

Os resultados obtidos nestes experimentos com a análise de correlação de Pearson mostram que existe uma correlação moderada para fraca entre o grau de herança de um projeto OO e seu desempenho no hardware. Tal correlação existe principalmente entre as métricas DIT e o número de *instruction cache misses*, algo que ocorreu em todos casos quando as aplicações foram analisadas de forma individual. Esta correlação também apareceu, na Tabela 3.5, onde todas as aplicações foram avaliadas conjuntamente. Por outro lado, todos os outros indicadores variaram com a aplicação e a correlação encontrada é apenas moderada. A partir desses resultados conclui-se que a hipótese original de existência de uma correlação linear entre métricas de alto nível e métricas de execução indiferente de aplicação não se mostrou verdadeira nos cenários em que foram estimulados neste trabalho. Isto pode se dar por diferentes razões, tais como: 1) a relação entre as métricas não é linear; 2) a relação entre as métricas pode depender de vários fatores simultaneamente; 3) a relação pode envolver uma combinação de métricas.

Outra constatação importante desta primeira análise é que o uso de métricas como base da análise exige cuidado. Embora a extração das métricas seja um processo simples e

automatizado, sua análise é complicada pela quantidade de dados e pela variação não apenas da natureza desses dados mas também dos próprios objetos sob análise. De fato, cada métrica de modelo ou execução avalia um aspecto distinto da aplicação. Devido ao grande número de implementações disponíveis, qualquer análise comparativa manual tornou-se infrutífera e inviável. Dessa forma, a segunda abordagem proposta nessa dissertação baseia-se em uma técnica de descoberta do conhecimento e mineração de dados. Tal técnica realiza uma análise estruturada sobre um conjunto grande de dados permitindo justamente identificar relações não triviais ou não explícitas entre os dados. Optamos por adaptar, para o contexto deste trabalho, um processo completo de extração de conhecimento proposto por Fayyad (1996), conhecido como *Knowledge-Discovery in Databases* (KDD). Esta abordagem é detalhada a seguir.

3.2 KNOWLEDGE-DISCOVERY IN DATABASES (KDD)

Conforme o breve resumo dos nossos resultados iniciais apresentados na seção anterior, que indicaram a ausência de correlações consistentes entre as métricas de diagramas de classes com as métricas de hardware, tornou-se necessário para o andamento deste trabalho a investigação de uma técnica que pudesse vir a ser capaz de encontrar tais relações.

O KDD, conforme apresentado na Seção 2.2, é um processo iterativo composto de diversas etapas e que tem como objetivo extrair conhecimento de uma grande massa de dados. Nas próximas subseções iremos apresentar as etapas da nossa abordagem, as quais serão baseadas no processo de KDD. No núcleo desta abordagem, como técnica de mineração de dados, estará a análise de regressão, e portanto, as etapas que sucedem a mineração de dados terão como objetivo preparar os dados para esta técnica.

3.2.1 Extração das Métricas

A necessidade da utilização de ferramentas para automatizar tarefas nas mais diferentes áreas do conhecimento é algo comumente discutido e isto não seria diferente para a extração das métricas de software. Para o desenvolvimento deste trabalho, foram avaliadas inúmeras ferramentas de extração de métricas, das quais apenas quatro foram as escolhidas: SDMetrics (WÜST, 2014) para extração das métricas de modelos UML; Understand (TOOLWORKS, 2014) para extração das métricas de código fonte; gem5 (BINKERT, BECKMANN, *et al.*,

2011) para simulação e extração das métricas de métricas de hardware; e o McPAT (LI, AHN, *et al.*, 2009) para extração de energia consumida.

O SDMetrics é uma ferramenta que tem como objetivo medir os atributos de qualidade interna de projetos orientados a objetos, analisando a estrutura dos modelos UML. O SDMetrics extrai métricas de vários tipos de diagramas UML, entre eles, de classe, de sequência, de casos de uso e de atividade. Entretanto, por só termos a nossa disposição como estudo de caso da etapa de modelagem diagramas de classes, consequentemente só puderam ser extraídas as métricas desses diagramas, as quais podem ser vistas na Tabela A 1 em anexo. A lista completa de todas as métricas que a ferramenta oferece pode ser vista em seu site[1] oficial.

O Understand é uma ferramenta de análise de código fonte e extração de métricas. Ele foi projetado para ajudar a manter e compreender grandes quantidades de código fonte recém-criados ou legados. O Understand fornece um Ambiente de Desenvolvimento Integrado (IDE – do inglês Integrated Development Environment) multi-plataforma para várias linguagens de programação. É possível analisar códigos C/C++, C#, Java, VHDL, Python, PHP, entre outros. O *Understand* é muito eficiente na coleta de métricas de código. As métricas podem ser extraídas automaticamente através de linha de comando, exportadas para planilhas ou vistas graficamente. As métricas também podem ser extraídas em diferentes níveis (projeto, arquivos, classes, funções) ou arquiteturas definidas pelo usuário.

O Understand contém um total de 99 métricas das quais 53 podem ser aplicadas a projetos Java. A lista completa de todas as métricas que a ferramenta oferece pode ser vista em seu site[2] oficial. Por outro lado, as que o uso foi possível nessa dissertação podem ser vistas na Tabela B 1, Tabela B 2 e Tabela B 3 em anexo.

O gem5 (BINKERT, BECKMANN, *et al.*, 2011) é um simulador amplamente configurável, com diversos modelos de CPU vários conjuntos de instruções (*Instruction Set Architecture* - ISA), entre elas MIPS, ARM e x86. Além disso, o seu sistema de memória também é amplamente configurável, podendo-se escolher quantos níveis a hierarquia de memória terá, qual será a política de substituição da cache, incluindo suporte para múltiplos protocolos de coerência de cache, entre outras características. Como resultado de uma simulação, o gem5 gera um *log* de estatísticas (arquivo "stats.txt"), onde cada linha pode ser vista como uma métrica

[1] http://www.sdmetrics.com/LoM.html
[2] https://scitools.com/support/metrics_list

dinâmica. Um exemplo de parte desse arquivo pode ser visto na Figura 3.1. Dentre as métricas disponíveis, as selecionadas são apresentadas na Tabela 2.1.

Figura 3.1 – Exemplo de Log do gem5

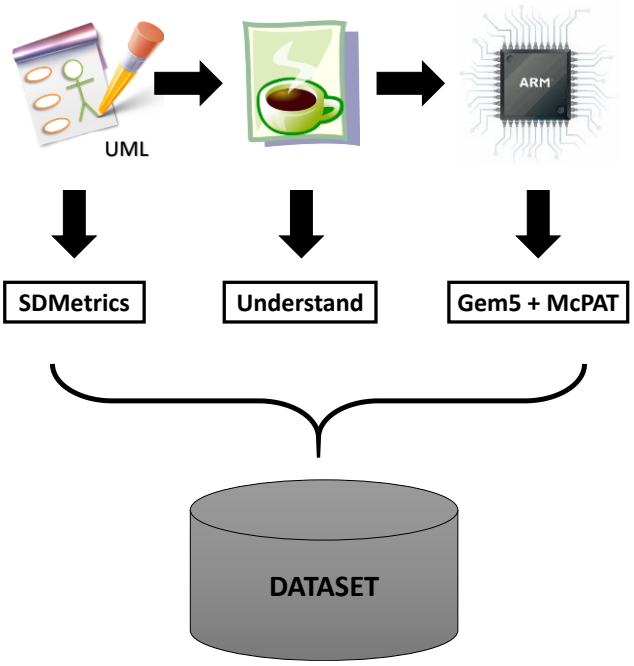
```
1
2    ---------- Begin Simulation Statistics ----------
3    sim_seconds                                  5.293499      # Number of seconds simulated
4    sim_ticks                                 5293499319500     # Number of ticks simulated
:
:
585  system.cpu.branchPred.lookups                710592807      # Number of BP lookups
586  system.cpu.branchPred.condPredicted          498684212      # Number of conditional branches predicted
587  system.cpu.branchPred.condIncorrect           52548492      # Number of conditional branches incorrect
588  system.cpu.branchPred.BTBLookups             496555083      # Number of BTB lookups
:
:
1249 system.cpu.kern.inst.quiesce                   3752574      # number of quiesce instructions executed
1250
1251 ---------- End Simulation Statistics   ----------
```

Fonte: Autor

Já o McPAT (LI, AHN, *et al.*, 2009) é usado para complementar o uso do gem5, pois uma das poucas informações que o gem5 não fornece é o consumo de energia. Entretanto, com o auxílio do McPAT é possível estimar o consumo de energia da aplicação simulada no gem5. Isso é possível fornecendo ao McPAT o arquivo de estatísticas da simulação (Figura 3.1) e mais um arquivo chamado “config.ini” que contém todas as caraterísticas da plataforma simulada no gem5 como, por exemplo, frequência, processador, tamanho das memorias, entre tantas outras.

A coleta de métricas de modelo e código é feita estaticamente, ou seja, sem a necessidade de executar a aplicação. Porém, para a coleta das métricas físicas é necessário simular a aplicação em uma plataforma de hardware para extrair os dados. Após todo o processo de extração de métricas, conforme ilustra a Figura 3.2, um grande conjunto de dados brutos é constituído, contendo informações deste a etapa de modelagem até a etapa do software (produto) em execução em uma plataforma. A partir deste momento se faz necessária a aplicação de uma metodologia que possa extrair conhecimento desses dados, ou em outras palavras, que possa encontrar informações relevantes que sejam capazes de relacionar as métricas de modelo, código e hardware.

Figura 3.2 – Processo de Extração das Métricas



Fonte: Autor

3.2.2 Seleção dos Dados

A fase de seleção de dados é a primeira propriamente dita do KDD e possui impacto significativo sobre a qualidade do resultado final, uma vez que nesta fase é escolhido o conjunto de dados contendo todas as possíveis variáveis (atributos) e registros (observações) que farão parte da análise, podendo ser um conjunto de dados ou um subconjunto de variáveis onde a extração será realizada.

Nessa primeira fase do processo de KDD, decidimos por selecionar, nos níveis de modelo e código, apenas as métricas de classe, desprezando métricas do métodos, pacotes e etc. Esta escolha foi feita porque o objetivo de predizer o desempenho da aplicação em nível de projeto é identificar e explorar diferentes alternativas na decomposição do problema em classes e, dessa forma encontrar a alternativa que atenda, ao mesmo tempo, os requisitos de qualidade de software (projeto modular que facilite reúso, manutenção e teste) e de desempenho. Assim, métricas relativas a métodos são detalhadas demais enquanto métricas de pacotes são abstratas demais para este momento da exploração do espaço de projeto.

Já com relação às métricas de hardware, nosso principal interesse era na métrica que nos fornecesse a energia consumida durante a execução. Entretanto também selecionamos algumas outras métricas de hardware, conforme apresentado na Tabela 2.1. Para fazer essa seleção sobre o conjunto de dados brutos, principalmente das métricas estáticas (modelo e código), foi necessário um grande esforço na programação de *scripts.*

Note que, no nível de modelo e código, as métricas são extraídas de todos os artefatos possíveis, logo, foi necessário identificar as métricas reverentes apenas às classes que existiam na estrutura do modelo e código. Além disso, uma única aplicação terá *n* valores de DIT, onde *n* será o número de classes que essa aplicação possui. Por outro, a mesma aplicação terá apenas uma métrica de hardware de cada tipo para toda aplicação, tornando deste modo a seleção dos dados mais simples no caso das métricas de hardware.

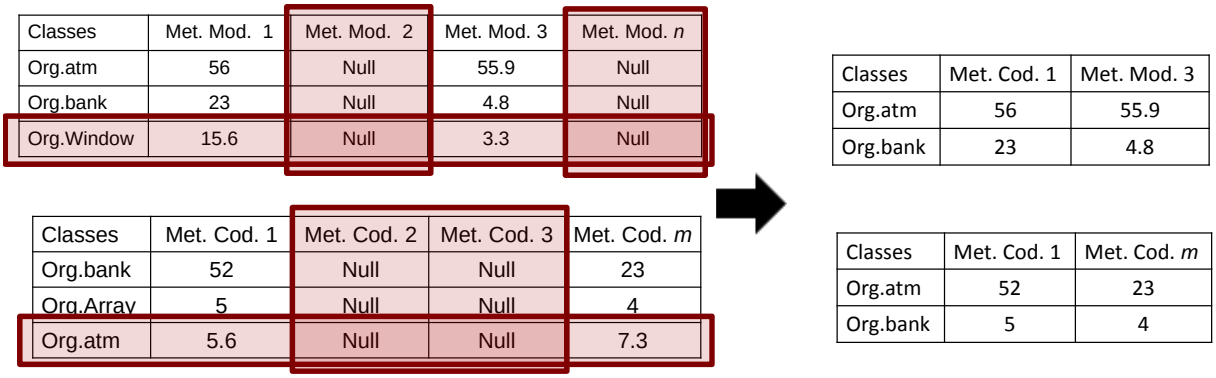
3.2.3 Limpeza e Pré-processamento

O pré-processamento e limpeza dos dados é outra parte crucial no processo de KDD, pois a qualidade dos dados vai determinar a eficiência dos algoritmos de mineração. Na etapa de limpeza e pré-processamento devem ser realizadas tarefas que eliminem dados redundantes e inconsistentes. Nesta implementação, apenas as métricas estáticas necessitaram passar pelo processo de limpeza, pois, para as métricas de hardware, todas as aplicações geram sempre o mesmo tipo de log, com exatamente os mesmos campos, sem nenhum atributo nulo ou faltante.

Entretanto, com relação às métricas de modelo e código, foi necessário excluir colunas com atributos nulos, por exemplo, quando essa coluna representava uma métrica que não era referente a uma métrica de classe, conforme a Figura 3.3, onde é ilustrado que a métrica de modelo 2 e a *n* não valores nulos, enquanto nas métricas de código isso ocorre com as métricas de 2 e 3. Note que, essa figura é um exemplo hipotético.

Além disso, foi necessária uma análise cruzada entre as métricas de modelo e código verificando a presença de classes que existiam em apenas um nível (código ou modelo). Nesses casos, os dados relativos à classe em questão são excluídos, conforme também ilustrado na Figura 3.3, neste caso pelas classes "*Org.Window*" e "*Org.atm*". Isso se tornou necessário pois em uma das análises propostas iremos tentar encontrar relações entre classes, ou seja, dada a métrica *X* de uma classe *C* em UML, qual será o valor ou a relação com uma métrica *Y* de código dessa mesma classe *C.*

Figura 3.3 – Pré-processamento das métricas estáticas



Fonte: Autor

3.2.4 Transformação dos dados

A Transformação dos dados é a fase do KDD que antecede a fase de mineração propriamente dita. Após serem selecionados, limpos e pré-processados, os dados necessitam ser armazenados e formatados adequadamente para que os algoritmos possam ser aplicados.

Dependendo do objetivo de mineração, será preciso preparar os dados de forma diferente. Com base nos dados que foram extraídos neste trabalho, três cenários possíveis de mineração foram explorados: 1) estimar as métricas de código a partir das métricas de modelo; 2) estimar as métricas de hardware a partir das métricas de código; 3) estimar as métricas de hardware a partir das métricas de modelo.

No cenário 1, o objetivo de mineração é estimar as métricas de código a partir de métricas de modelo. Neste cenário o processo de transformação será bem simples, pois, após os dados passarem pela etapa de pré-processamento (conforme a Figura 3.3), a etapa de transformação terá apenas que associar as métricas referentes a uma mesma classe nos níveis de modelo e código e gerar um único registro.

O resultado desse processo é ilustrado na Figura 3.4, onde cada linha (registro) representa uma classe, e as colunas (atributos) serão as métricas de cada registro, divididas em Variáveis Independentes (VI), ou seja, as métricas de modelos, e as Variáveis Dependentes (VD) que, por sua vez, serão as métricas de código. Esse tipo de abordagem aumentará muito o número de registros em nosso *dataset*, comparado ao número de aplicações, fator que possibilitará o uso de regressão múltipla, que por sua vez, será comentando com detalhes no capítulo 4.

Figura 3.4 – Estilo de Dataset para o Cenário 1

Classes	**VI 1**	**VI 2**	**...**	**VI *i***	**VD 1**	**VD 2**	**...**	**VD *j***
Classe 1	VI_1^1	VI_1^2	...	VI_1^i	VD_1^1	VD_1^2	...	VD_1^j
Classe 2	VI_2^1	VI_2^2	...	VI_2^i	VD_2^1	VD_2^2	...	VD_2^j
Classe *n*	VI_n^1	VI_n^2	...	VI_n^i	VD_n^1	VD_n^2	...	VD_n^j

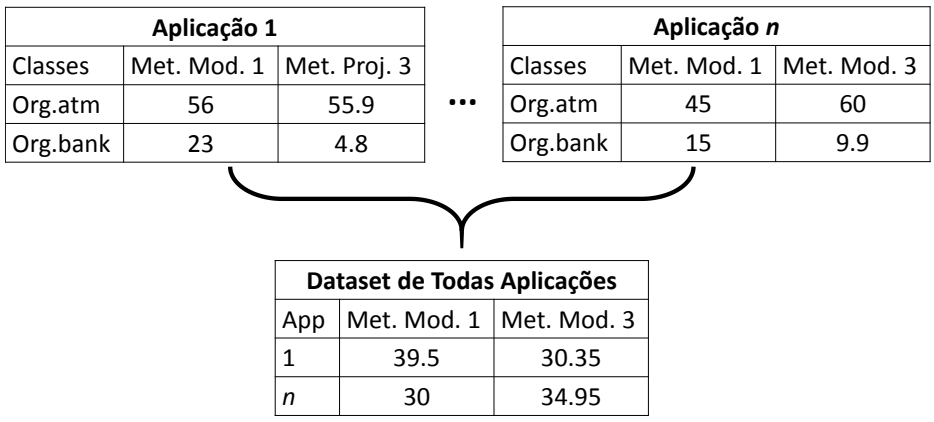
VI = Variável Independente
VD = Variável Dependente

Fonte: Autor

Nos cenários 2 e 3, as métricas de hardware assumiram o papel de variáveis dependentes e, respectivamente, as métricas de código e modelo assumiram o papel de as variáveis independentes.

Entretanto, como as métricas que serviriam como variáveis independentes são referentes às classes e não à aplicação como um todo, optou-se por fazer uma média dos valores mensurados a nível de classe para chegar a único valor de uma métrica de código e modelo por aplicação. Um exemplo simples desse processo é apresentado na Figura 3.5, onde o número de classes seria igual a dois. Note que, neste exemplo, são consideradas as métricas de modelo. Entretanto, o mesmo processo é aplicado também para as métricas de código.

Figura 3.5 – Exemplo da Primeira Etapa de Transformação dos Dados



Fonte: Autor

Deste modo, temos para cada aplicação um único valor para cada métrica, logo, resta apenas juntar as métricas que serão as variáveis independentes (recém transformadas) com as métricas que serão as variáveis dependentes (as métricas de hardware) em um único *dataset*. Tal processo resultará em um dataset semelhante ao ilustrado na Figura 3.6, onde cada linha (registro) representa agora uma aplicação, e as colunas (atributos) serão as métricas de cada registro, divididas em variáveis dependentes (métricas de hardware) e variáveis independentes que no cenário 2 foram as métricas de código e no cenário 3 as métricas de modelos.

Figura 3.6 – Estilo de Dataset para o Cenário 2 e 3

App	VI 1	VI 2	...	VI *i*	VD 1	VD 2	...	VD *j*
1	VI_1^1	VI_1^2	...	VI_1^i	VD_1^1	VD_1^2	...	VD_1^j
2	VI_2^1	VI_2^2	...	VI_2^i	VD_2^1	VD_2^2	...	VD_2^j
n	VI_n^1	VI_n^2	...	VI_n^i	VD_n^1	VD_n^2	...	VD_n^j

VI = Variável Independente
VD = Variável Dependente

Fonte: Autor

3.2.5 Escolha da Técnica de Data Mining

Após todo o processo de preparação dos dados, tem-se um *dataset* onde todos os valores são numéricos e o objetivo de mineração é estimar algumas métricas (hardware e/ou código) a partir da extração de conhecimento de outras métricas (modelo e/ou código). Logo, optamos por uma técnica de predição numérica, que visa descobrir um possível valor futuro de uma variável dependente. As predições numéricas visam prever valores para variáveis contínuas, diferente de técnicas de classificação que tentam prever variáveis discretas.

Os métodos mais conhecidos para predição numérica são as regressões. As técnicas de regressão modelam o relacionamento de variáveis independentes com uma variável dependente conforme já foi detalhado na subseção 2.2.3. Entretanto, conforme também destacado anteriormente, existem inúmeros algoritmos para análise de regressão que identificam relações distintas entre as variáveis. Neste trabalho, foi utilizada a linguagem/ambiente de programação R (R, 2014) que disponibiliza um bom conjunto de implementações de algoritmos de regressão.

3.2.5.1 Algoritmos de Regressão Escolhidos

A partir dos resultados da primeira abordagem apresentada na Seção 3.1, é razoável supor a existência de relações de diferentes naturezas entre as métricas de alto e baixo nível. Dessa forma, é interessante verificar como diferentes algoritmos de regressão modelam essas relações. Dessa forma, foram utilizados algoritmos distintos (tanto lineares como não-lineares) disponíveis em pacotes R para realizar a regressão e os resultados são comparados com base no erro de modelagem de cada algoritmo. Dessa forma, é possível escolher o melhor modelo (gerado pelo melhor algoritmo) para cada métrica de interesse. Neste trabalho, usamos sete diferentes algoritmos para análise de regressão. Todos algoritmos escolhidos fazem parte de pacotes do R (R, 2014) e foram escolhidos devido à familiaridade e facilidade de uso. Segue uma breve descrição dos algoritmos selecionados:

Regressão linear por mínimos quadrados (LM): Esta é uma das técnicas mais utilizadas e também mais simples de regressão linear onde a função alvo é estimada através da minimização da soma dos quadrados das diferenças entre os valores reais e os estimados, conforme já comentado na subseção 2.2.3.1. Como em todos os métodos lineares, o algoritmo supõe que existe uma relação linear entre os atributos preditores e o atributo alvo que se deseja estimar, o que pode tornar a técnica um pouco limitada. Para esta técnica foi utilizado o método "*lm*" do pacote "*Stats*" (The R Stats Package, 2014).

Modelo Linear Generalizado/Regressão de Poisson (*GLM*): É uma generalização flexível da regressão por mínimos quadrados, que permite que a variável alvo tenha modelos de distribuição de erro diferentes de uma distribuição normal. A regressão linear é generalizada, permitindo o modelo linear ser associado à variável alvo através de uma função de ligação e permitindo que magnitude da variância de cada medida seja uma função do seu valor previsto (MCCULLAGH and NELDER, 1989). Como descrição de distribuição de erro e função de ligação foi utilizada a regressão Poisson, que geralmente é utilizada para modelar contagem de dados e tabelas de contingência. Para a realização desse trabalho, foi utilizado o método *glm* do pacote "*Stats*" (The R Stats Package, 2014).

Máquinas de Vetor de Suporte (*SVM*): Máquinas de vetor de suporte são modelos de aprendizagem supervisionada que podem ser utilizados tanto para tarefas de classificação como para tarefas de regressão. A ideia básica do SVM consiste na construção de hiperplanos para

separação de dados linearmente separáveis e transformações de dados não linearmente separados através de uma função kernel, a qual foi utilizada uma função radial de grau 3. Para o trabalho foi utilizado o método "*svm*" do pacote "*e1071*" (MEYER, DIMITRIADOU, *et al.*, 2014).

Multivariate Adaptive Regression Splines (MARS): É uma técnica de regressão não paramétrica que pode ser vista como uma extensão de modelos lineares que modela não-linearidades e interações entre as variáveis automaticamente (FRIEDMAN, 1991). MARS não faz nenhuma hipótese sobre a relação funcional subjacente entre as variáveis dependentes e independentes. Em vez disso, MARS constrói essa relação de um conjunto de coeficientes e funções de base que são inteiramente "conduzidod" a partir dos dados de regressão. Num certo sentido, o método é baseado no paradigma "dividir para conquistar", que divide o espaço de entrada em regiões, onde cada uma terá a sua própria equação de regressão. Isso faz com que o MARS seja adequado para problemas com dimensões grandes (ou seja, com mais de 2 variáveis), onde a dimensionalidade provavelmente criaria problemas para outras técnicas. Neste trabalho foi utilizado o pacote "*Earth"* (MILBORROW, 2014) do R para construir o modelo.

Árvores de Regressão (CART[3]): Árvores de regressão são parecidas com árvores de decisão (ROKACH and MAIMON, 2008), na verdade uma árvore de regressão utiliza uma árvore de decisão como um modelo preditivo que mapeia as folhas para previsões numéricas ao invés de decisões, ou seja, a análise de árvores de regressão é quando o resultado previsto pode ser considerado um número real (por exemplo, uma métrica de software). Para o trabalho foi utilizado o pacote *rpart* (THERNEAU, ATKINSON and RIPLEY, 2014). Neste trabalho, as árvores geradas não foram podadas e para realizar a divisão dos nodos foi utilizado o método anova.

Redes Neurais (MLP): As redes neurais artificiais (RNA) também são vistas como modelos paramétricos não-lineares. No trabalho utilizou-se uma rede *Multilayer Perceptron* (MLP) com o algoritmo *Backpropagation*. Para a minimização dos erros foi utilizado o algoritmo BFGS, com *weight decay* de 0,0001. Configurou-se para que a camada oculta tivesse

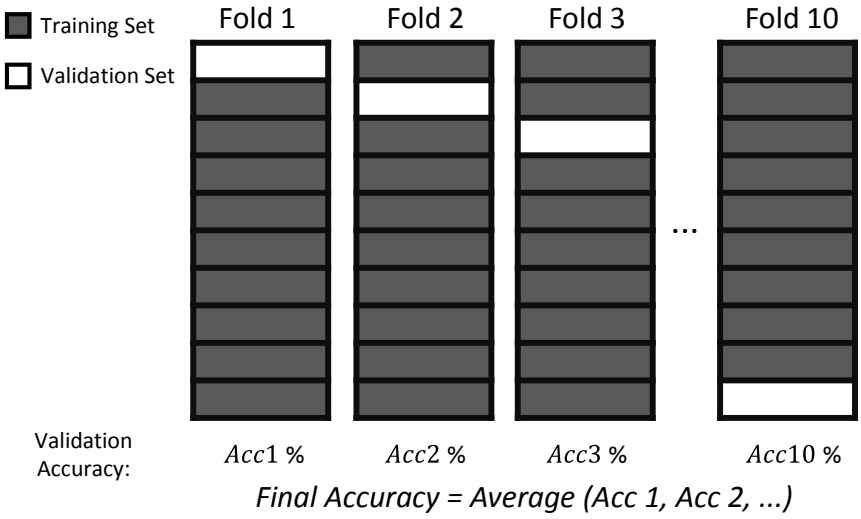
[3] O termo CART é um termo genérico usado para se referir a árvore de classificação e árvore de regressão.

20 neurônios, e o treinamento da rede fosse feito em até 1.000 iterações. Para o trabalho foi utilizado o pacote "*nnet"* (RIPLEY and VENABLES, 2014).

Random Forest (RF): O algoritmo Random Forest constrói uma floresta de árvores de decisões, cada uma utilizando subconjuntos aleatórios das variáveis candidatas a preditoras escolhendo, como modelo final, a árvore que apresenta o melhor desempenho (HO, 1995). Neste trabalho foram utilizadas florestas com 500 árvores. Foi utilizado o pacote "*randomForest*" (BREIMAN, CUTLER, *et al.*, 2014) também do R.

Para avaliar os modelos preditivos gerados pelos diferentes algoritmos, foi utilizada a técnica de *10-fold cross-validation* (KOHAVI, 1995). Nesta técnica a amostra (*dataset*) é randomicamente particionada em dez subamostras mutuamente exclusivas do mesmo tamanho. Em seguida, uma única subamostra (das dez definidas) é retirada para ser usada como elemento de teste (validação), e as noves subamostras restantes são usadas como dados de treinamento e então calcula-se a acurácia do modelo. Esse processo é repetido dez vezes, com cada umas das dez subamostras sendo utilizada como dado de teste uma vez. Logo, deve ser calculada a média entre as acurácias de cada etapa do *cross-validation* para se chegar em uma única taxa de erro. A Figura 3.7 ilustra esse processo.

Figura 3.7 – 10-Fold Cross-Validation



Fonte: (VIEIRA, FAUSTINI, *et al.*, 2014)

A técnica de *cross-validation* tem a capacidade de avaliar a generalização de um modelo, evitando possíveis vieses no processo de estimação. Isso se dá pois não é fixada apenas uma única porção do dataset para a avaliação, ou seja, todo conjunto de dados será utilizando tanto

para teste como também para treinamento. Garante-se assim, de certa forma, uma maior generalização do modelo preditivo, principalmente quando comparado a um método tradicional que pegaria 70% dos dados para treino e 30% para teste (KOHAVI, 1995).

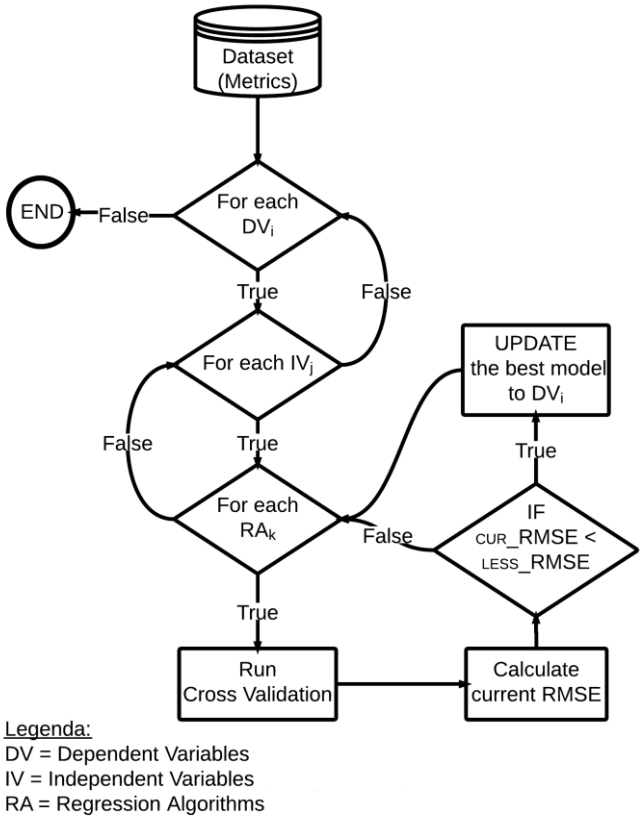
A cada etapa do *cross validation*, a acurácia é medida pelo RMSE (ver Equação 2.4), que indica o quão perto os valores previstos estão dos valores observados (reais)**.** É calculado então um total de dez RMSE's para cada um dos 7 algoritmos, para então finalmente escolher o modelo preditivo com menor RSME médio, gerado pelo algoritmo que melhor se ajustou aos dados.

A partir desse momento dois cenários são possíveis: 1) utilizar todas as métricas disponíveis como variáveis independentes, ou seja, uma regressão múltipla; 2) utilizar apenas uma variável independente, neste caso, uma regressão simples. Neste segundo cenário é preciso escolher qual métrica terá melhor desempenho como variável preditora, ou seja, para cada requisito não funcional (variável independente) que se queira estimar, deve-se criar um modelo para cada possível combinação de métrica independente, métrica dependente e algoritmo de regressão, para então selecionar o modelo preditor com menor taxa de erro.

Esse processo de treinamento pode ser visto na Figura 3.8. A partir de um dataset seleciona-se uma métrica dependente, ou seja, uma métrica que se quer estimar. Então, para cada métrica independente, criam-se modelos preditivos com todos os algoritmos considerados e armazena-se apenas o modelo que apresenta menor RMSE. Por fim, o processo se repete para todas as métricas que se queira estimar. No caso de uma regressão múltipla, as métricas independes não seriam selecionadas uma a uma, e sim todas ao mesmo tempo.

.

Figura 3.8 – Processo de Treinamento para Regressão Simples

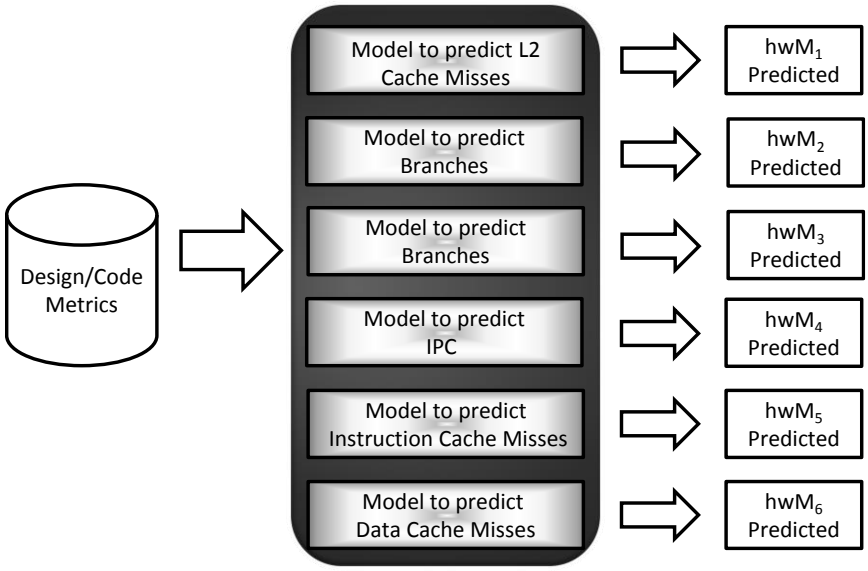


Fonte: (VIEIRA, FAUSTINI, *et al.*, 2014)

Após todo o processo de treinamento, o framework terá escolhido os melhores modelos preditivos criados com os diferentes algoritmos e variáveis independentes. Assim, tem-se um modelo preditivo para cada variável alvo que for do interesse do projetista. A Figura 3.9 ilustra como seria resultado deste treinamento visando estimar, por exemplo, as métricas de hardware.

Com os modelos preditivos prontos, em princípio espera-se principalmente poder estimar as métricas de hardware de forma antecipada. Por exemplo, quando o projetista de software embarcado estiver modelando um software em alto nível, ele poderia extrair as métricas dos diagramas e estimar os requisitos não funcionais, tendo assim, um *feedback* de forma antecipada no ciclo no desenvolvimento do software, e, se porventura ele precisar realizar alguma alteração, seria um processo menos custoso, pois ele não teria avançado para etapas mais detalhadas do projeto, onde necessitaria de um esforço maior para alteração do projeto.

Figura 3.9 – Modelos Preditivos após Treinamento



Fonte: (VIEIRA, FAUSTINI, *et al.*, 2015)

No próximo capítulo serão apresentados nossos resultados experimentais que visam validar nosso framework.

4 RESULTADOS EXPERIMENTAIS

4.1 *SETUP* EXPERIMENTAL (PREPARAÇÃO DAS APLICAÇÕES E EXTRAÇÃO DAS MÉTRICAS)

A validação da metodologia proposta para estimação dos requisitos não funcionais será feita a partir de três conjuntos de aplicações. Cada conjunto consiste em diferentes implementações elaboradas a partir de uma mesma especificação. As implementações foram desenvolvidas por alunos da graduação dos cursos de Ciência e Engenharia de Computação da UFRGS na disciplina de Técnicas de Construção de Programas. A partir de uma mesma especificação, diferentes grupos de alunos projetaram e implementaram diferentes versões da aplicação proposta. Todas as implementações utilizadas nos experimentos atendem aos mesmos requisitos funcionais propostos na especificação, mas representam diferentes decisões de projeto e implementação. Todos os projetos foram desenvolvidos segundo o paradigma de orientação a objetos e seguindo três etapas básicas do ciclo de desenvolvimento de software: modelagem do sistema em UML; codificação em Java; e elaboração de testes funcionais. O primeiro conjunto de aplicações é composto por 21 implementações de um sistema bancário com funcionalidades básicas sobre uma conta corrente (saldo, transferência, depósito, etc). Esta é uma aplicação tipicamente orientada a dados ou do tipo CRUD (*create-read-update-delete*), uma vez que cada operação no sistema altera o estado da conta corrente do usuário.

O segundo conjunto de aplicações consiste em 12 implementações de um sistema de votação para reuniões não presenciais de colegiados. Nesta aplicação, alguns usuários (chefia do colegiado) podem incluir um conjunto de documentos e uma solicitação de votação enquanto os outros usuários (membros do colegiado) podem revisar os documentos e votar. Esta aplicação é basicamente orientada a controle pois prevê basicamente interações com o usuário e pouca atividade sobre dados.

O terceiro conjunto de aplicações é composto por 11 implementações de um sistema básico de comércio eletrônico. Nesta aplicação o usuário pode pesquisar, selecionar e comprar itens de dois tipos de produtos (vestuário e eletroeletrônicos). Essa aplicação envolve tanto atividades de controle (interação com o usuário para busca e seleção de itens) quanto de dados (gerenciamento de estoque, carrinho de compras, etc).

Estas aplicações foram escolhidas basicamente por três razões. Em primeiro lugar, elas representam sistemas com características distintas (mais orientada a dados, mais reativa e mista,

respectivamente). Em segundo lugar, para todos os projetos tivemos acesso a diferentes implementações que realmente representam decisões distintas de modelagem e codificação a partir de uma única especificação. Por último, as implementações foram feitas por alunos com diferentes níveis de conhecimento em projetos orientados a objetos. Deste modo, as implementações disponíveis apresentam, naturalmente, atributos de qualidade de projeto bastante distintos.

A coleta de métricas de modelo e código é feita estaticamente, ou seja, sem a necessidade de executar a aplicação. Porém, para a coleta das métricas físicas é necessário simular a aplicação em uma plataforma de hardware. Essa simulação implica a existência de um conjunto de estímulos ou, em outras palavras, em um conjunto de testes funcionais automatizados. Para que uma comparação justa entre as diferentes implementações seja possível, é importante que os estímulos de entrada (as funcionalidades simuladas) sejam os mesmos para todas as versões da aplicação. Além disso, é importante garantir que o resultado da simulação não seja determinado por uma única parte do código.

Assim, antes de coletar qualquer métrica, algumas normalizações foram realizadas em praticamente todas aplicações. Essas normalizações foram necessárias por três motivos: 1) para deixar as aplicações com as mesmas operações do ponto de vista do usuário; 2) para que todas as aplicações de cada conjunto aceitassem exatamente os mesmos dados de entrada; 3) para garantir um nível mínimo de cobertura de execução de todo código.

Para analisar a cobertura das execuções foi utilizado o plugin EclEmma (Java Code Coverage for Eclipse, 2006) para Eclipse. Essa ferramenta analisa, a partir da execução de um conjunto de testes, quais partes do código foram ou não foram executadas. Essa informação ajuda a identificar novos testes que exercitem partes do código ainda não cobertas por nenhum teste. A Tabela 4.1 apresenta os dados de cobertura de código fornecidos pelo EclEmma para todas implementações de todas as aplicações. Em média, mais de 90% de todo código fonte foi executado pelos testes gerados. Algumas aplicações tiveram uma taxa de cobertura um pouco menor, devido ao fato de terem boa parte do código dedicado a tratar exceções. Esses trechos não foram estimulados propositadamente para facilitar o uso do simulador da plataforma de hardware.

Tabela 4.1 – Cobertura de código para os conjuntos de testes funcionais usados na simulação

Imp.	Sistema Bancário	Sistema de Votação	Comercio Eletrônico
01	77,0%	90,1%	88,7%
02	91,3%	91,1%	92,8%
03	84,1%	89,8%	91,0%
04	92,6%	90,7%	95,8%
05	86,1%	89,9%	89,1%
06	91,9%	88,0%	90,6%
07	87,3%	91,8%	87,9%
08	89,5%	91,6%	89,9%
09	88,5%	87,1%	94,2%
10	83,5%	96,9%	93,1%
11	75,7%	95,0%	92,6%
12	95,2%	96,8%	
13	82,0%		
14	94,0%		
15	94,7%		
16	94,6%		
17	97,0%		
18	96,6%		
19	91,9%		
20	88,8%		
21	76,1%		
Média	**89,0%**	**91,5%**	**91,4%**

Fonte: Autor

Após a normalização de todas as implementações, as métricas de cada nível foram extraídas para cada versão de cada aplicação. As métricas de modelos foram extraídas a partir dos diagramas de classes UML originais (elaborados pelos alunos) que modelam toda a estrutura e relações entre classes. Conforme já apresentado no capítulo anterior, as métricas dos diagramas de classes foram extraídas com auxílio da ferramenta SDMetrics (WÜST, 2014). A extração das métricas de código se deu pelo uso da ferramenta Understand (TOOLWORKS, 2014), conforme destacado no capítulo 3. Por fim, o simulador gem5 (BINKERT, BECKMANN, *et al.*, 2011) foi utilizado para extração das métricas dinâmicas. A plataforma embarcada definida no simulador gem5 foi a seguinte: processador ARMv7 Cortex-A15 1.0 GHz com 64 KB de

cache L1 (32 KB para cache de instruções, 32 KB cache de dados) e 1MB de cache L2. Cada implementação foi executada no simulador usando o mesmo conjunto de testes funcionais e, para cada simulação, as métricas de hardware fornecidas pelo gem5 foram coletadas.

A Tabela 4.2 exemplifica como cada implementação de uma mesma especificação (Sistema Bancário neste caso) pode ser diferente, tanto do ponto de vista de tamanho e qualidade de software, quanto do ponto de vista de desempenho na plataforma alvo. Na tabela, cada coluna representa uma métrica e foram escolhidas duas métricas de cada nível (modelo, código e hardware, respectivamente). Cada linha representa uma implementação distinta do Sistema Bancário. As métricas extraídas dos diagramas de classes foram o número de classes (NC), que indica o tamanho da implementação, e o número de métodos públicos (NMP), que pode ser um indicador da qualidade do código do ponto de vista de encapsulamento. Pode-se notar, em relação a essas métricas, uma variação de quatro vezes ou mais entre as diferentes implementações. O mesmo acontece para as métricas de código LOC (número de linhas de código fonte) e LCOM (indicador de falta de coesão). Nessa última métrica, valores baixos indicam módulos mais coesos e, portanto, melhor qualidade interna do software. Pode-se observar na tabela que, por exemplo, a implementação 15 apresenta um nível de coesão (9.5 %) bem melhor que a implementação 10 (60,5%). As métricas de hardware são apresentadas nas duas últimas colunas da Tabela 4.2. O número de *misses* na cache L2 é um dado fornecido diretamente pelo gem5 e a energia é estimada pelo McPAT com base nas informações geradas pelo gem5. Novamente, é possível notar que a variação para energia e *misses* da cache L2 ocorre por um fator de 1,97 e 2,35 respectivamente. Este dado confirma o que já foi reportado em outros trabalhos: um sistema pode consumir mais de duas vezes a quantidade de energia realmente necessária para sua execução, dependendo de como é feita a implementação.

O mesmo tipo variação pode ser observado nos outros dois sistemas. Por exemplo, nas implementações do sistema de comercio eletrônico nota-se uma variação de 610 a 1273 na métrica LOC e uma variação por um fator de 1,67 no consumo de energia.

Embora apenas duas métricas de cada nível tenham sido usadas para exemplificar a variação entre as implementações, foram coletadas, no total, 19 métricas de modelo (Tabela A 1) em anexo), 39 métricas de código (Tabela B 1, Tabela B 2 e Tabela B 3 em anexo) e 7 métricas de hardware (Tabela 2.1) para cada uma das 44 implementações disponíveis. As métricas de código e modelo ainda foram extraídas de cada classe de cada aplicação.

Tabela 4.2 – Compacto de Métricas Extraídas das Versões do Sistema Bancário

Imp.	Métricas de Modelo		Métricas de Código		Métricas de Hardware	
	NC	NPM	LOC	LCOM	L2 Missses	Energia (mW)
01	20	79	615	34.3%	8,237,759	4546.121
02	20	108	882	46.6%	10,601,839	3098.227
03	15	83	822	32.5%	11,346,185	4300.066
04	11	93	601	34.2%	7,394,867	3784.24
05	13	86	879	52.4%	15,257,805	3894.887
06	13	78	809	49.5%	9,631,411	4030.917
07	16	73	1,101	51.4%	10,157,944	4442.694
08	11	57	995	50.0%	9,943,623	3950.377
09	16	101	761	47.7%	10,953,311	4832.087
10	15	92	931	60.3%	9,409,752	3756.130
11	14	70	942	46.1%	11,302,669	3592.989
12	12	123	1,001	52.7%	9,534,775	5241.942
13	11	110	1,184	51.4%	8,838,877	3474.389
14	19	74	738	50.6%	11,650,151	3118.913
15	21	44	712	9.5%	9,368,276	3982.241
16	9	32	381	35.4%	7,041,519	3142.246
17	8	34	315	19.8%	6,485,546	2660.449
18	22	28	530	17.7%	8,217,963	3632.594
19	12	60	750	40.4%	11,126,327	5077.998
20	16	83	940	41.6%	8,955,259	4271.184
21	33	102	830	17.1%	7,668,811	3213.211

Fonte: Autor

A partir deste conjunto de dados iniciais, o processo de estimação pode acontecer de três formas: i) usando as métricas de modelo para estimar métricas de código; 2) usando as métricas de código para estimar as métricas de hardware; ou 3) usando as métricas de modelo para estimar as métricas de hardware. A terceira opção é a mais interessante do ponto de vista de exploração do espaço de projeto pois alterações no nível de modelagem são mais simples, rápidas e mais eficazes que alterações sobre o código. No entanto, as três possibilidades foram estudadas neste trabalho de forma incremental. Se a primeira opção se mostrar factível, ela já pode trazer algum ganho ao projetista de software em relação ao tempo de projeto, pois ele poderá avaliar o impacto que uma escolha/alteração feita no modelo resultará no código fonte. Por outro lado, diagramas de classe são bastante próximos do código fonte e sabe-se da

literatura (JIANG, CUKI, *et al.*, 2008) (SAMI and FAKHRAHMAD, 2010), que existe uma relação direta entre as métricas de modelo e de código. Sendo assim, a análise deste cenário de uso serviu para uma primeira validação da metodologia e sua implementação. Os outros dois cenários representam cenários de uso mais realistas para aplicações embarcadas.

4.2 TÉCNICA DE VALIDAÇÃO

4.2.1 Estimação das Métricas de Código

Para validar a abordagem visando estimar as métricas de código, os dados usados como *dataset* foram preparados conforme descrito no cenário 1 da transformação dos dados, ou seja, cada instância (registro/observação) do *dataset* é formada por dados referentes a uma classe e não à aplicação como um todo, conforme a Figura 3.4. Deste modo, além de possivelmente ter-se uma estimativa mais precisa (pois são estimados valores de métricas de classes a partir de métricas de classes), também ter-se-á um número muito maior de observações para treinamento. Por exemplo, o *dataset* com as métricas do Sistema Bancário tem 327 instâncias, que é o número total de classes da Tabela 4.2.

Segundo Harrel (2002), para cada variável independente que se queira incorporar no treinamento do modelo preditivo, é necessário um mínimo de 10 a 20 observações, para que o modelo preditivo resultante tenha um nível de significância estatística aceitável. Assim, ter um número expressivo de registros no *dataset* é importante pois permitirá que todas as métricas dos diagramas sejam usadas como variáveis preditoras (independentes), ou seja, a análise pode ser feita usando regressão múltipla.

A Tabela 4.3 apresenta os resultados da validação da abordagem proposta no cenário de estimação das métricas de código. Foram realizados testes com os três *datasets* de forma independente, ou seja, utilizando cada *dataset* de forma individual, foram criados modelos preditivos para cada métrica de código utilizando todas as métricas de diagramas como variáveis independentes.

Conforme mostrado no fluxograma apresentado na Figura 3.8 para cada *dataset*, uma métrica de interesse (variável dependente) é selecionada por vez e nesse caso todas as métricas de diagrama são consideradas como variáveis independentes. Com isso, são criados 7 modelos

preditivos para cada métricas de interesse, ou seja, um modelo para cada algoritmo de regressão. No fim desse processo, as taxas de erro (RMSE) dos 7 modelos são avaliadas e o modelo com melhor precisão é escolhido.

A Tabela 4.3 mostra o RMSE normalizado dos melhores modelos preditivos criados para cada dataset. O NRMSE (do inglês, *Normalized* RMSE) é o RMSE dividido pelo intervalo de valores observados, conforme expresso na Equação 4.1. Este erro é expresso como uma porcentagem, logo ele ajuda em uma melhor visualização dos resultados, por ser independente de unidade.

$$NRMSE = \frac{RMSE}{X_{obs,\max} - X_{obs,\min}} \qquad \text{Equação 4.1}$$

Na Tabela 4.3 também é apresentado o algoritmo que atinge o melhor resultado e define, portanto, o modelo de predição para cada métrica de interesse. É notório que diferentes algoritmos alcançam melhores resultados em diferentes situações, dependendo do tipo métrica e do tipo de aplicação, ou seja, dependendo do dado analisado. Este resultado comprova a premissa da metodologia proposta de que diferentes relações entre as métricas devem ser pesquisadas.

Um resumo da quantidade de vezes que cada algoritmo gerou o melhor modelo pode ser visto na Tabela 4.4. Dentre os 117 modelos preditivos criados (uma para cada métrica de código e para cada dataset), o algoritmo Random Forest é o melhor modelo em quase 54% dos casos (63 modelos usam este algoritmo). Por outro lado, se apenas este algoritmo for usado como modelo preditivo, em 54 outras oportunidades o erro de previsão seria maior do que os apresentados na Tabela 4.3.

Um exemplo onde o algoritmo Random Forest não apresentou o melhor resultado é ilustrado na Figura 4.1, que representa todo processo de treinamento do modelo preditivo para a métrica *CountDecLMethodAll*. O eixo x apresenta os valores estimados pelos modelos e o eixo y mostra os valores reais obtidos. Os algoritmos são representados por diferentes cores e formas geométricas. Neste diagrama de dispersão, o ideal é que os pontos estejam o mais perto possível da linha diagonal entre x e y, pois isso quer dizer que o valor predito é exatamente (ou quase) igual ao valor observado. Por outro lado, quanto mais o ponto se afasta da diagonal, mas distante o valor estimado está do valor real. Logo, pela Figura 4.1, é possível notar mais pontos do algoritmo RF longe da diagonal, do que os pontos do algoritmo MARS, por exemplo.

Tabela 4.3 – Resultados para Estimação das Métricas de Código por Especificação

#	Métricas Estimadas	Sistema Bancário		Sistema de Votação		E-commerce	
		NRMSE	Alg.	NRMSE	Alg.	NRMSE	Alg.
1	AvgCyclomatic	2.43%	RF	3.12%	RF	1.84%	SVM
2	AvgCyclomaticModified	4.55%	CART	4.06%	RF	2.43%	SVM
3	AvgCyclomaticStrict	2.50%	RF	4.43%	RF	3.11%	SVM
4	AvgEssential	3.35%	RF	2.12%	RF	7.60%	RF
5	MaxCyclomatic	6.50%	LM	3.30%	RF	3.08%	RF
6	MaxCyclomaticModified	7.88%	LM	3.49%	RF	3.77%	RF
7	MaxCyclomaticStrict	5.54%	LM	2.62%	RF	3.60%	SVM
8	MaxInheritanceTree	1.48%	CART	0.21%	MARS	0.20%	MLP
9	MaxNesting	6.65%	LM	9.00%	RF	10.69%	LM
10	SumCyclomatic	2.56%	RF	2.99%	RF	4.78%	RF
11	SumCyclomaticModified	2.40%	RF	2.53%	MLP	5.19%	RF
12	SumCyclomaticStrict	2.34%	RF	2.60%	RF	4.87%	RF
13	SumEssential	1.97%	RF	1.64%	RF	3.27%	RF
14	AvgLine	3.96%	RF	6.10%	RF	4.41%	RF
15	AvgLineBlank	5.25%	RF	8.86%	RF	5.03%	RF
16	AvgLineCode	3.30%	RF	4.51%	RF	4.32%	RF
17	AvgLineComment	2.46%	SVM	2.35%	SVM	1.06%	SVM
18	CountLine	4.99%	RF	2.64%	LM	7.33%	RF
19	CountLineBlank	7.39%	LM	2.22%	MARS	7.57%	RF
20	CountLineCode	2.97%	LM	3.18%	RF	5.67%	RF
21	CountLineCodeDecl	5.29%	LM	2.88%	LM	5.28%	LM
22	CountLineCodeExe	3.11%	LM	3.11%	RF	5.38%	RF
23	CountLineComment	3.41%	SVM	0.90%	RF	2.25%	RF
24	CountSemicolon	3.89%	LM	2.84%	MARS	4.55%	RF
25	CountStmt	3.31%	LM	3.04%	MARS	5.02%	RF
26	CountStmtDecl	5.49%	LM	1.90%	LM	4.56%	LM
27	CountStmtExe	2.93%	LM	3.18%	RF	4.59%	RF
28	CountClassCoupled	8.15%	RF	4.04%	MLP	10.13%	RF
29	CountClassDerived	0.00%	LM	0.33%	MARS	0.34%	LM
30	CountDeclClassMethod	2.84%	RF	1.76%	RF	3.30%	RF
31	CountDeclClassVariable	2.37%	MARS	4.39%	RF	4.34%	RF
32	CountDeclInstanceMethod	6.89%	RF	1.32%	MARS	4.28%	RF
33	CountDeclInstanceVariable	4.22%	RF	4.64%	LM	3.73%	LM
34	CountDecLMethod	0.93%	LM	1.08%	MARS	2.06%	LM
35	CountDecLMethodAll	0.43%	MARS	1.50%	MARS	2.80%	RF
36	CountDecLMethodPrivate	4.41%	LM	0.44%	MARS	1.62%	MLP
37	CountDecLMethodProtected	1.33%	SVM	0.72%	RF	1.06%	RF
38	CountDecLMethodPublic	0.97%	LM	1.53%	MARS	1.86%	LM
39	PercentLackOfCohesion	12.99%	RF	9.03%	RF	5.66%	RF

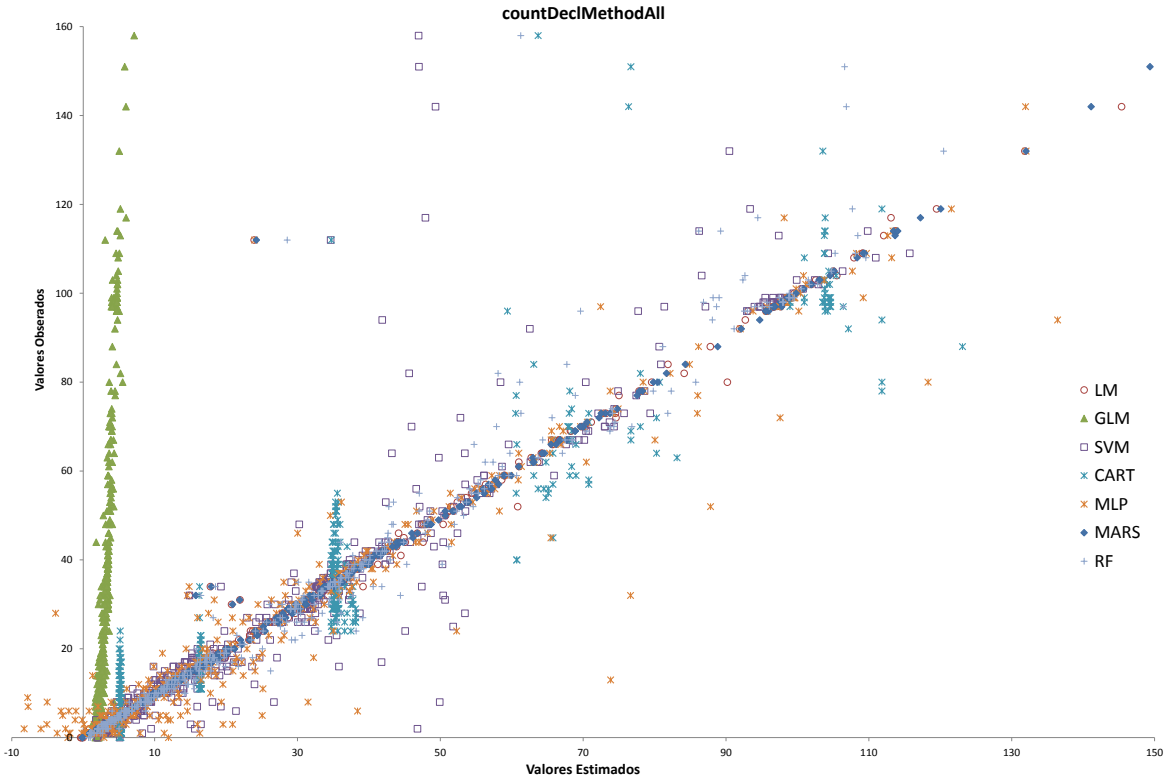
Fonte: Autor

Tabela 4.4 – Algoritmos escolhidos

Algoritmo	**LM**	**SVM**	**RF**	**CART**	**MARS**	**MLP**	**GLM**
Sistema Bancário	16	3	16	2	2	0	0
Sistema de Votação	4	1	22	0	10	2	0
E-commerce	7	5	25	0	0	2	0
Total	**27**	**9**	**63**	**2**	**12**	**4**	**0**

Fonte: Autor

Figura 4.1 – Diagrama de Dispersão



Fonte: Autor

Considerando a proximidade entre um diagrama de classe e sua própria codificação já se esperava taxas de erro bastante baixas, e isso de fato aconteceu. Por exemplo, a métrica de código *CountClassDerived* que mensura a mesma coisa que a métrica de modelo NOC, a tendência é que o erro da estimação dessa métrica tenda a zero e conforme a Tabela 4.3 isso realmente acontece. Outro exemplo no mesmo sentido é a métrica *MaxInheritanceTree* que mensura a mesma coisa que a métrica de modelo DIT, onde a taxa de erro foi praticamente zero

também. O motivo para estimação dessas métricas não ser realmente perfeita pode se dar por dois motivos: 1) inconsistência entre modelo e código, ou seja, o que está no modelo não reflete exatamente o que está no código, provavelmente o código foi alterado e o modelo não; 2) Alguma outra métrica que está sendo usada como variável independente pode estar inserindo ruído no modelo.

Por outro lado, algumas métricas de código não têm uma relação direta com as medidas extraídas do modelo, por exemplo, as complexidades ciclomáticas, e mesmo para essas métricas a abordagem proposta ainda mantém uma taxa de erro bem baixa, sendo a maior 7,8% e a menor 1,8%. Logo, o projetista de software poderia avaliar o impacto de possíveis alterações no projeto com um alto nível de abstração, tendo em vista o quanto tais alterações iriam impactar na qualidade do código fonte, com uma precisão maior que 90% em praticamente todos atributos.

Conforme pode ser notado, a definição dos *datasets* com as métricas de cada tipo de aplicação em separado, para o treinamento dos modelos preditivos, pressupõe um cenário de evolução de um sistema único, ou seja, os modelos de estimação gerados serão usados para avaliar o impacto de uma operação de manutenção (preventiva ou corretiva) ou evolução (adição de funcionalidades) do sistema. Pois nesse caso a base do software (sua estrutura geral) se mantém e a versão modificada é ainda muito parecida com a anterior, provavelmente mais parecida do que as diferentes implementações definidas para uma mesma especificação. Os resultados dentro deste contexto mostram que o erro de predição é realmente baixo e o uso da abordagem proposta pode fornecer informação confiável ao projetista.

No entanto, nem sempre existe, dentro de uma organização, uma grande quantidade de versões ou implementações do mesmo sistema, o que inviabiliza o uso da metodologia proposta. Além disso, é interessante pensar na estimativa de desempenho nas primeiras etapas de um novo projeto, para o qual não existe ainda nenhuma implementação disponível. Nessa situação, uma solução pode ser usar um cenário de treinamento mais heterogêneo, onde diferentes aplicações (diferentes funcionalidades) definem um modelo preditivo para uma nova aplicação. Essa solução busca expor, talvez, características que diferentes projetos possam ter em comum e que possam indicar os atributos de uma futura implementação. Tal solução será viável, porém, apenas se o erro de previsão for aceitável.

Os resultados apresentados na Tabela 4.5 derivam deste raciocínio. Neste experimento, propõe-se a união de dois ou dos três *datasets* originais em um único conjunto que é usado para

o treinamento dos modelos preditivos. Na Tabela 4.5 são apresentados os dados da validação cruzada para o *dataset* composto, respectivamente, pelas métricas do sistema bancário e de votação juntos (B + V), pelos dados do sistema bancário e de comércio eletrônico (B + E), pelos dados do sistema de votação e comércio eletrônico (V + E) e, por último, por todos os dados misturados em um único *dataset* (B + E + V).

Logo, o objetivo desse experimento foi investigar a possibilidade de criar modelos preditivos mais genéricos sem perder precisão, ou seja, não ter vários modelos preditivos, um para cada tipo de aplicação, mas sim um único modelo preditivo mais abrangente, partindo de um treinamento incremental com informações de diferentes aplicações. Os resultados apresentados na Tabela 4.5 demostram que é possível tornar o modelo preditivo mais genérico, sem perder muita precisão ou, até melhorá-la em alguns casos. Considere, como exemplo a métrica *AVGEssential*. O menor erro de predição para esta métrica (usando como *dataset* apenas as métricas do sistema bancário) é de 3.35%. Este cai para 2,84% quando o modelo é definido a partir das métricas do sistema bancário e do sistema de votação juntos. Por outro lado, o erro de treinamento usando apenas o sistema de votação é menor (2,12%). Note, porém, que o aumento do erro em relação ao melhor caso é pequeno e, principalmente, a taxa de erro é realmente baixa, gerando ainda uma informação confiável para o projetista.

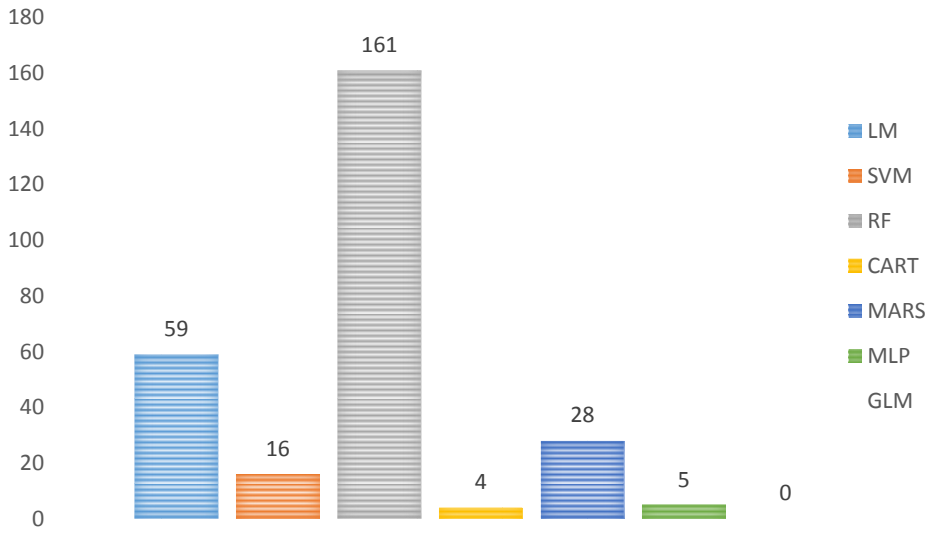
Um resultado que se consolida ainda mais é com relação ao algoritmo *Random Forest*, que novamente apresenta um número maior de melhores resultados. A Figura 4.2 representa o número total de seleções de cada algoritmo, considerando todos os resultados apresentados (Tabela 4.3 e Tabela 4.5). Pode-se notar que o RF se manteve como o algoritmo que mais vezes foi selecionado como melhor modelo preditivo. Este resultado pode indicar que, na falta de condições para treinamento e geração de modelos preditivos específicos, o algoritmo RF pode ser uma boa escolha.

Tabela 4.5 – Resultados para os Datases Heterogêneos

Métricas Estimadas	B + V		B + E		E + V		B + E + V	
	NRMSE	Alg.	NRMSE	Alg.	NRMSE	Alg.	NRMSE	Alg.
AvgCyclomatic	4.18%	RF	4.59%	RF	2.89%	RF	3.90%	RF
AvgCyclomaticModified	5.49%	RF	3.80%	SVM	3.21%	RF	4.26%	RF
AvgCyclomaticStrict	4.41%	RF	4.48%	SVM	2.26%	SVM	4.41%	RF
AvgEssential	2.84%	RF	3.92%	RF	6.73%	RF	2.98%	RF
MaxCyclomatic	5.49%	RF	4.95%	LM	4.00%	RF	4.91%	RF
MaxCyclomaticModified	6.27%	RF	5.72%	LM	4.20%	RF	5.11%	RF
MaxCyclomaticStrict	6.44%	RF	4.75%	LM	4.72%	RF	5.16%	RF
MaxInheritanceTree	0.37%	MARS	0.29%	RF	0.23%	MARS	0.27%	MARS
MaxNesting	5.80%	RF	5.06%	SVM	9.86%	RF	5.20%	RF
SumCyclomatic	2.85%	RF	2.82%	LM	2.21%	RF	2.53%	RF
SumCyclomaticModified	2.78%	RF	2.26%	RF	2.26%	RF	2.49%	RF
SumCyclomaticStrict	2.92%	RF	2.10%	RF	2.58%	RF	2.32%	RF
SumEssential	2.51%	RF	1.78%	RF	2.44%	MARS	2.94%	LM
AvgLine	5.24%	RF	4.88%	RF	4.24%	RF	4.38%	RF
AvgLineBlank	5.40%	RF	5.67%	RF	4.61%	RF	5.04%	RF
AvgLineCode	4.82%	RF	4.13%	RF	4.59%	RF	3.80%	RF
AvgLineComment	4.20%	RF	2.74%	SVM	1.78%	SVM	2.97%	RF
CountLine	3.84%	LM	4.37%	RF	2.72%	LM	4.31%	LM
CountLineBlank	4.57%	LM	4.69%	LM	2.17%	MARS	3.22%	LM
CountLineCode	3.29%	RF	2.62%	RF	2.18%	RF	2.97%	RF
CountLineCodeDecl	3.81%	LM	5.22%	LM	2.64%	LM	3.22%	LM
CountLineCodeExe	3.17%	RF	2.45%	RF	1.94%	RF	2.28%	RF
CountLineComment	4.14%	RF	3.84%	SVM	1.60%	MARS	4.27%	RF
CountSemicolon	4.06%	RF	3.85%	LM	1.98%	RF	2.96%	RF
CountStmt	3.64%	RF	3.58%	LM	3.24%	MARS	3.00%	RF
CountStmtDecl	3.87%	LM	5.51%	LM	2.46%	LM	3.38%	MARS
CountStmtExe	3.74%	RF	2.33%	RF	1.90%	RF	2.52%	RF
CountClassCoupled	6.39%	RF	7.92%	RF	7.14%	RF	9.10%	RF
CountClassDerived	0.16%	CART	0.05%	MARS	0.09%	MARS	0.06%	MARS
CountDeclClassMethod	1.52%	RF	3.42%	RF	1.71%	RF	1.50%	RF
CountDeclClassVariable	1.12%	CART	2.08%	MARS	4.92%	RF	1.63%	MARS
CountDeclInstanceMethod	2.38%	LM	4.57%	RF	1.37%	LM	2.02%	LM
CountDeclInstanceVariable	4.78%	RF	3.80%	RF	4.80%	RF	4.80%	RF
CountDecLMethod	0.88%	MARS	1.08%	LM	0.77%	LM	0.99%	MARS
CountDecLMethodAll	1.75%	LM	7.90%	RF	6.62%	RF	7.52%	RF
CountDecLMethodPrivate	2.05%	LM	4.48%	LM	1.08%	MLP	1.32%	LM
CountDecLMethodProtected	1.40%	RF	2.20%	RF	2.84%	RF	2.18%	RF
CountDecLMethodPublic	1.75%	LM	1.11%	LM	1.00%	LM	1.10%	MARS
PercentLackOfCohesion	15.31%	RF	8.65%	RF	8.46%	RF	15.35%	RF

Fonte: Autor

Figura 4.2 – Seleção de Todos os Modelos



Fonte: Autor

4.2.2 Estimação das Métricas de Hardware

Para validar a abordagem proposta visando estimar as métricas de hardware, os dados usados como *dataset* foram preparados conforme descrito nos cenários 2 e 3 da Seção 3.2.4 (transformação dos dados), ou seja, cada registro do *dataset* é composto por métricas referentes a uma aplicação como um todo, onde optou-se por fazer uma média dos valores mensurados a nível de classe (ao invés de se ter uma métrica para cada classe de cada implementação).

Pelo fato de termos um número menor de registros no *dataset* nestes cenários, apenas uma métrica de modelo ou código pode ser usada como variável preditora, ou seja, neste caso, será usada a análise de regressão simples. Neste contexto, além de encontrar o melhor algoritmo de predição, é preciso também considerar que diferentes métricas de modelo ou código podem apresentar diferentes relacionamentos com as métricas de hardware. Dessa forma, a metodologia proposta define a métrica de modelo (ou código) e o algoritmo de regressão que predizem, com a menor margem de erro possível, o valor para cada métrica de hardware de interesse. No processo de treinamento são criados então 7xN modelos de predição para cada métrica de hardware, para N o número de métricas de alto nível (modelo ou código). Assim, considerando as 39 métricas de código presentes no *dataset* tem-se 273 modelos possíveis para

cada métrica de hardware. Cada modelo é uma combinação de uma métrica de alto nível e um algoritmo de regressão para uma métrica de hardware. Dentre esses modelos, aquele com o menor RMSE é escolhido como o modelo preditivo para a métrica de hardware em questão. Da mesma forma, considerando as 19 métricas de modelo presentes no *dataset*, tem-se 133 possíveis modelos de predição para cada métrica de hardware.

A Tabela 4.6 apresenta os melhores resultados (os modelos com menor RMSE) para um *dataset* composto apenas pelas métricas do sistema bancário e assumindo como variáveis independentes, respectivamente, as métricas de modelo e de código. Dos 2842[4] modelos criados no total (7x(7x39) + 7x(7*19)), apenas os 14 melhores são apresentados na tabela. A primeira coluna indica a métrica de hardware que o modelo se destina a estimar. As próximas três colunas apresentam, respectivamente, a métrica de modelo que melhor se ajustou para prever a métrica de hardware, o algoritmo que juntamente com esse métrica chegou no melhor resultado, e por último, a taxa de erro da operação de validação cruzada. Já as três últimas colunas representam as mesmas informações em relação à métrica de código que melhor se relacionou com a de hardware.

Tabela 4.6 – Estimação das Métricas de Hardware para o Sistema Bancário

Métricas de HW	**Aplicação 1 (Sistema Bancário)**					
	Diagrama de Classes			**Código Fonte**		
	Métrica	Alg.	Erro	Métrica	Alg.	Erro
L2 Misses	Dep_In	SVM	0.32%	CountDeclMethodAll	LM	1.43%
Predicted Branches	Dep_Out	RF	1.66%	CountDeclInstanceVariable	RF	1.00%
Missed Branches	NumOps	SVM	0.34%	CountDeclMethodAll	SVM	1.09%
IPC	NumOps	SVM	0.37%	MaxInheritanceTree	SVM	0.39%
Icache Misses	Dep_Out	SVM	1.50%	AvgLineCode	RF	2.07%
Dcache Misses	NumOps	SVM	1.04%	CountClassDerived	RF	0.27%
Energy	DIT	SVM	0.94%	SumEssential	LM	1.15%

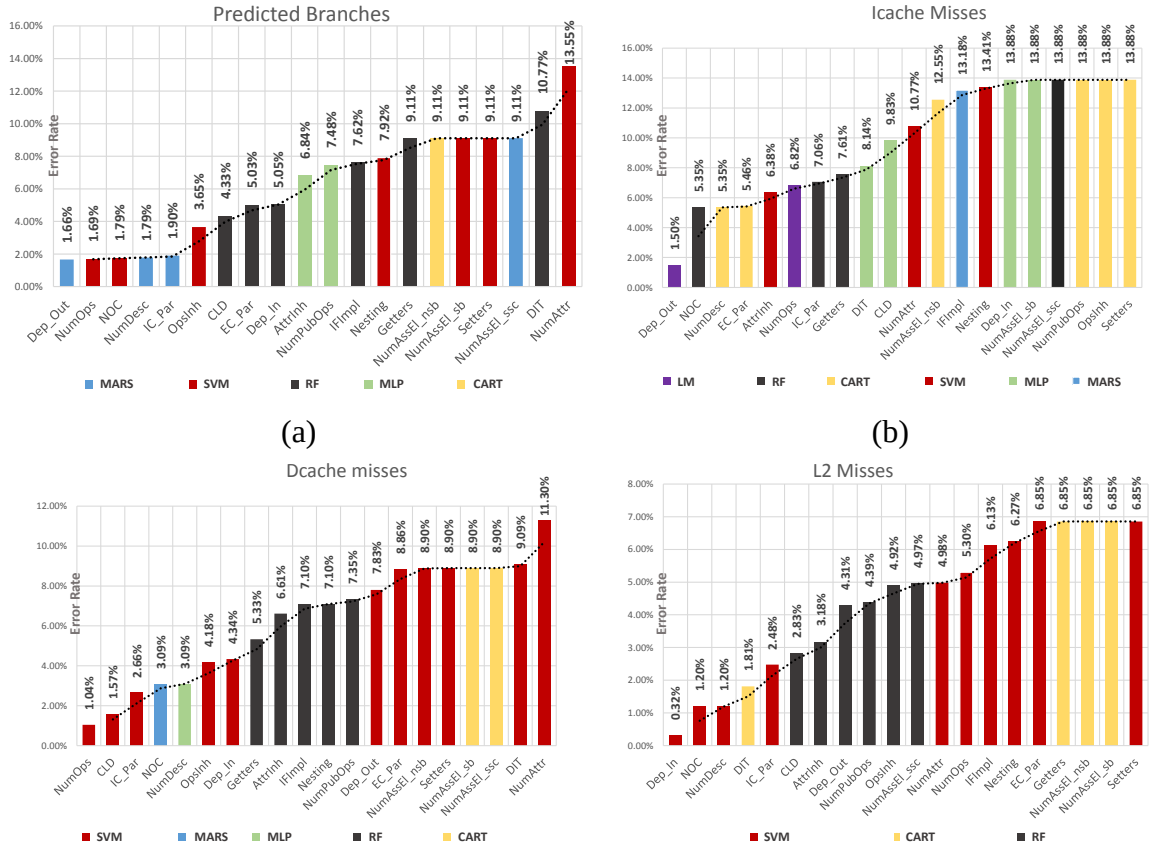
Fonte: Autor

É possível notar na Tabela 4.6 uma taxa de erro muito baixa para todos os casos (máximo de 2.07%), mesmo considerando uma previsão a partir do nível mais alto. Isso se deu principalmente por três razões. Quando analisadas em um âmbito global (uma métrica por

[4] Sem considerar que cada modelo é criado e treinado 10 vezes pelo 10-fold cross-validation

implementação ao invés de uma métrica por classe) notam-se os valores para as métricas mais próximos entre elas do que em relação às métricas de classes. Isso deixa o modelo preditivo mais específico, logo, a taxa de erro para aplicações semelhantes tende a ser baixa. Em segundo lugar, mas ainda na mesma linha, é possível atribuir esse excelente resultado ao fato do *dataset* ser relativamente pequeno nesse experimento, mais uma vez contribuindo para que os modelos sejam, de certa forma, mais específicos. Assim, a utilização desses modelos para estimar métricas de uma nova aplicação totalmente diferente deverá apresentar uma taxa de erro bem mais alta. Por último, o fato de não escolher apenas o melhor algoritmo mas sim a melhor combinação entre um algoritmo e uma métrica (de código ou modelo) que melhor se relaciona com uma métrica de hardware. De fato, o resultado de uma má escolha para a variável independente pode prejudicar muito os resultados finais do experimento, conforme pode ser visto na Figura 4.3.

Figura 4.3 – Seleção dos Modelos para Aplicação 1



Fonte: Autor

A Figura 4.3 apresenta os resultados durante os treinamentos dos modelos preditivos a partir das métricas de modelo para estimar: a) o número de *branches* previstos; b) o número de *misses* na cache de instruções; c) o número de *misses* na cache de dados; d) o número de *misses* da cache L2. Nestas figuras são apresentados os melhores resultados (algoritmos com menor RMSE) para a combinação entre métrica de hardware e modelo. A diferença entre os algoritmos selecionados se dá pela cor das barras.

Em todos os casos mostrados na Figura 4.3 é notório que uma escolha pouco informada sobre a métrica usada como variável independente pode gerar erros até 10 vezes maiores que o mínimo possível, mesmo considerando que o melhor algoritmo tenha sido selecionado. Esta figura corrobora a premissa inicial sobre a necessidade de uma exploração mais detalhada dos relacionamentos entre métricas de alto e baixo nível.

Ainda é apresentada a mesma análise para os *datasets* contendo, respectivamente, as métricas dos sistemas de Votação e de Comércio Eletrônico na Tabela 4.7 e na Tabela 4.8 respectivamente. Nestas duas tabelas, a taxa de erro é ainda menor, algo já esperado, pois esses *datasets* possuem uma quantidade de dados um pouco menor que o anterior, tornando o modelo mais específico.

Tabela 4.7 – Estimação das Métricas de Hardware para o Sistema de Votação

Métricas de HW	**Aplicação 2 (Sistema de Votação)**					
	Diagrama de Classes			**Código Fonte**		
	Métrica	Alg.	Erro	Métrica	Alg.	Erro
L2 Misses	NumDesc	SVM	0.02%	CountClassDerived	SVM	0.01%
Predicted Branches	NumOps	SVM	0.19%	AvgCyclomaticModified	SVM	0.09%
Missed Branches	CLD	SVM	0.04%	MaxInheritanceTree	SVM	0.05%
IPC	NumAttr	MARS	0.54%	CountDeclClassVariable	RF	0.22%
Icache Misses	NumPubOps	SVM	0.48%	MaxInheritanceTree	SVM	0.04%
Dcache Misses	Dep_Out	SVM	0.00%	AvgCyclomaticStrict	SVM	0.01%
Energy	Setters	SVM	0.31%	CountDeclMethodPrivate	SVM	0.12%

Fonte: Autor

Tabela 4.8 – Estimação das Métricas de Hardware para o E-commerce

Métricas de HW	**Aplicação 3 (E-commerce)**					
	Diagrama de Classes			**Código Fonte**		
	Métrica	Alg.	Erro	Métrica	Alg.	Erro
L2 Misses	NumPubOps	CART	0.29%	SumCyclomaticStrict	SVM	0.14%
Predicted Branches	NumAssEl_nsb	LM	0.19%	CountClassDerived	MLP	0.00%
Missed Branches	CLD	RF	0.39%	CountDeclClassVariable	SVM	0.00%
IPC	Dep_In	MLP	0.44%	MaxNesting	LM	0.14%
Icache Misses	NumAttr	SVM	0.08%	AvgCyclomaticModified	LM	0.18%
Dcache Misses	AttrInh	SVM	0.07%	CountStmtDecl	MLP	0.05%
Energy	NumOps	SVM	0.14%	MaxInheritanceTree	RF	0.17%

Fonte: Autor

Os resultados descritos acima confirmam a validade da metodologia proposta para um cenário de manutenção e evolução de um sistema. Neste caso, o projetista pode avaliar o impacto de alterações feitas no modelo do sistema rapidamente e com alta confiabilidade. Esta análise pode ser feita da seguinte forma: após alterar o projeto do sistema, extraem-se as métricas de modelo que foram identificadas como as melhores opções para a previsão das métricas de execução de interesse. Usam-se então as métricas extraídas como entradas para os modelos preditivos gerados durante o treinamento. O modelo informa o valor esperado para cada métrica de hardware e o projetista pode avaliar se a alteração feita teve impacto negativo no desempenho do sistema em uma dada plataforma de hardware. Considerando a precisão dos modelos preditivos gerados (pelo menos 98% em todos os experimentos), a estimativa feita já na etapa de modelagem é bastante confiável, sem a necessidade de codificação, *deployment*, depuração e execução na plataforma alvo. Neste sentido, a metodologia proposta oferece ao projetista uma realimentação rápida e passível de ação imediata: o projetista pode alterar o projeto e fazer novas estimativas até que o impacto no desempenho final esteja dentro dos limites aceitáveis.

Apesar dos resultados promissores, a questão da especificidade do modelo preditivo deve ser considerada com cuidado. Conforme explicado, uma das razões para o baixo erro de previsão está relacionada à similaridade dos dados de entrada. Apesar do uso da técnica de *10-cross validation*, se os dados de todo o *dataset* são muito similares, qualquer sub-conjunto usado na etapa de teste será, na prática, muito similar ao conjunto do treinamento.

O experimento que utiliza um *dataset* misto (ou seja, usando métricas de mais de uma aplicação) pode trazer luz sobre essa questão. Assim, semelhantemente ao cenário descrito na

subseção 4.2.1, foram avaliadas todas as diferentes combinações possíveis de *datasets* (B+V, B+E, V+E, B+V+E).

A Tabela 4.9 mostra o resultado do treinamento com a junção das métricas dos sistemas Bancário e de Votação. Conforme esperado, o erro aumentou, pois os modelos preditivos são mais genéricos. Por outro lado, os modelos gerados são mais confiáveis pois há maior diferença entre os subconjuntos usados para treinamento e teste de cada modelo. Ainda assim, a precisão mínima dos modelos gerados é da ordem de 93%. Considerando uma estimativa feita nas primeiras etapas de projeto, sem a necessidade de codificação, essa taxa de erro é ainda bastante aceitável.

As demais combinações foram todas testadas e, como esperado, os resultados são semelhantes aos da Tabela 4.9, não existindo nenhuma taxa de erro maior que 7%. Logo, para evitar dados redundantes, essas tabelas não foram inseridas no texto. É importante ressaltar que, embora mais genéricos, os modelos gerados ainda assumem o cenário de evolução e manutenção dos sistemas usados no treinamento. Este tipo de treinamento apenas reduz a chance de *overfitting* do modelo.

Tabela 4.9 – Estimação das Métricas de Hardware para Sistema bancário e Sistema de Votação juntos

Métricas de HW	Sistema Bancário e Sistema de Votação					
	Diagrama de Classes			**Código Fonte**		
	Métrica	Alg.	Erro	Métrica	Alg.	Erro
L2 Misses	NumAssEl_sb	CART	2.90%	AvgLineCode	SVM	5.25%
Predicted Branches	DIT	RF	5.19%	CountDeclClassMethod	CART	4.51%
Missed Branches	NumAssEl_sb	RF	6.18%	CountLineCodeExe	MLP	2.62%
IPC	DIT	MLP	0.91%	MaxInheritanceTree	MLP	1.18%
Icache Misses	AttrInh	SVM	3.66%	CountDeclClassMethod	MARS	6.19%
Dcache Misses	DIT	RF	6.02%	CountDeclClassMethod	CART	4.93%
Energy	CLD	SVM	6.15%	AvgCyclomatic	RF	3.57%

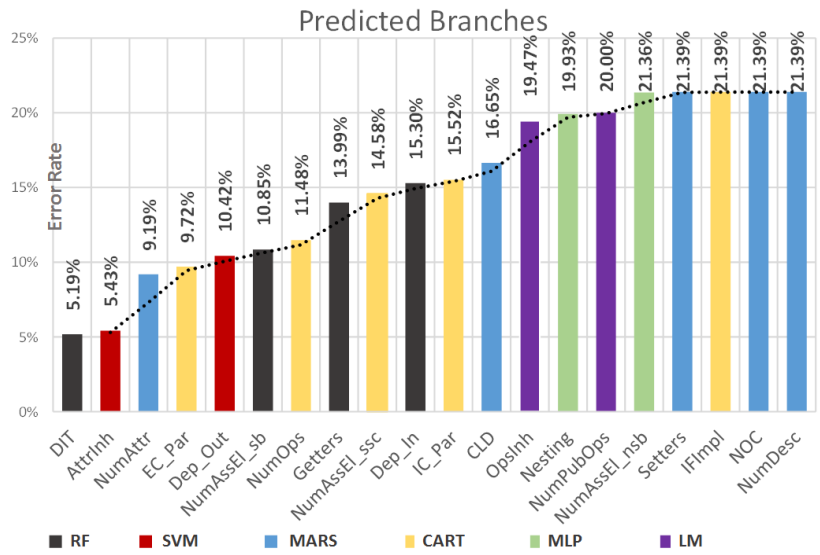
Fonte: Autor

A Figura 4.4 apresenta os resultados parciais (durante o treinamento) dos modelos preditivos para a métrica de execução "número de *branches* previstos" e usando as métricas de modelo como variáveis independentes. O *dataset* é formado por métricas do Sistema Bancário e do Sistema de Votação juntas (Tabela 4.9). Esta figura, em conjunto com a Figura 4.3(a), tem

o objetivo de ilustrar que, mesmo criando um modelo para a mesma métrica de hardware, dependendo dos dados, ou seja, do histórico do projeto, métricas diferentes podem se relacionar melhor e, consequentemente, gerar resultados melhores. Na Figura 4.4 (ou Tabela 4.9) é possível notar que a melhor métrica de modelo que se relaciona com *branches* previstos é a DIT, já na Figura 4.3(a) (ou Tabela 4.6) é possível ver que *branches* previstos se relaciona melhor com a métrica Dep_Out.

A mesma observação também se aplica aos algoritmos selecionados. Em grande parte das combinações entre a métrica de hardware e todas as métricas de modelo, os algoritmos que obtêm os melhores resultados são diferentes. Por exemplo, na Figura 4.3 (a) o algoritmo que melhor relaciona a métrica *Dep_Out* é o MARS, já na Figura 4.4 é o SVM.

Figura 4.4 – Seleção dos Modelos para a Métrica *Predicted Branches* da Tabela 4.9



Fonte: Autor

Como último experimento, foi elaborado um cenário visando analisar o uso da abordagem proposta para estimação dos requisitos não funcionais de um projeto novo, para o qual nenhuma implementação ou projeto estejam disponíveis. Ou seja, assume-se uma situação onde não há dados históricos de aplicações semelhantes e deseja-se observar a acurácia dos modelos preditivos nesta situação. Para tal, os modelos foram treinados com os dados de duas aplicações (sistema bancário e sistema de Votação) e testado com todas as métricas da aplicação do sistema de Comércio Eletrônico. Neste caso, não foi utilizada a técnica de *cross-validation*. As taxas de erro para esse cenário podem ser vistas na Tabela 4.10.

Neste cenário a taxa de erro aumentou bastante comparado às situações anteriores. Isso mostra que, de fato, as taxas de erro dos modelos gerados no cenário de evolução são bem específicas deste tipo de cenário. Entretanto, considerando o uso dos modelos preditivos para estimação de um projeto totalmente novo, ou seja, sem dados históricos de implementações semelhantes, onde erro máximo é menor do que 20%, dependendo do tipo de projeto pode ser aceitável, ou ao menos, servir de guia para o projetista de software poder fazer escolhas melhores dentro da taxa de erro.

Considere, por exemplo, a métrica de energia consumida, utilizando os modelos preditivos disponíveis, o projetista pode estimar rapidamente o consumo de energia esperado para uma nova aplicação, em uma dada plataforma de hardware, com uma margem de 13% de erro, antes mesmo de executar, simular ou mesmo escrever o código fonte da nova aplicação. Esta análise inicial pode ser feita para diferentes opções de projeto de classes (diferentes decomposições do sistema) e pode, inclusive, ajudar na definição do projeto da própria plataforma de hardware (indicando a necessidade de uma versão diferente do processador para melhorar o IPC, por exemplo). Um erro de 13% neste momento do projeto pode ser bem mais aceitável do que o esforço de medida e refatoração que uma análise tardia do desempenho pode gerar. Logo, para fins de guiar o projetista de software a tomar melhores decisões nas etapas inicias do projeto com relação a produto final, a taxa de erro gerada pela metodologia proposta pode ser considerada aceitável, pois ela já permite avaliar como o produto irá se comportar em um hardware específico de forma rápida, eficaz e com razoável precisão, desde as etapas iniciais do projeto. Além disso, à medida que o projeto é desenvolvido, a estimativa pode ser refeita a partir das métricas de código, permitindo uma ação assim que um problema de desempenho seja detectado.

Tabela 4.10 – Estimação das Métricas de Hardware da Aplicação Comércio Eletrônico utilizando modelos treinados com as métricas das aplicações Sistema Bancário e de Votação

Métricas de HW	**Treinamento com as Aplicações Sistema Bancário e Sistema de Votação Teste com a Aplicação Comércio Eletrônico**					
	Diagrama de Classes			**Código Fonte**		
	Métrica	Alg.	Erro	Métrica	Alg.	Erro
L2 Misses	NumDesc	MLP	19.18%	PercentLackOfCohesion	CART	18.09%
Predicted Branches	NumAssEl_nsb	MLP	16.08%	CountClassCoupled	LM	15.06%
Missed Branches	NumAssEl_nsb	CART	14.76%	CountDeclMethodAll	RF	12.09%
IPC	NumAssEl_sb	CART	2.79%	CountLineCode	CART	3.16%
Icache Misses	Dep_Out	SVM	17.85%	CountClassCoupled	LM	12.05%
Dcache Misses	NumAssEl_nsb	CART	11.95%	CountClassCoupled	CART	11.39%
Energy	NOC	CART	12.87%	CountClassDerived	CART	12.56%

Fonte: Autor

4.3 ANÁLISE DOS RESULTADOS

A abordagem proposta visivelmente se adapta melhor em um cenário onde já se tenha um histórico de métricas de aplicações semelhantes. Por exemplo, se um engenheiro precisa prestar manutenção a um software que já tem várias versões, ele pode usar a abordagem proposta para treinar um modelo com base nas versões anteriores e então depois utilizar esses modelos para avaliar o impacto de suas decisões tomadas ainda nas fases iniciais do projeto em relação aos requisitos não funcionais do software. Deste modo, não é necessário esperar até a implantação do software para constatar, por exemplo, que o consumo de energia está exagerado, quando a refatoração é mais difícil, custosa e de menor impacto.

Como exemplo de uso da abordagem proposta, considere dois projetos alternativos (imp02 e imp03) como possíveis evoluções de um projeto original (imp01) do sistema de Comércio Eletrônico. Considere ainda que cada versão do novo projeto apresenta atributos de qualidade de software distintos. Embora seja interessante optar pelo projeto de melhor qualidade do ponto de vista de software, sabe-se que, em muitos casos, o custo em hardware desta qualidade é pior. Assim, o projetista deseja achar o melhor compromisso entre qualidade de software e desempenho no hardware. Considere, por exemplo, a métrica de consumo de energia como fator

crucial neste projeto. A partir da Tabela 4.8, sabe-se que, para este sistema, o consumo de energia se relaciona melhor com a métrica de modelo *NumOps*. Assim, para cada projeto alternativo, extrai-se esta métrica e calcula-se, usando o modelo de predição correspondente, o provável consumo de energia para cada caso.

A Tabela 4.11 apresenta os dados relativos a este exemplo. O valor da métrica do diagrama de classes usada como entrada no modelo preditivo é dado na segunda coluna e a energia estimada pelo modelo é apresentada na terceira coluna da tabela. As variações percentuais das métricas extraídas com respeito ao projeto original também são mostradas.

Com essas informações, o desenvolvedor pode refletir e escolher a melhor solução. Se um menor consumo de energia é absolutamente necessário, a primeira alternativa (imp02) é a melhor para esta evolução. Se, porém, um menor valor para numOps é interessante para este projeto (a evolução pode ser uma refatoração visando melhorar a qualidade do projeto), o projetista pode avaliar que o impacto no consumo de energia, embora menor, ainda é positivo, ou seja, o projeto melhorou e a energia consumida diminuiu. Para validação, a quarta coluna da Tabela 4.11 mostra o consumo real de energia para cada alternativa de projeto (obtido através da simulação do código correspondente usando o gem5 configurado conforme explicado anteriormente), enquanto que a última coluna apresenta a diferença entre o valor estimado e o valor real. Pode-se observar que, de fato, o consumo real é bastante similar ao valor previsto.

Tabela 4.11 – Uso da Abordagem Proposta para Estimar Energia – Cenário de Evolução

Opção de Modelagem	**Métrica de Modelo Extraída (NumOps)**	**Energia Estimada (mJ)**	**Energia Realmente Consumida (mJ)**	**Resíduo (mJ)**
imp01	3.5	4294.177	4270.207	23.97
imp02	5.79 (+39%)	3506.084 (-18.3%)	3431.016 (-19.6%)	75.06
imp03	2.69 (-23%)	3817.796 (-11%)	3743.042 (-12.3%)	74.75

Fonte: Autor

Resumindo, nossa abordagem mostrou-se reagir muito bem em um cenário de manutenção ou evolução de um sistema. Por outro lado, conforme apresentado na Tabela 4.10, mesmo com dados de estimação não tão precisos, ainda podemos utilizá-la em um cenário de projeto totalmente novo, pelo menos para ter uma ideia (mesmo que não tão precisa) de como o software poderá vir a se comportar no hardware.

Também foi possível comprovar que diferentes algoritmos de regressão podem alcançar diferentes resultados dependendo dos dados com que eles estão trabalhando. E por fim,

corroborando com os resultados apresentados como nossa motivação (seção 3.1), não foi possível encontrar uma relação entre métrica de alto nível e baixo nível que sempre exista na mesma magnitude. Fica claro, que esta relação irá depender do tipo de aplicação que estiver em análise.

4.3.1 Ameaças à Validade

Umas das principais ameaças a validade desta abordagem é o problema de *overfitting*. O *overfitting* acontece quando o modelo estatístico se ajusta em demasiado ao *dataset*. É comum que os dados apresentem desvios causados por erros ou fatores aleatórios, e quando o modelo se ajusta a estes fatores temos um modelo com *overfitting*. Um modelo com este problema apresenta alta precisão quando testado com esses dados, porém tal modelo não é uma boa representação da realidade e por isso deve ser evitado.

Uma das maneiras de reduzirmos a chance de ocorrer *overfitting* é ter um *dataset* com várias observações, pois com um número maior de observações, as medidas que apresentam erros de medição e *outliers* tendem a ser amortizadas, diminuindo assim, a chance de *overfitting*. Existem outras técnicas que visam diminuir as chances de *overfitting*, entre elas o *cross-validation* que foi utilizado.

Além do tamanho pequeno dos *datasets* para estimação das métricas de hardware, outro contratempo é no que diz respeito à plataforma de hardware, que em nosso caso, foi fixado apenas uma. Entretanto, poderiam e deveriam ser avaliadas diferentes configurações de hardware, mas devido a um limite de tempo, não foi possível analisar diferentes configurações de hardware neste trabalho.

5 CONCLUSÕES E TRABALHOS FUTUROS

Essa dissertação apresentou uma metodologia capaz de auxiliar o projetista de software embarcado na implementação de um sistema de qualidade dentro de um tempo de projeto mais curto, sendo capaz de estimar os requisitos não funcionais do software embarcado, por exemplo energia, desde as etapas iniciais do ciclo de desenvolvimento de software. Essa metodologia é baseada no processo de KDD, que permite a extração de conhecimento previamente desconhecido e potencialmente útil a partir de um grande conjunto de dados.

Como núcleo do processo de KDD é utilizada a técnica de análise de regressão. Por essa técnica são criados modelos preditivos capazes de estimar os requisitos não funcionais a partir de um processo de treinamento baseado em métricas de software. Usando a abordagem proposta, o impacto de diferentes alternativas de projeto para uma plataforma embarcada podem ser rapidamente analisados antes da implantação.

Os resultados experimentais mostraram a aplicabilidade da abordagem, principalmente para evoluções e manutenções de projetos, onde já se tenha um histórico de métricas de aplicações semelhantes para serem usadas como dados de treinamento. Entretanto, mesmo com dados de estimação não tão precisos, ainda pode-se utilizar a abordagem proposta em um cenário de um projeto totalmente novo, ou seja, onde os modelos preditivos são treinados para um tipo de sistema mas usados na estimativa de um sistema completamente novo. Neste cenário, o erro máximo de previsão é menor que 20%, o que ainda pode ser considerado aceitável para uma primeira estimativa de projeto.

Também foi possível comprovar que diferentes algoritmos de regressão podem alcançar diferentes resultados dependendo dos dados que eles estão manipulando. Por fim, corroborando os resultados apresentados como motivação para este trabalho, não foi possível ainda encontrar uma relação exata entre uma métrica de alto nível e outra de baixo nível. Resultados experimentais mostram que o tipo de relação existente entre as métricas depende do tipo de aplicação que estiver em análise.

Até o momento os resultados atingidos se mostraram promissores, sendo aceitos por diferentes comunidades (VIEIRA, FAUSTINI and COTA, 2014) (VIEIRA, FAUSTINI, *et al.*, 2014) , (VIEIRA, FAUSTINI, *et al.*, 2015). Por outro lado, este trabalho abre espaço para diferentes linhas de investigação, tais como:

- Avaliar métricas de diferentes artefatos do ciclo de desenvolvimento de software.
- Expandir o conjunto de aplicações para extração das métricas.
- Aplicar e avaliar novas técnicas de mineração de dados.
- Identificar métricas preditoras promissoras, aplicando técnicas de *feature selection.*
- Explorar o comportamento das aplicações em diferentes configurações de hardware.
- Investigar a possibilidade da criação de novas métricas tanto de software quanto de hardware (e.g. métricas de hardware por classe).
- Desenvolver um framework que auxilie o projetista na gestão de sistemas embarcados, visando tomar as melhores decisões de projeto desde suas fases iniciais de desenvolvimento, onde o projetista terá:
    - um ambiente para modelagem do software;
    - geração automática de partes do código a partir da modelagem;
    - extração automática das métricas estáticas;
    - integração com o simulador Gem5 para extração de métricas dinâmicas;
    - treinamento de modelos preditivos automaticamente;
    - sugestões de alterações com base no prévio conhecimento já adquirido.

REFERÊNCIAS

AGRAWAL, R.; SRIKANT, R. Fast Algorithms for Mining Association Rules in Large Databases. In: 20TH INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 1994. **proceedings**. Santiago: Morgan Kaufmann. p. 487-499.

BINKERT, N. et al. The Gem5 Simulator. **IGARCH Comput. Archit. News**, v. 39, n. 2, p. 1-7, 2011.

BOEHM, B. W. et al. **Characteristics of software quality**. Amsterdam: North Holland Publishing, 1978. ISBN 9780444851055.

BREIMAN, L. et al. Package 'randomForest', August 2014. Disponivel em: <http://cran.r-project.org/web/packages/randomForest/randomForest.pdf>. Acesso em: April 2014.

BRIAND, L. Measurement and Modeling in Software Engineering. In: INTERNATIONAL CONFERENCE SOFTWARE ENGINEERING, 2005. **proceedings**. [S.l.]: [s.n.].

BRIAND, L. C.; MELO, W. L.; WÜST, J. Assessing the Applicability of Fault-Proneness Models Across Object-Oriented Software Projects. **IEEE Transactions on Software Engineering**, Jul 2002. 706-720.

CARRO, L.; WAGNER, F. Sistemas computacionais embarcados. In: _____ **Jornadas de Atualização em Informática**. Campinas: SBC, 2003.

CHATZIGEORGIOU, A.; STEPHANIDES, G. Evaluating Performance and Power of Object-Oriented Vs. Procedural Programming in Embedded Processors. In: 7TH ADA-EUROPE INTERNATIONAL CONFERENCE ON RELIABLE SOFTWARE TECHNOLOGIES, 2002. **proceedings**. London: [s.n.].

CHHABRA, J. K.; GUPTA, V. A Survey of Dynamic Software Metrics. **Journal of Computer Science and Technology**, v. 25, n. 5, p. 1016-1029, 2010.

CHIDAMBER, S. R.; KEMERER, C. F. A Metrics Suite for Object Oriented Design. **IEEE Transactions on Software Engineering**, v. 20, n. 6, p. 476-493, 1994.

CHOWDHURY, I.; ZULKERNINE, M. Can Complexity, Coupling, and Cohesion Metrics Be Used As Early Indicators of Vulnerabilities? In: PROCEEDINGS OF THE 2010 ACM SYMPOSIUM ON APPLIED COMPUTING, 2010. **proceedings**. Switzerland: [s.n.].

COOK, M. L. Software metrics: an introduction and annotated bibliography. **SIGSOFT Softw. Eng. Notes**, New York, Abril 1982. 41-60. Disponivel em: <http://doi.acm.org/10.1145/1005937.1005946>.

CORRÊA, U. B. et al. Towards estimating physical properties of embedded systems using software quality metrics. In: IEEE 10TH INT. CONF. COMPUT. AND INFORM. TECHNOLOGY (CIT), 2010. **proceedings**. Bradford: [s.n.]. p. 2381-2386.

CURTIS, B. The Business Value of Application Internal Quality, Abril 2009. Disponivel em: <http://www.davidconsultinggroup.com/blogs/harris/wp-content/uploads/2009/04/the-business-value-of-application-internal-quality.pdf>. Acesso em: 14 Agosto 2014.

DANCEY, C.; REIDY, J. **Statistics without Maths for Psychology**. 5th. ed. [S.l.]: Prentice Hall, 2011.

DE JONG, G. A UML-based design methodology for real-time and embedded systems. In: DESIGN, AUTOMATION AND TEST IN EUROPE (DATE), 2002. **proceedings**. Paris: IEEE. p. 776-779.

DUFOUR, B. et al. **Dynamic Metrics for Compiler Developers**. McGill University. Montreal, p. 19. 2002.

DUFOUR, B. et al. Dynamic metrics for java. In: ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, 2003. **proceedings**. Anaheim: ACM. p. 149-168.

EBERT, C.; JONES, C. Embedded Software: Facts, Figures, and Future. **IEEE Computer**, v. 42, n. 4, p. 42-52, April 2009. ISSN 0018-9162.

ENGELBRECHT, A. P. **Computational Intelligence An Introduction**. [S.l.]: Wiley Publishing, 2007.

FAYYAD, U.; PIATETSKY-SHAPIRO, G.; SMYTH, P. Databases, From Data Mining to Knowledge Discovery in Databases. **AI Magazine**, v. 17, p. 37-54, 1996.

FENTON, N. E.; PFLEEGER, S. L. **Software Metrics:** A Rigorous and Practical Approach. 2nd. ed. Boston: PWS Publishing Co., 1998. ISBN 0534954251.

FILZMOSER, P. **Linear and Nonlinear Methods for Regression and Classification and applications in R**. Vienna University of Technology. Vienna, p. 52. 2008.

FOWLER, M. **Refactoring:** Improving the Design of Existing Code. [S.l.]: Addison-Wesley Professional, 1999.

FRIEDMAN, J. Multivariate Adaptive Regression Splines. **The Annals of Statistics**, v. 19, n. 1, p. 1-67, 1991. ISSN doi:10.1214/aos/1176347963.

GRAYBILL, F. A.; IYE, H. K. **Regression Analysis:** Concepts and Applications. 1st. ed. [S.l.]: Duxbury Pr, 1994.

GRIES, M. Methods for evaluating and covering the design space during early design development. **Integr. VLSI J.**, v. 38, n. 2, p. 131-183, Dec 2004. ISSN 10.1016/j.vlsi.2004.06.001.

GUIMARÃES, P. **Métodos Quantitativos Estatísticos**. Curitiba: IESDE Brasil, 2008.

HARRELL, F. E. **Regression Modeling Strategies**. NY: Springer, 2002.

HASTIE, T.; TIBSHIRANI, R.; FRIEDMAN, J. **The Elements of Statistical Learning**. 1nd. ed. [S.l.]: Springer, 2009.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture, Fifth Edition:** A Quantitative Approach. 5th. ed. [S.l.]: Morgan Kaufmann, 2011.

HO, T. K. Random decision forests. In: INTERNATIONAL CONFERENCE ON DOCUMENT ANALYSIS AND RECOGNITION, 1995. **proceedings**. [S.l.]: IEEE. p. 278-282.

HOFFMANN, R.; VIEIRA, S. **Análise de regressão: uma introdução à econometria**. São Paulo. 1977. (8527100231).

HUMPHREY, W. S. **Managing the software process**. Boston: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN 0-201-18095-2.

JAVA Code Coverage for Eclipse, 2006. Disponivel em: <http://www.eclemma.org/>.

JIANG, Y. et al. Comparing Design and Code Metrics for Software Quality Prediction. In: INTERNATIONAL WORKSHOP ON PREDICTOR MODELS IN SOFTWARE ENGINEERING, 2008. **proceedings**. New York: ACM. p. 11-18.

JONES, C. **Programming Productivity**. New York: Mcgraw-Hill College, 1986.

KANMANI, S. et al. Object Oriented Software Quality Prediction Using General Regression Neural Networks. **SIGSOFT Softw. Eng. Notes**, 2004. 1-6.

KNORRECK, D.; APVRILLE, L.; PACALET, R. Formal system-level design space exploration. In: INTERNATIONAL CONFERENCE ON NEW TECHNOLOGIES OF DISTRIBUTED SYSTEMS , 2010. **proceedings**. [S.l.]: IEEE. p. 1-8.

KOHAVI, R. A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection. In: INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, 1995. **proceedings**. San Francisco: Morgan Kaufmann Publishers Inc. p. 1137-1143.

LANZA, M.; MARINESCU, R. **Object-Oriented Metrics in Practice:** Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. [S.l.]: Springer, 2006.

LI, S. et al. McPAT An integrated power, area, and timing modeling framework for multicore and manycore architectures. In: ANNUAL IEEE/ACM INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 2009. **proceedings**. New York: IEEE. p. 469-480.

LORENZ, M.; KIDD, J. **Object-Oriented Software Metrics**. [S.l.]: Prentice Hall, 1994.

MARIANI, G. et al. A correlation-based design space exploration methodology for multi-processor systems-on-chip. In: 47TH DESIGN AUTOMATION CONFERENCE, 2010. **proceedings**. California: [s.n.].

MARTIN, G. UML for embedded systems specification and design motivation and overview. In: DESIGN, AUTOMATION AND TEST IN EUROPE (DATE), 2002. **proceedings**. [S.l.]: IEEE. p. 773-775.

MARTIN, G.; LAVAGNO, L.; LOUIS-GUERIN, J. Embedded UML a merger of real-time UML and co-design. In: INTERNATIONAL SYMPOSIUM ON HARDWARE/SOFTWARE CODESIGN, 2001. **proceedings**. Copenhagen: IEEE. p. 23-28.

MARTIN, R. C. **Agile Software Development, Principles, Patterns, and Practices**. 1st. ed. [S.l.]: Prentice Hall, 2002.

MARWEDEL, P. **Embedded System Design**. Dordrecht: Springer, 2006.

MATTOS, J. C. B. et al. Making Object Oriented Efficient for Embedded System Applications. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 2005. **proceedings**. Florianopolis: [s.n.].

MCCABE, J. A Complexity Measure. **IEEE Transactions on Software Engineering**, 4, Dezembro 1976. 308-320.

MCCABE, T. J.; BUTLER, C. W. Design Complexity Measurement and Testing. **Commun. ACM**, New York, v. 32, n. 12, p. 1415-1425, Dec 1989.

MCCULLAGH, P.; NELDER, J. A. **Generalized Linear Models**. 2nd. ed. London: Chapman and Hall, 1989.

MENZIES, J. et al. Assessing Predictors of Software Defects. In: WORKSHOP PREDICTIVE SOFTWARE MODELS, 2004. **proceedings**. Chicago: [s.n.].

MENZIES, T.; GREENWALD, J.; FRANK, A. Data Mining Static Code Attributes to Learn Defect Predictors. **Software Engineering, IEEE Transactions on**, 2007. 2-13.

MEYER, D. et al. Package ‘e1071’, March 2014. Disponivel em: <http://cran.r-project.org/web/packages/e1071/e1071.pdf>. Acesso em: April 2014.

MIHANCEA, P. F.; MARINESCU, C. . Changes, Defects and Polymorphism Is There Any Correlation? In: SOFTWARE MAINTENANCE AND REENGINEERING (CSMR), 2013 17TH EUROPEAN CONFERENCE ON , 2013. **proceedings**. Genova: [s.n.].

MILBORROW, S. Package ‘earth’, January 2014. Disponivel em: <http://cran.r-project.org/web/packages/earth/earth.pdf>. Acesso em: April 2014.

MILLS, E. E. **Software Metrics**. Software Engineering Institute. Seattle, p. 39. 1988. (CMU/SEI-88-CM-012}).

NAGAPPAN, N.; BALL, T.; MURPHY, B. Using Historical In-Process and Product Metrics for Early Estimation of Software Failures. In: INTERNATIONAL SYMPOSIUM ON SOFTWARE RELIABILITY ENGINEERING, 2006. **proceedings**. Washington: [s.n.].

NAKAMURA, M. A. H. T. A Software Quality Evaluation Method Using the Change of Source Code Metrics. In: SOFTWARE RELIABILITY ENGINEERING WORKSHOPS (ISSREW), 2012 IEEE 23RD INTERNATIONAL SYMPOSIUM ON, 2012. **proceedings**. Dallas: [s.n.].

NASCIMENTO, F. A.; OLIVEIRA, M. F.; WAGNER, F. R. Model-driven Engineering Framework for Embedded Systems Design. **Innov. Syst. Softw. Eng.**, Mar 2012. 19-33.

OHLSSON, N.; ALBERG, H. Predicting fault-prone software modules in telephone switches. **IEEE Transactions on Software Engineering**, 1996. 886-894.

OLIVEIRA, M. F. D. S. et al. Exploiting the Model-Driven Engineering Approach to Improve Design Space Exploration of Embedded Systems. In: ANNUAL SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEM DESIGN, 2009. **proceedings**. Natal: [s.n.].

OLIVEIRA, M. F. S. et al. Design space abstraction and metamodeling for embedded systems design space exploration. In: 7TH INTERNATIONAL WORKSHOP ON MODEL-BASED METHODOLOGIES FOR PERVASIVE AND EMBEDDED SOFTWARE, 2010. **proceedings**. Belgium: [s.n.].

OMG. **Unified Modeling Language™ (UML®) Resource Page**, 2014. Disponivel em: <http://www.uml.org/>. Acesso em: 2015.

PORTER, A. A.; SELBY, R. W. Empirically guided software development using metric-based classification trees. **IEEE Software**, v. 7, n. 2, p. 46-54, 1990.

PORTER, A. A.; SELBY, R. W. Evaluating Techniques for Generating Metric-Based Classification Trees. **Journal of Systems and Software**, v. 12, p. 209-218, 1990.

PRESMAN, R. S. E. D. S. **Engenharia de Software**. 6ª Edição. ed. São Paulo: McGraw-Hill, 2006.

PRESSMAN, R. S. **Engenharia de Software:** uma abordagem profissional. 7th. ed. Porto Alege: AMGH, 2011.

R, D. C. T. R: A Language and Environment for Statistical Computing, 2014. ISSN 3-900051-07-0. Disponivel em: <http://www.r-project.org/>. Acesso em: 2014.

REDIN, R. M. et al. On the Use of Software Quality Metrics to Improve Physical Properties of Embedded Systems. **Distributed Embedded Systems: Design, Middleware and Resources**, v. 271, n. IFIP – The International Federation for Information Processing, p. 101-110, 2008.

RICCOBENE, E. et al. A SoC design methodology involving a UML 2.0 profile for SystemC. In: DESIGN, AUTOMATION AND TEST IN EUROPE (DATE), 2005. **proceedings**. [S.l.]: IEEE. p. 704-709.

RIPLEY, B.; VENABLES, W. Package 'nnet', March 2014. Disponivel em: <http://cran.r-project.org/web/packages/nnet/nnet.pdf>. Acesso em: April 2014.

ROKACH, L.; MAIMON, O. **Data mining with decision trees:** theory and applications. [S.l.]: World Scientific Pub Co Inc, 2008.

SAMI, A.; FAKHRAHMAD, S. M. Design-level metrics estimation based on code metrics. In: ACM SYMPOSIUM ON APPLIED COMPUTING, 2010. **proceedings**. New York: ACM. p. 2531-2535.

SHULL, F. et al. What We Have Learned About Fighting Defects. In: INTERNATIONAL SYMPOSIUM ON SOFTWARE METRICS, 2002. **proceedings**. Washington: [s.n.].

SMITH, P. G. Accelerated Product Development: Techniques and Traps. In: _____ **The PDMA Handbook of New Product Development**. Hoboken: Weley, 2004.

SOMMERVILLE, I. **Engenharia de Software**. São Paulo: Pearson Addison Wesley, 2007.

SOMMERVILLE, I. **Software engineering**. 9th. ed. Harlow: Addison-Wesley, 2010.

SRIKANT, R.; AGRAWAL, R. Mining Sequential Patterns Generalizations and Performance Improvements. In: 5TH INTERNATIONAL CONFERENCE ON EXTENDING DATABASE TECHNOLOGY: ADVANCES IN DATABASE TECHNOLOGY, 1996. **proceedings**. London: Springer-Verlag. p. 3-17.

TAN, P.-N.; STEINBACH, M.; KUMAR, V. **Introduction to Data Mining**. Boston: Addison-Wesley Longman Publishing Co., Inc., 2005.

THE R Stats Package, 2014. Disponivel em: <http://stat.ethz.ch/R-manual/R-patched/library/stats/html/00Index.html>. Acesso em: 2014.

THERNEAU, T.; ATKINSON, B.; RIPLEY, B. Package 'rpart', March 2014. Disponivel em: <http://cran.r-project.org/web/packages/rpart/rpart.pdf>. Acesso em: April 2014.

TOOLWORKS, S. Understand. **Understand Your Code**, Sept. 2014. Disponivel em: <http://www.scitools.com>. Acesso em: Jul 2014.

TURHAN, B. et al. On the Relative Value of Cross-company and Within-company Data for Defect Prediction. **Empirical Software Engineering**, v. 14, n. 5, p. 540--578, 2009.

UBM TECH ELECTRONICS. **Embedded Market Study**. [S.l.]. 2013.

VIDAL, J. et al. A co-design approach for embedded system modeling and code generation with UML and MARTE. In: DESIGN, AUTOMATION TEST IN EUROPE (DATE), 2009. **proceedings**. Nice: IEEE. p. 226-231.

VIEIRA, A. et al. Early estimation of NFRs for Embedded System Using Design Metrics. In: BRAZILIAN SYMPOSIUM ON COMPUTING SYSTEM ENGINEERING, 2014. **proceedings**. Manaus: [s.n.].

VIEIRA, A. et al. NFRs Early Estimation Through Software Metrics. In: DESIGN AUTOMATION AND TEST IN EUROPE CONFERENCE AND EXHIBITION (DATE), 2015. **proceedings**. Grenoble: [s.n.].

VIEIRA, A.; FAUSTINI, P.; COTA, É. Using software metrics to estimate the impact of maintenance in the performance of embedded software. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION, 2014. **proceedings**. Victoria: [s.n.]. p. 521-525.

WANG, G. Modeling C-based embedded system using UML design. In: INTERNATIONAL CONFERENCE ONMECHATRONICS AND AUTOMATION, 2009. **proceedings**. Changchun: IEEE. p. 2973 - 2977.

WEHRMEISTER, M. A.; PACKER, J. G.; CERON, L. M. Support for Early Verification of Embedded Real-time Systems Through UML Models Simulation. **SIGOPS Oper. Syst. Rev.**, v. 46, n. 1, p. 73-81, 2012.

WOLF, W. **Computer as Components:** Principles of Embedded Computer Systems Design. 3rd. ed. [S.l.]: Morgan Kaufmann Publishers, 2012.

WÜST, J. SDMetrics. **SDMetrics**, Zellertal, Sept. 2014. Disponivel em: <http://sdmetrics.com/>. Acesso em: April 2014.

YACOUB, S. M.; AMMAR, H. H.; ROBINSON, T. Dynamic Metrics for Object Oriented Designs. In: INTERNATIONAL SYMPOSIUM ON SOFTWARE METRICS, 1999. **proceedings**. Washington: ACM.

YADAV, V.; SINGH, R. Predicting Design Quality of Object-Oriented Software using UML diagrams. In: ADVANCE COMPUTING CONFERENCE (IACC), 2013 IEEE 3RD INTERNATIONAL, 2013. **proceedings**. Ghaziabad: [s.n.].

ANEXO A – MÉTRICAS DE DIAGRAMAS DE CLASSE EXTRAÍDAS PELO SDMETRICS

Tabela A 1 – Métricas de Classe

API Name	Category	Description
NumAttr	Size	The number of attributes in the class.
NumOps	Size	The number of operations in a class.
NumPubOps	Size	The number of public operations in a class.
Setters	Size	The number of operations with a name starting with 'set'.
Getters	Size	The number of operations with a name starting with 'get', 'is', or 'has'.
NOC	Inheritance	The number of children of the class (UML Generalization).
NumDesc	Inheritance	The number of descendents of the class (UML Generalization).
NumAnc	Inheritance	The number of ancestors of the class.
DIT	Inheritance	The depth of the class in the inheritance hierarchy.
CLD	Inheritance	Class to leaf depth.
OpsInh	Inheritance	The number of inherited operations.
AttrInh	Inheritance	The number of inherited attributes.
Dep_Out	Coupling	The number of elements on which this class depends.
Dep_In	Coupling	The number of elements that depend on this class.
NumAssEl_ssc	Coupling	The number of associated elements in the same scope (namespace) as the class.
NumAssEl_sb	Coupling	The number of associated elements in the same scope branch as the class.
NumAssEl_nsb	Coupling	The number of associated elements not in the same scope branch as the class.
EC_Par	Coupling	The number of times the class is externally used as parameter type.
IC_Par	Coupling	The number of parameters in the class having another class or interface as their type.

Fonte: (WÜST, 2014)

ANEXO B – MÉTRICAS DE CÓDIGO FONTE EXTRAÍDAS PELO UNDERSTAND

Tabela B 1 – Métricas de Complexidade

API Name	Description
AvgCyclomatic	Average cyclomatic complexity for all nested functions or methods.
AvgCyclomaticModified	Average modified cyclomatic complexity for all nested functions or methods.
AvgCyclomaticStrict	Average strict cyclomatic complexity for all nested functions or methods.
AvgEssential	Average Essential complexity for all nested functions or methods.
MaxCyclomatic	Maximum cyclomatic complexity of all nested functions or methods.
MaxCyclomaticModified	Maximum modified cyclomatic complexity of nested functions or methods.
MaxCyclomaticStrict	Maximum strict cyclomatic complexity of nested functions or methods.
MaxInheritanceTree	Maximum depth of class in inheritance tree. [aka DIT]
MaxNesting	Maximum nesting level of control constructs.
SumCyclomatic	Sum of cyclomatic complexity of all nested functions or methods. [aka WMC]
SumCyclomaticModified	Sum of modified cyclomatic complexity of all nested functions or methods.
SumCyclomaticStrict	Sum of strict cyclomatic complexity of all nested functions or methods.
SumEssential	Sum of essential complexity of all nested functions or methods.

Fonte: (TOOLWORKS, 2014)

Tabela B 2 – Métricas de Volume

API Name	Description
AvgLine	Average number of lines for all nested functions or methods.
AvgLineBlank	Average number of blank for all nested functions or methods.
AvgLineCode	Average number of lines containing source code for all nested functions or methods.
AvgLineComment	Average number of lines containing comment for all nested functions or methods.
CountLine	Number of all lines. [aka NL]
CountLineBlank	Number of blank lines. [aka BLOC]
CountLineCode	Number of lines containing source code. [aka LOC]
CountLineCodeDecl	Number of lines containing declarative source code.
CountLineCodeExe	Number of lines containing executable source code.
CountLineComment	Number of lines containing comment. [aka CLOC]
CountSemicolon	Number of semicolons.
CountStmt	Number of statements.
CountStmtDecl	Number of declarative statements.
CountStmtExe	Number of executable statements.

Fonte: (TOOLWORKS, 2014)

Tabela B 3 – Métricas de Orientação a Objetos

API Name	Description
CountClassCoupled	Number of other classes coupled to. [aka CBO (coupling between object classes)]
CountClassDerived	Number of immediate subclasses. [aka NOC (number of children)]
CountDeclClassMethod	Number of class methods.
CountDeclClassVariable	Number of class variables.
CountDeclInstanceMethod	Number of instance methods. [aka NIM]
CountDeclInstanceVariable	Number of instance variables. [aka NIV]
CountDeclMethod	Number of local methods.
CountDeclMethodAll	Number of methods, including inherited ones. [aka RFC (response for class)]
CountDeclMethodPrivate	Number of local private methods. [aka NPM]
CountDeclMethodProtected	Number of local protected methods.
CountDeclMethodPublic	Number of local public methods. [aka NPRM]
PercentLackOfCohesion	100% minus the average cohesion for package entities. [aka LCOM, LOCM]

Fonte: (TOOLWORKS, 2014)