Instituto Politécnico de Viseu
Escola Superior de Tecnologia e Gestão de Viseu

Nelson Duarte Marques

Modelo de Custo Híbrido para o Desenvolvimento de Software em Ambientes Ágeis

Outubro de 2017

Instituto Politécnico de Viseu

Escola Superior de Tecnologia e Gestão de Viseu

Nelson Duarte Marques

Modelo de Custo Híbrido para o Desenvolvimento de Software em Ambientes Ágeis

Dissertação de Mestrado
Sistemas e Tecnologias de Informação para as Organizações

Professor Doutor José Francisco Monteiro Morgado

Outubro de 2017

Aos meus pais, por tudo, desde sempre.

RESUMO

A estimativa de esforço é uma das principais tarefas no planeamento e gestão de qualquer projeto de desenvolvimento de *software*. Desde a sua proposta inicial até ao seu desenvolvimento e manutenção é crucial ter uma previsão precisa do esforço necessário em cada etapa. Estimativas muito elevadas irão certamente levar à perda de competitividade no mercado, por outro lado estimativas muito baixas poderão levar à falha de compromissos, datas de entrega e consequentemente à perda de dinheiro.

Embora exista uma grande quantidade de técnicas e modelos de estimativa de esforço de *software*, a grande maioria foca-se no desenvolvimento de *software* tradicional. O surgimento de novas metodologias de trabalho, como as metodologias ágeis, levou a que a aplicabilidade dos modelos existentes seja reduzida, pois estas novas metodologias baseiam-se num conceito totalmente diferente do desenvolvimento de *software* tradicional. Apesar de nos últimos anos a utilização de metodologias ágeis, nas mais diversas áreas, ter crescido, continuam a ser escassos os métodos de estimativa criados especificamente para este tipo de ambientes.

Nesta dissertação foi proposto um modelo de custo híbrido para o desenvolvimento de *software* em ambientes ágeis. O modelo proposto combina vertentes tanto de modelos de estimativas ágeis, bem como de modelos de estimativas de esforço tradicionais com técnicas de *Machine Learning* de modo a aumentar a precisão das estimativas produzidas. Os resultados dos modelos referentes às várias técnicas de *Machine Learning* utilizadas no modelo proposto foram comparados entre si e também com outros modelos existentes na literatura.

ABSTRACT

Effort estimation is one of the major tasks in planning and managing any software development project. From the initial proposal to its development and maintenance it is crucial to have an accurate forecast of the effort required at each stage. Very high estimates will certainly lead to loss of competitiveness in the market, on the other hand very low estimates may lead to failure to meet commitments, delivery dates and consequently loss of money.

Although there are a lot of techniques and models for software effort estimation, the vast majority focuses on traditional software development. The emergence of new methodologies, such as agile methodologies, has led to the reduced applicability of existing models, as these new methodologies are based on a totally different concept from the development of traditional software. Although in recent years the use of agile methodologies in the most diverse areas has grown, estimation methods created specifically for agile environments are still scarce.

In this dissertation a hybrid cost model was proposed for software development in agile environments. The proposed model combines aspects of both agile estimation models as well as traditional effort estimation models with Machine Learning techniques in order to increase the precision of the estimates produced. The results of the models related to the various Machine Learning techniques used in the proposed model were compared with each other and also with other models present in the literature.

PALAVRAS-CHAVE

Modelo de Custo

Estimativa de Esforço

Desenvolvimento de Software

Metodologias Ágeis

Planning Poker

Story Points

Machine Learning

Scikit Learn

KEY WORDS

Cost Model

Effort Estimation

Software Development

Agile Methodologies

Planning Poker

Story Points

Machine Learning

Scikit Learn

AGRADECIMENTOS

Gostaria de agradecer em primeiro lugar a toda a minha família. Em especial aos meus pais, Manuel Marques e Aida Marques, que sempre me apoiaram e incentivaram, não só durante a execução desta dissertação, mas ao longo de todo o mestrado e todo o meu percurso académico.

Ao meu orientador, o Professor Doutor José Francisco Monteiro Morgado, da Escola Superior de Tecnologia e Gestão de Viseu que sempre se mostrou disponível para me apoiar durante a realização desta dissertação, um muito obrigado.

Agradeço a todos os meus colegas de trabalho e amigos o apoio constante, em especial ao Guilherme Laranjeira, João Sousa, Júlio Florentino, Roberto Rocha, Tiago Gomes e Tiago Rebelo.

Aos meus colegas do mestrado, em especial ao João Sousa e ao Tiago Gomes, pelo apoio e incentivo e também por todos os momentos passados em conjunto, um muito obrigado.

Um grande obrigado a todos, nada disto teria sido possível sem o contributo de todos vós.

ÍNDICE GERAL

Índice de Figuras xvii

Índice de Tabelas xix

Abreviaturas e Siglas xxi

1. Introdução 1

1.1 Motivação e Objetivos 3

1.2 Metodologia de Investigação 3

1.3 Organização do Documento 4

2. Estimativa de Esforço de Software 7

2.1 Modelos Algorítmicos 8

2.1.1 Source Lines of Code 9

2.1.2 Function Point Analysis 11

2.1.3 Putnam Model 15

2.1.4 Constructive Cost Model 20

2.2 Modelos Não Algorítmicos 31

2.2.1 Expert Judgment 32

2.2.2 Analogy 32

2.2.3 Price-to-win 33

2.2.4 Bottom-up e Top-down 34

2.2.5 Wideband Delphi 35

2.3 Sumário 36

3. Estimativa de Esforço de Software em Ambientes Ágeis 39

3.1 Story Points 40

3.1.1 Story Points vs Tempo 41

3.2 Ideal Days 44

3.3 Planning Poker 44

3.4 Sumário 46

4. Machine Learning 47

4.1 Aplicações na Estimativa de Esforço de Software 48

4.2 Técnicas 52

4.2.1 Linear Regression 54
4.2.2 Decision Trees 54
4.2.3 K-Nearest Neighbors 56
4.2.4 Support Vector Regression 58
4.2.5 Multi Layer Perceptron 60
4.2.6 Ensembles 61
4.2.6.1 Bagging 63
4.2.6.2 Random Forest 63
4.2.6.3 Extremely Randomized Trees 63
4.2.6.4 Gradient Boosting 64
4.3 Sumário 64
5. Critérios de Avaliação 67
5.1 Mean Absolute Error 68
5.2 Magnitude of Relative Error 68
5.3 Mean Magnitude of Relative Error 68
5.4 Mean of Magnitude of Error Relative to the Estimate 69
5.5 Percentage Relative Error Deviation 69
5.6 Sumário 69
6. Modelo Proposto 71
6.1 Dataset 72
6.2 Detalhes Experimentais 79
6.2.1 Scikit Learn 79
6.2.2 Pré-Processamento de Dados 80
6.2.3 Hyperparameters 83
6.2.4 Validação dos modelos 84
6.3 Metodologia 88
6.4 Resultados 90
6.5 Análise Comparativa 93
6.6 Sumário 97
7. Conclusão 101
7.1 Limitações 102
7.2 Trabalhos Futuros 103

Referências 105

Anexos 115

Anexo A – Código Fonte do Modelo Proposto 117

Anexo B – Resultados completos do modelo proposto (datasets de dimensão reduzida).. 127

ÍNDICE DE FIGURAS

Figura 2-1: Distribuição de Rayleigh ... 15
Figura 2-2: Distribuição de Rayleigh aplicada às fases de desenvolvimento de um projeto ... 16
Figura 2-3: Distribuição de Rayleigh aplicada ao desenvolvimento de software ... 16
Figura 2-4: Esforço acumulado gasto no desenvolvimento do software ... 17
Figura 2-5: Distribuição de esforço gasto no desenvolvimento do software ... 18
Figura 2-6: Exemplo de utilização da técnica Wideband Delphi ... 35
Figura 3-1: Distribuição do tempo correspondente a um story point ... 43
Figura 3-2: Distribuição do tempo correspondente a um, dois e três story points ... 43
Figura 4-1: Relação entre Engenharia de Software e Machine Learning ... 48
Figura 4-2: Distribuição das técnicas de ML nos estudos selecionados por Wen et al. ... 49
Figura 4-3: Classificação vs Regressão ... 53
Figura 4-4: Subdivisão de um conjunto de dados utilizando uma árvore de decisão ... 55
Figura 4-5: Previsão de uma nova instância utilizando K-Nearest Neighbors ... 57
Figura 4-6: Maximização da margem do plano que separa as classes utilizando SVM ... 58
Figura 4-7: Mapeamento de atributos para um espaço vetorial com mais dimensões ... 59
Figura 4-8: Regressão utilizando Support Vector Regression ... 59
Figura 4-9: Representação de um Multi Layer Perceptron ... 61
Figura 4-10: Exemplo dos modelos base de um modelo de Gradient Boosting ... 64
Figura 6-1: Aplicação de cross-validation na prototipagem de um modelo ... 85
Figura 6-2: Holdout cross-validation ... 86
Figura 6-3: k-fold cross-validation ... 87
Figura 6-4: Nested k-fold cross-validation ... 88
Figura 6-5: Etapas da metodologia utilizada ... 89
Figura 6-6: Resultados do modelo proposto – MAE ... 91
Figura 6-7: Resultados do modelo proposto – MMRE ... 92
Figura 6-8: Resultados do modelo proposto – PRED(25) ... 93
Figura 6-9: Resultados do modelo proposto (datasets de dimensão reduzida) – MAE ... 95
Figura 6-10: Resultados do modelo proposto (datasets de dimensão reduzida) – MMRE ... 96
Figura 6-11: Resultados do modelo proposto (datasets de dimensão reduzida) – PRED(25) .. 97

ÍNDICE DE TABELAS

Tabela 2-1: Tipos de função e níveis de complexidade 11
Tabela 2-2: Características gerais do sistema e graus de influência 12
Tabela 2-3: Modos de desenvolvimento 21
Tabela 2-4: Coeficientes Basic COCOMO 22
Tabela 2-5: Multiplicadores de esforço Intermediate COCOMO 24
Tabela 2-6: Coeficientes Intermediate COCOMO 25
Tabela 2-7: Multiplicadores de esforço Analyst capability Detailed COCOMO 25
Tabela 2-8: Multiplicadores de esforço scale factors COCOMO II 27
Tabela 2-9: Fatores de esforço Early Design e Post-Architecture 28
Tabela 2-10: Multiplicadores de esforço COCOMO II Post-Architecture 29
Tabela 2-11: Multiplicadores de esforço COCOMO II Early Design 30
Tabela 4-1: Datasets utilizados nos estudos selecionados por Wen et al. 50
Tabela 4-2: Resumo dos resultados dos estudos selecionados por Wen et al. 51
Tabela 4-3: Resumo dos resultados dos estudos baseados em Story Points 52
Tabela 6-1: Distribuição de user stories e iterações por projeto 73
Tabela 6-2: Distribuição de user stories por story points 73
Tabela 6-3: Distribuição de user stories por custo real 76
Tabela 6-4: Níveis de Experiencia 77
Tabela 6-5: Níveis de impacto 78
Tabela 6-6: Exemplo de uma user story do dataset criado 79
Tabela 6-7: Resultados do modelo proposto 90
Tabela 6-8: Resultados do modelo proposto (datasets de dimensão reduzida) 94

ABREVIATURAS E SIGLAS

AFP	Adjusted Function Points
ANN	Artificial Neural Network
AR	Association Rules
BN	Bayesian Networks
CBR	Case-Based Reasoning
COCOMO	Constructive Cost Model
DT	Decision Tree
EAF	Effort Adjustment Factors
FP	Function Points
FPA	Function Point Analysis
GA	Genetic Algorithms
GP	Genetic Programming
GSC	General System Characteristics
IFPUG	International Function Point Users Group
ISPA	International Society of Parametric Analysis
KDSI	Thousand Delivered Source Instructions
KNN	K-Nearest Neighbors
LR	Linear Regression
MAE	Mean Absolute Error
ML	Machine Learning
MLP	Multi Layer perceptron
MMER	Mean of Magnitude of Error Relative to the Estimate
MMRE	Mean Magnitude of Relative Error
MRE	Magnitude of Relative Error
PRED	Percentage Relative Error Deviation
RF	Random Forest
SDEE	Software Development Effort Estimation
SF	Scale Factors
SLOC	Source Lines of Code

SP	Story Points
SVM	Support Vector Machines
SVR	Support Vector Regression
SVR	Support Vector Regression
TCF	Technical Complexity Factor
UFP	Unadjusted Function Points
US	User Story
VAF	Value Adjustment Factor

1. Introdução

Historicamente, o desenvolvimento de *software* é percecionado como sendo notoriamente ineficiente. Em 2013, o Chaos Manifesto do Standish Group (Manifesto, 2013) constatou que 43% dos projetos de TI foram entregues com atraso, com menos funcionalidades e/ou ultrapassaram o orçamento. Uma causa de ineficiência é a inadequação da informação económica disponível para apoiar a tomada de decisões. Na ausência de estimativas de custos razoáveis, os projetos estão em risco. Melhorar a capacidade das organizações de modelar e analisar aspetos económicos pode ajudar a torná-las significativamente mais produtivas (B. Boehm e Sullivan, 1999). Uma análise mais recente realizada pela Sociedade Internacional de Análise Paramétrica (*International Society of Parametric Analysis* - ISPA) identificou as principais razões responsáveis pela falha da maioria dos *softwares* (ISPA, 2008). Estas razões podem ser resumidas da seguinte forma:

- Falta de compreensão dos requisitos;
- Estimativa do tamanho do *software* incorreta;
- Falha na avaliação do nível de especialização dos recursos humanos.

A estimativa de esforço de desenvolvimento de *software* (*Software Development Effort Estimation* – SDEE) é o processo de previsão do esforço necessário para desenvolver um sistema de *software*. De um modo geral, a SDEE pode ser considerada como um subdomínio da estimativa do esforço de *software*, que inclui as previsões não só do esforço de desenvolvimento de *software*, mas também do esforço de manutenção de *software*. A terminologia estimativa de custo de desenvolvimento de *software* é frequentemente utilizada indistintamente com estimativa de esforço de desenvolvimento de *software*, embora o esforço apenas forme a maior parte do custo de *software*. Estimativa do esforço de desenvolvimento com precisão na fase inicial do ciclo de vida do *software* desempenha um papel crucial na gestão eficaz de projetos (Wen et al., 2012). A precisão da estimativa de esforço tem implicações no

resultado de um projeto de *software*. Estimativas muito baixas podem levar a falhas nos prazos de entrega e superações orçamentárias, enquanto estimativas demasiado altas podem ter um impacto negativo na competitividade das organizações (Choetkiertikul et al., 2016).

Desde o início da era dos computadores na década de 40 que a estimativa de custo de desenvolvimento de *software* tem sido uma tarefa importante, mas difícil. Como as aplicações de *software* têm aumentado em tamanho e importância, também a necessidade de precisão na estimativa de custo de *software* tem aumentado. Desde o início dos anos 50 que os profissionais ligados ao desenvolvimento de *software* e os investigadores da área têm tentado desenvolver métodos para estimar custos de *software*. Os modelos de estimativa de custos de *software* têm aparecido na literatura ao longo das últimas décadas, no entanto a área de investigação de estimativa de custos de *software* ainda está na sua infância (Z. Zia, Rashid e uz Zaman, 2011).

Nos últimos anos, o *software* tornou-se o componente mais caro de projetos de sistemas de informação. A maior parte do custo do desenvolvimento de *software* é devido ao esforço humano, consequentemente a maioria dos métodos de estimativa de custo concentra-se neste aspeto e estima as diferentes tarefas em meses ou horas de trabalho. Estimativas precisas de custo de *software* são fundamentais tanto para os programadores como para os clientes (Z. Zia, Rashid e uz Zaman, 2011).

Ao longo dos anos, foram introduzidas diferentes técnicas de estimativa do esforço de *software*. Estas são utilizadas eficazmente no desenvolvimento de *software* tradicional, porém, a diversidade de novas metodologias de desenvolvimento de *software* resultou numa situação em que a aplicabilidade dos modelos de previsão de esforço existentes é limitada. O desenvolvimento de *software* utilizando metodologias ágeis é uma dessas situações. Estas metodologias ágeis baseiam-se num conceito totalmente diferente do desenvolvimento de *software* tradicional. Como consequência os métodos clássicos de estimativa de esforço não podem ser aplicados porque foram especificamente desenvolvidos para metodologias tradicionais (Z. K. Zia, Tipu e Zia, 2012).

O surgimento das metodologias ágeis resulta do reconhecimento da necessidade de uma alternativa ao desenvolvimento tradicional que segue uma estrutura rígida e é baseado em requisitos preestabelecidos. O desenvolvimento de *software* utilizando metodologias ágeis baseia-se na boa comunicação entre a equipa de desenvolvimento, na entrega rápida de partes do *software* e na facilidade de mudança em qualquer altura do ciclo de vida do *software*. Devido à crescente utilização de métodos ágeis na indústria nos últimos anos, a estimativa de esforço de desenvolvimento de *software* em ambientes ágeis é uma área de grande interesse (Satapathy, Panda e Rath, 2014).

Recentemente modelos baseados em *Machine Learning* têm recebido um aumento gradual de atenção na área de investigação de estimativas de esforço de desenvolvimento. Apesar do grande número de estudos já realizados sobre os modelos baseados em *Machine Learning*,

foram encontrados resultados inconsistentes quanto à precisão das estimativas produzidas pelos mesmos, quanto às comparações entre modelos baseados em *Machine Learning* e modelos mais tradicionais e quanto às comparações entre diferentes modelos baseados em *Machine Learning*. Por exemplo, constatou-se que a precisão da estimativa produzida pelos modelos baseados em *Machine Learning* varia de acordo com os diferentes *datasets* utilizados no seu desenvolvimento e treino (Wen et al., 2012).

1.1 Motivação e Objetivos

A popularidade da utilização de metodologias ágeis no desenvolvimento de *software* tem aumentado de ano para ano. Porém, ainda existe uma falta generalizada de técnicas adequadas para a obtenção de estimativas de esforço de desenvolvimento em ambientes ágeis.

Esta dissertação visa avaliar a viabilidade da combinação de modelos de estimativas ágeis com técnicas de *Machine Learning* de modo a aumentar a precisão das estimativas produzidas. Serão também analisadas e possivelmente incorporadas algumas vertentes dos modelos de estimativas de esforço tradicionais nesta combinação.

Tendo isto em conta, o principal objetivo da presente dissertação é a proposta de um modelo de custo híbrido para o desenvolvimento de *software* em ambientes ágeis. O modelo posposto deve tomar partido da capacidade dos algoritmos de *Machine Learning* conseguirem modelar relacionamentos complexos entre as diversas variáveis em estudo sem serem explicitamente programados. Os resultados dos modelos referentes às várias técnicas de *Machine Learning* utilizadas no modelo proposto serão comparados entre si e também com outros modelos existentes na literatura.

Pretende-se também analisar o impacto da utilização de um conjunto de dados de dimensões reduzidas no poder de generalização do modelo proposto.

1.2 Metodologia de Investigação

A escolha de um desenho de investigação adequado é uma tarefa essencial para o sucesso do estudo em causa. O desenho de investigação refere-se à estrutura geral ou plano de investigação de um estudo. Depois de definido o desenho, é necessário especificar o método de estudo e de recolha de dados. Por método de investigação entende-se as técnicas e práticas utilizadas para recolher, processar e analisar os dados (Bowling, 2014).

O estudo foi realizado recorrendo a uma perspetiva de investigação quantitativa utilizando uma abordagem experimental.

A pesquisa quantitativa é uma investigação empírica sistemática de fenômenos observáveis através de técnicas estatísticas, matemáticas ou computacionais. O objetivo da pesquisa quantitativa é desenvolver ou aplicar modelos matemáticos, teorias e/ou hipóteses relativas a determinado fenômeno (Bhawna e Gobind, 2015).

A abordagem quantitativa experimental tem como base uma visão positivista do mundo. Esta defende a existência de uma realidade objetiva que pode ser expressa numericamente. Como consequência, a perspetiva quantitativa tem tendência a dar origem a estudos de natureza experimentais. Neste cenário, o investigador testa uma teoria especificando hipóteses e efetuando a recolha de dados para apoiar ou refutar as hipóteses. É utilizado um desenho experimental em que os dados são recolhidos antes e depois de um tratamento experimental. Os dados recolhidos são posteriormente analisados recorrendo a procedimentos estatísticos e testes de hipóteses (Creswell, 2014).

Foi escolhido este desenho de investigação visto que a presente dissertação visa aferir a viabilidade do modelo proposto para a estimativa de esforço de desenvolvimento de *software* em ambientes ágeis. É esperado que por meio de procedimentos matemáticos seja possível concluir se o modelo proposto é ou não vantajoso para o desenvolvimento de *software* em ambientes ágeis.

1.3 Organização do Documento

A presente dissertação encontra-se dividia em sete capítulos, incluindo a Introdução. Os restantes capítulos encontram-se estruturados da seguinte forma.

O capítulo 2, Estimativa de Esforço de Software, apresenta os métodos tradicionais de estimativa de esforço de desenvolvimento de *software*. Este capítulo encontra-se dividido em duas secções – Modelos Algorítmicos e Modelos Não Algorítmicos – que representam as duas grandes categorias de técnicas de estimativa de custos de *software*.

O capítulo 3, Estimativa de Esforço de Software em Ambientes Ágeis, apresenta as métricas e os métodos de estimativa de esforço de desenvolvimento de *software* em ambientes ágeis.

O capítulo 4, Machine Learning, apresenta as aplicações de técnicas de *Machine Learning* à estimativa de esforço de desenvolvimento de *software* existentes na literatura e ainda uma descrição de todas as técnicas utilizadas posteriormente no modelo proposto.

O capítulo 5, Critérios de Avaliação, apresenta vários critérios de avaliação utilizados na avaliação do desempenho dos modelos no que diz respeito à precisão das estimativas produzidas por estes.

O capítulo 6, Modelo Proposto, apresenta uma descrição detalhada do modelo proposto e das várias técnicas de *Machine Learning*, do *dataset* e da metodologia a utilizar na sua implementação. Por fim também são apresentados os resultados do modelo proposto bem como uma análise comparativa com outros estudos semelhantes existentes na literatura.

Por fim o capítulo 7, Conclusão, apresenta as conclusões finais, limitações e possíveis trabalhos futuros no âmbito da presente dissertação.

2. Estimativa de Esforço de Software

A estimativa de esforço de *software* é o processo através do qual é obtida uma previsão do esforço necessário para desenvolver um determinado sistema de *software* (Kumari e Pushkar, 2013a). Este processo de estimativa é uma das tarefas mais importantes, e também mais difíceis, na fase inicial do ciclo de vida de qualquer *software*. A precisão da estimativa de esforço tem implicações no resultado de um projeto de *software*. Estimativas muito baixas podem levar a falhas nos prazos de entrega e superações orçamentárias, enquanto estimativas demasiado altas podem ter um impacto negativo na competitividade das organizações (Choetkiertikul et al., 2016). Quando esta é realizada de forma incorreta ou quando os resultados obtidos são imprecisos os custos dos projetos tendem a crescer rapidamente, podendo por vezes ascender a cerca de 150 a 200% a mais do custo originalmente planeado (N. Sharma e Litoriya, 2012). A percentagem de projetos de *software* que ultrapassaram o orçamento aumentou de 56% em 2004 para 59% em 2012, enquanto a percentagem de projetos de *software* que foram entregues com atraso aumentou de 71% em 2010 para 74% em 2012 (Manifesto, 2013). Estimativas de esforço de *software* mais precisas podem trazer diversos benefícios a todas as partes envolvidas no desenvolvimento de um sistema de *software*, desde o aumento dos lucros dos programadores até ao aumento da sua satisfação dos clientes (Munialo e Muketha, 2016).

Genericamente as informações utilizadas nesta estimativa são uma medida do tamanho do *software* (ou de determinada parte do mesmo) e um conjunto de fatores de custo. No fim do processo é obtido uma medida de esforço, geralmente expressa numa unidade de tempo, como horas, dias ou meses de trabalho (Kumari e Pushkar, 2013a). Os primeiros modelos de estimativa de esforço de *software* produziam uma estimativa baseando-se apenas no tamanho do *software* utilizando o número de linhas de código ou o número de Function Points (FP) (Litoriya e Kothari, 2013). No entanto, o desejo de novas funcionalidades, de entregas rápidas e de novos tipos de *software* deu origem à necessidade de novos métodos ou metodologias de

desenvolvimento de *software*. Aspetos como a reutilização de *software*, o desenvolvimento baseado em componentes, sistemas distribuídos e desenvolvimento iterativo são características que têm vindo a ganhar cada vez mais importância na engenharia de *software*. Com a evolução da engenharia de *software* ao longo dos tempos, também os métodos de estimativa de esforço de *software* foram evoluindo com o objetivo de tornar as suas estimativas mais precisas (Munialo e Muketha, 2016). Atualmente, outros fatores de custo, como fatores do projeto, fatores do processo de desenvolvimento, fatores de recursos e fatores humanos, foram incluídos nos modelos de estimativa de esforço de *software* com o intuito de melhorar a precisão da estimativa do mesmo (Litoriya e Kothari, 2013; Munialo e Muketha, 2016).

As técnicas de estimativa de custos de *software* podem ser classificadas em duas grandes categorias: modelos algorítmicos e modelos não algorítmicos. Os modelos algorítmicos são baseados na análise estatística de dados históricos de projetos anteriores, enquanto os métodos não-algorítmicos dependem essencialmente de especialistas que possuam experiência em projetos anteriores semelhantes (Khatibi e Jawawi, 2011; Kumari e Pushkar, 2013b; Leung e Fan, 2002; Litoriya e Kothari, 2013; Munialo e Muketha, 2016).

2.1 Modelos Algorítmicos

Os modelos algorítmicos utilizam fórmulas matemáticas para calcular a estimativa do custo do *software*. Estas fórmulas matemáticas baseiam-se na análise de dados históricos obtidos de projetos anteriores e dependem da combinação diversos fatores de custo dos mesmos (tais como o número de linhas de código, o número de funções a serem executadas, a linguagem de programação, a metodologia de desenvolvimento de *software* utilizada, o nível de proficiência das pessoas envolvidas nas diversas tarefas do projeto, avaliações de risco, entre muitos outros) (Kumari e Pushkar, 2013b). A aplicação destas fórmulas a dados provenientes de projetos novos produz uma estimativa do seu custo (Munialo e Muketha, 2016).

Todas as fórmulas utilizadas pelos métodos algorítmicos podem ser representadas genericamente por (2-1).

$$E = f(x_1, x_2, \dots, x_n) \qquad (2\text{-}1)$$

Onde E representa o esforço estimado e $(x_1, x_2, \dots, x_n)$ representa os fatores de custo utilizados no cálculo da estimativa. As diferenças entre os diversos modelos algorítmicos existentes relacionam-se principalmente com a escolha do o tipo de função e dos fatores de custo a utilizar (Borade e Khalkar, 2013; Khatibi e Jawawi, 2011).

Os fatores de custo utilizados na maioria dos modelos existentes podem ser divididos da seguinte forma (Khatibi e Jawawi, 2011):

- Fatores do produto: fiabilidade necessária; complexidade do produto; tamanho da base de dados; reutilização de componentes; existência e/ou necessidade de documentação;
- Fatores de *hardware*: restrições no tempo de execução de funcionalidades; restrições de memória e armazenamento; volatilidade das plataformas;
- Fatores dos recursos humanos: capacidade do analista; capacidade do programador; experiencia com a linguagem de programação; experiencia no tipo de aplicações; experiencia com as ferramentas de desenvolvimento; volatilidade da equipa;
- Fatores do projeto: desenvolvimento distribuído por vários locais; utilização de ferramentas de desenvolvimento; restrições/necessidades de horário.

Os modelos algorítmicos são modelos determinísticos, isto é, dado um conjunto de valores de entrada, o resultado será sempre igual. Como as fórmulas são conhecidas, estas podem ser analisadas e aperfeiçoadas o que permite não só ter um maior conhecimento do seu funcionamento, mas também personalizar os modelos de modo a que adequem melhor a determinados contextos (N. Sharma e Litoriya, 2012). Por outro lado, este tipo de modelos requer, para os fatores de custo, valores precisos de diversos atributos do projeto. Estes valores são difíceis de obter durante a fase inicial de um projeto de desenvolvimento de *software*. Os modelos também têm dificuldade em modelar as relações complexas entre os vários fatores, são incapazes de lidar com dados categóricos e não possuem nenhuma capacidade de raciocino (Sehra, Brar e Kaur, 2013).

Os modelos algorítmicos mais comuns incluem o número de linhas de código fonte (*Source lines of code* – SLOC), a *Function Point Analysis* (FPA) (Allan J. Albrecht e Gaffney, 1983), o Modelo Putnam e ainda o *Constructive Cost Model* (COCOMO) (B. W. Boehm, 1981) (Kumari e Pushkar, 2013b; Munialo e Muketha, 2016).

2.1.1 Source Lines of Code

O número de linhas de código fonte (*Source lines of code* – SLOC) é uma métrica utilizada para medir o tamanho de um determinado *software*. Esta medição é obtida através da contagem do número de linhas de código fonte do *software* (Balaji, Shivakumar e Ananth, 2013). O SLOC é o método de estimativa de esforço de *software* mais antigo, sendo originalmente utilizado para estimar o tamanho de *software* desenvolvido em *Fortran* e *Assembly*. Para produzir uma estimativa são utilizados dados históricos de *softwares* anteriores com aproximadamente o mesmo tamanho, cujos SLOC foram estimados e posteriormente comparados com o seu valor real. Esta estimativa será eventualmente utilizada para estimar o esforço e/ou custo do desenvolvimento do *software* (Khatibi e Jawawi, 2011).

As duas formas mais populares e aceites de contar SLOC são a contagem do número de linhas físicas de código fonte e o número de linhas logicas de código fonte. Apesar das definições específicas de ambas as formas de contagem variarem ligeiramente de autor para autor, geralmente o número de linhas físicas de código fonte conta todas as linhas exceto linhas em branco e comentários. Por outro lado, o número de linhas logicas de código fonte é geralmente definido como sendo a intenção de contar instruções individuais (em muitas linguagens denotado pela terminação num ponto e virgula). Desta forma a contagem de linhas logicas é independente do formato físico do código, o que significa que uma linha pode conter múltiplas instruções ou que uma instrução pode estar dividida por várias linhas. Devido a esta vantagem sobre a contagem de linhas físicas, a contagem do número de linhas logicas é a forma recomendada por vários métodos de estimativa de esforço de *software* (Nguyen et al., 2007).

Para além de não ser possível calcular o SLOC para todas as linguagens de programação, este cálculo não tem em conta a complexidade do *software*. Como os valores de SLOC estão largamente relacionados com a linguagem de programação utilizada em cada projeto a comparação de valores referentes a projetos realizados com recurso a diferentes linguagens de programação não é possível (Munialo e Muketha, 2016).

A estimativa do SLOC para determinado *software* pode ser obtida através de experiencias passadas, de projetos passados, de *softwares* concorrentes ou ainda através da divisão do *software* em vários componentes mais pequenos e do cálculo do SLOC de todos os componentes. Geralmente o SLOC é calculado utilizando três estimativas distintas para cada componente a estimar (Khatibi e Jawawi, 2011; Touesnard, 2004):

- Estimativa do SLOC mínimo - a
- Estimativa do SLOC mais provável - m
- Estimativa do SLOC máximo - b

O SLOC estimado de cada componente E_i pode ser calculado somando a estimativa mínima, a estimativa máxima e quatro vezes a estimativa mais provável e posteriormente dividindo o resultado por seis (Touesnard, 2004). Este cálculo pode ser representado por (2-2).

$$E_i = \frac{a + 4m + b}{6} \quad (2\text{-}2)$$

Por fim, o SLOC do *software* completo E é simplesmente a soma do SLOC estimado de cada um dos componentes, como é mostrado em (2-3).

$$E = \sum_{i=1}^{n} E_i \quad (2\text{-}3)$$

Onde n é o número total de componentes (Touesnard, 2004).

2.1.2 Function Point Analysis

A *Function Point Analysis* (FPA), desenvolvida por Allan Albrecht da IBM, foi inicialmente publicada em 1979 (A. J. Albrecht, 1979) e, em 1984, o *International Function Point Users Group* (IFPUG) foi criado para esclarecer regras, estabelecer padrões e promover seu uso e evolução. Apesar disto, o IFPUG permanece largamente desconhecido pela grande maioria dos profissionais da área (Abran e Robillard, 1994; Raju e Krishnegowda, 2013). Os *Function Points* (FP) têm como objetivo providenciar um método estandardizado de quantificação do tamanho e complexidade de um sistema em termos das funções que os sistemas oferecem ao utilizador (Abran e Robillard, 1994; Borade e Khalkar, 2013; Kumari e Pushkar, 2013b; Munialo e Muketha, 2016; N. Sharma, Bajpai e Litoriya, 2012).

Ao longo dos anos, foram feitas várias melhorias à especificação inicial de 1979, tendo sido publicadas versões sucessivas (Abran e Robillard, 1994). A mudança mais relevante foi a expansão do modelo em 1983 (Allan J. Albrecht e Gaffney, 1983). O modelo inicial, Albrecht 79, tinha em conta quatro tipos de funções, um conjunto de pesos (Tabela 2-1, esquerda) e 10 características gerais do sistema (*General System Characteristics* – GSC) para um fator de ajuste (*Value Adjustment Factor* – VAF) máximo de 25% (Tabela 2-2, coluna 1) (Abran e Robillard, 1994).

Tabela 2-1: Tipos de função e níveis de complexidade
Fonte: Function Points: A Study of Their Measurement Processes and Scale Transformations (Abran e Robillard, 1994)

Albrecht 79		Albrecht 83			
Function Types	Weights	Function Types	Low	Average	High
Files	10	Internal logical files	7	10	15
		External interface files	5	7	10
Inputs	4	External inputs	3	4	6
Outputs	5	External outputs	4	5	7
Inquiries	4	External inquiries	3	4	6

O modelo Albrecht 79 foi expandido para ter em conta cinco tipos de funções, três conjuntos de pesos (Tabela 2-1, direita) e 14 GSCs para um VAF máximo de 35% (Tabela 2-2, coluna 2) (Abran e Robillard, 1994) dando origem ao modelo Albrecht 83 (Allan J. Albrecht e Gaffney, 1983).

Tabela 2-2: Características gerais do sistema e graus de influência
Fonte: Function Points: A Study of Their Measurement Processes and Scale Transformations (Abran e Robillard, 1994)

General System Characteristics (GSC)		
Albrecht 79	Albrecht 83	Degrees of Influence
Backup	Data communications	0 – Not present/No influence
Data communications	Distributed functions	1 – Insignificant influence
Distributed processing	Performance	2 – Moderate influence
Performance issues	Heavily used configuration	3 – Average influence
Heavily used configuration	Transaction rate	4 – Significant influence
Online data entry	Online data entry	5 - Strong influence
Conversational data entry	End user efficiency	
Online update of master files	Online update	
Complex functions	Complex processing	
Complex internal processing	Reuseability	
	Installation ease	
	Operational ease	
	Multiple sites	
	Facilitate change	
VAF = (0.75 ± 25%)	VAF = (0.65 ± 35%)	

Os FP podem ser aplicados tanto na fase de especificação de requisitos bem como na fase de desenvolvimento do sistema (Allan J. Albrecht e Gaffney, 1983; N. Sharma, Bajpai e Litoriya, 2012). Além disto, os FP são independentes da linguagem e metodologia utilizada no desenvolvimento do sistema (Allan J. Albrecht e Gaffney, 1983) e ainda são fáceis de compreender por utilizadores com pouco ou nenhum conhecimento técnico. No entanto, os FP não podem ser utilizados em situações em que os requisitos do sistema são desconhecidos ou não são claros, como é o caso do desenvolvimento utilizando metodologias ágeis (Munialo e Muketha, 2016). Para além do esforço e do custo, os FP também podem ser utilizados para medir o número de erros, *bugs*, ou páginas de documentação por cada FP (Borade e Khalkar, 2013).

Os FP medem a funcionalidade do ponto de vista do utilizador, isto é, com base no que o utilizador solicita e recebe em retorno (Abran e Robillard, 1994; Borade e Khalkar, 2013; Kumari e Pushkar, 2013b; Munialo e Muketha, 2016; N. Sharma, Bajpai e Litoriya, 2012).

Do ponto de vista do utilizador final, a anatomia de um *software* pode ser vista como uma combinação de dados em repouso e dados em movimento. Os dados em repouso representam o armazenamento de dados na forma de ficheiros – chamados Funções de Dados. Os dados em movimento representam as atividades que os utilizadores realizam ao processar informações – chamadas Funções Transacionais (Raju e Krishnegowda, 2013).

As funções de dados representam agrupamentos lógicos de dados que os utilizadores necessitam para efetuar o seu trabalho. As funções de dados podem ser de dois tipos: *Internal Logical Files* (dados internos) and *External Interface Files* (dados externos) (Allan J. Albrecht e Gaffney, 1983; Borade e Khalkar, 2013; Raju e Krishnegowda, 2013).

- *Internal Logical Files* (ILF): é um conjunto de dados logicamente relacionados, que podem ser identificados pelo utilizador, que são mantidos dentro dos limites do sistema;
- *External Interface Files* (EIF): é um conjunto de dados logicamente relacionados, que podem ser identificados pelo utilizador, que são mantidos fora dos limites do sistema.

As funções transacionais representam a funcionalidade fornecida ao utilizador, pelo sistema, para o processamento de dados. As funções transacionais podem ser de três tipos: *External Inputs* (entradas externas), *External Outputs* (saídas externas) e *External Inquires* (consultas externas) (Allan J. Albrecht e Gaffney, 1983; Borade e Khalkar, 2013; Raju e Krishnegowda, 2013).

- *External Inputs* (EI): processam dados provenientes de fora dos limites do sistema. Os dados processados são mantidos por um ou mais ILFs;
- *External Outputs* (EO): enviam dados processados para fora dos limites do sistema;
- *External Inquires* (EQ): apresentam dados ao utilizador através da obtenção de dados de um ILF ou EIF.

Além das funções de dados e transacionais, a FPA também tem em consideração as características gerais do sistema (GSC) que classificam a funcionalidade geral e a complexidade do sistema. Essas características são utilizadas para ajustar a medida inicial de funções de dados e funções transacionais (Allan J. Albrecht e Gaffney, 1983; Borade e Khalkar, 2013; Raju e Krishnegowda, 2013).

- *Data communications*: refere-se a aspetos relacionados com os recursos utilizados para a comunicação de dados do sistema;
- *Distributed Functions*: refere-se à necessidade de o sistema utilizar dados ou processamento distribuído;
- Performance: refere-se à existência de restrições relativas ao tempo de resposta ou ao volume de processamento do sistema;
- *Heavily used configuration*: refere-se ao nível de utilização já existente dos equipamentos necessários para executar o sistema;
- *Transaction rate*: refere-se ao volume de transações do sistema;
- *Online data entry*: refere-se à quantidade de dados inseridos no modo *online* do sistema;
- *End-user efficiency*: refere-se à atenção dedicada à usabilidade das interfaces da aplicação;
- *Online update*: refere-se à quantidade de dados (ILFs) atualizados no modo *online* do sistema;

- *Complex processing*: refere-se à complexidade do processamento efetuado pelo sistema;
- *Reusability*: refere-se à preocupação com o reaproveitamento de partes do sistema em sistemas futuros;
- *Installation ease*: refere-se à facilidade com que é feita a instalação do sistema e a transição de sistemas anteriores;
- *Operational ease*: refere-se ao nível de automatização de processos como a inicialização, *backup* e recuperação e à minimização de tarefas manuais;
- *Multiple sites*: refere-se ao nível de preparação do sistema para ser instalado em múltiplos ambientes com *hardware* e *software* distintos;
- *Facilitate change*: refere-se ao nível de flexibilidade do sistema para alterações ao nível da sua logica de processamento ou das suas fontes e estruturas de dados.

O cálculo dos FP é realizado em três etapas (Allan J. Albrecht e Gaffney, 1983):

1. Contar e classificar os cinco tipos de função;
2. Calcular o ajuste tendo em conta a complexidade de processamento;
3. Cálculo dos FP finais.

A contagem dos tipos de função é alcançada contando as funções de dados (ILF e EIF) e as funções transacionais (EI, EO e EQ). A soma do numero de cada tipo de função, ponderada com o peso respetivo ao seu nível de complexidade (baixo, médio, alto) (Tabela 2-1), é chamada de *Unadjusted Function Points* (UFP) e pode ser representada por (2-4) (Allan J. Albrecht e Gaffney, 1983; Arifoglu, 1993; Borade e Khalkar, 2013; Khatibi e Jawawi, 2011; Kumari e Pushkar, 2013b; Munialo e Muketha, 2016; Raju e Krishnegowda, 2013).

$$UFP = \sum_{i=1}^{5} \sum_{j=1}^{3} N_{ij} W_{ij} \qquad (2\text{-}4)$$

Onde i representa um dos cinco tipos de funções, j representa um dos três níveis de complexidade, N_{ij} representa o número de funções do tipo i com complexidade j e W_{ij} representa o peso correspondente ao tipo de função i e ao nível de complexidade j.

O UFP pode ser agora ajustado através de um fator de ajuste (*Value Adjustment Factor* - VAF) para que seja tido em conta a complexidade de processamento do sistema. Este ajuste é baseado nas 14 GSCs e é calculado atribuindo um grau de influência (Tabela 2-2, coluna 3) a cada característica. O grau de influência a atribuir a cada GSC varia entre 1 e 5, se o GSC não tiver nenhuma influência ou se tiver uma grande importância no sistema respetivamente. A soma dos graus de influência dos GSCs é chamada *Technical Complexity Factor* (TCF) e pode ser representada por (2-5) (Allan J. Albrecht e Gaffney, 1983; Arifoglu, 1993; Borade e Khalkar, 2013; Khatibi e Jawawi, 2011; Kumari e Pushkar, 2013b; Munialo e Muketha, 2016; Raju e Krishnegowda, 2013).

$$TCF = \sum_{i=1}^{14} D_i \qquad (2\text{-}5)$$

Onde D_ié o grau de influência atribuído ao GSC i. O valor do TCF por sua vez utilizado para calcular o VAF. O calculo final do VAF pode ser representado por (2-6).

$$VAF = (TCF \times 0.01) + 0.65 \qquad (2\text{-}6)$$

O VAF pode variar deste 0.65 (se todos os D_i forem 0) e 1.35 (se todos os D_i forem 5), resultando num ajuste de ± 35% em relação ao UFP calculado inicialmente. O valor de UFP depois de ajustado pelo VAF é chamado de *Adjusted Function Points* (AFP), ou simplesmente FP, e pode ser representado por (2-7) (Allan J. Albrecht e Gaffney, 1983; Arifoglu, 1993; Borade e Khalkar, 2013; Khatibi e Jawawi, 2011; Kumari e Pushkar, 2013b; Munialo e Muketha, 2016; Raju e Krishnegowda, 2013).

$$AFP = UFP \times VAF \qquad (2\text{-}7)$$

2.1.3 Putnam Model

O Modelo Putnam foi publicado por Lawrence Putnam, da Quantitative Software Measurement (QSM), em 1978 (Barry Boehm, Abts e Chulani, 2000; Putnam, 1978). Este modelo tem por base a investigação realizada por Peter Norden (Norden, 1970), que por sua vez é baseada na distribuição de Rayleigh (Figura 2-1), inicialmente identificada por Lord Rayleigh (Tinnirello, 1999).

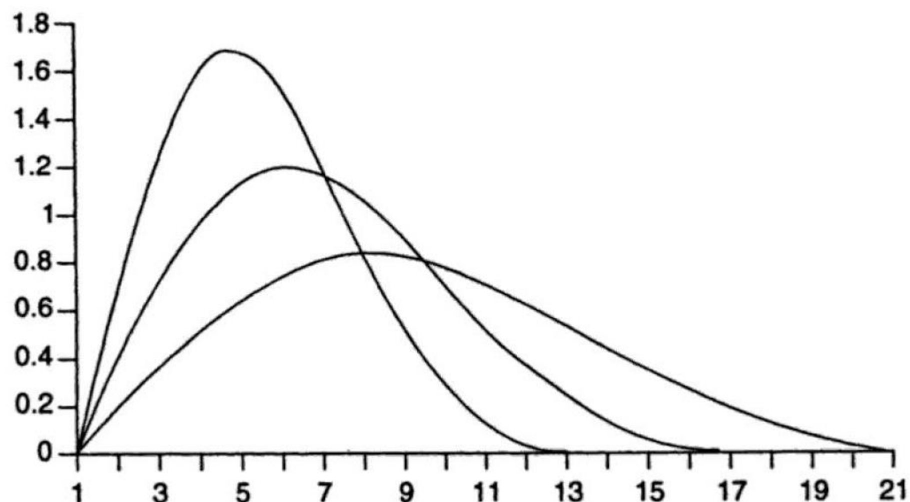

Figura 2-1: Distribuição de Rayleigh
Fonte: Systems development handbook (Tinnirello, 1999)

Norden assumiu que o esforço e a duração do tempo para projetos de investigação e desenvolvimento poderiam ser caracterizados utilizando curvas de Rayleigh. Norden colocou a hipótese de que as curvas de Rayleigh também poderiam descrever a forma como as pessoas abordam e resolvem os problemas, e que os projetos de investigação e desenvolvimento são apenas uma série de oportunidades de resolução de problemas. Portanto, cada fase do desenvolvimento de um novo projeto seria conforme à forma de uma curva de Rayleigh (Figura 2-2), e a soma de todas as curvas individuais também seguiria uma distribuição de Rayleigh (Norden, 1970; Tinnirello, 1999).

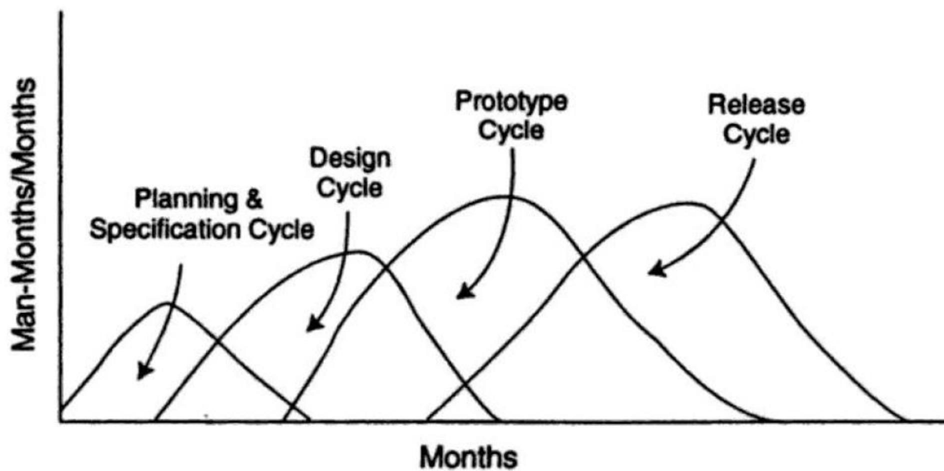

Figura 2-2: Distribuição de Rayleigh aplicada às fases de desenvolvimento de um projeto
Fonte: Systems development handbook (Tinnirello, 1999)

Putnam estendeu o trabalho de Norden aplicando as curvas de Rayleigh às fases do ciclo de desenvolvimento de *software* (Figura 2-3), e ao próprio ciclo. Putnam validou o seu trabalho recorrendo a dados de cerca de duzentos sistemas (de grandes dimensões) desenvolvidos para o Comando de Sistemas de Computadores do Exército dos EUA (*U.S. Army Computer Systems Command* – USACSC). Para a maioria dos sistemas, o processo de desenvolvimento de *software* seguiu o modelo de Norden/Rayleigh (Putnam, 1978; Tinnirello, 1999).

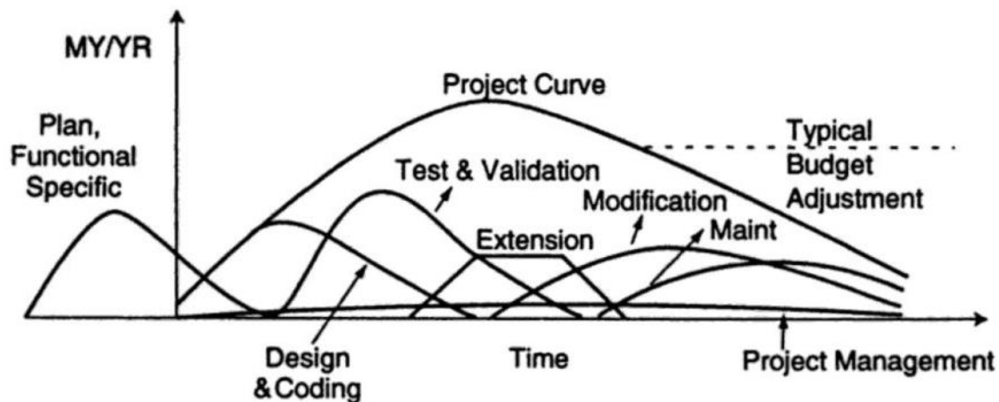

Figura 2-3: Distribuição de Rayleigh aplicada ao desenvolvimento de *software*
Fonte: A general empirical solution to the macro software sizing and estimating problem (Putnam, 1978)

A curva utilizada, por Putnam, para descrever o processo de desenvolvimento de *software* pode ser representada por (2-8) (Putnam, 1978; Tinnirello, 1999).

$$y = K(1 - e^{-at^2}) \tag{2-8}$$

Onde y é o esforço acumulado gasto no desenvolvimento do *software* em determinado instante do tempo t (Figura 2-4), K é o esforço total necessário durante a totalidade do ciclo de vida do *software*, a é o parâmetro que afeta a forma da curva e t é o tempo decorrido desde o início do projeto. (Basha e Ponnurangam, 2010; Putnam, 1978; Tinnirello, 1999).

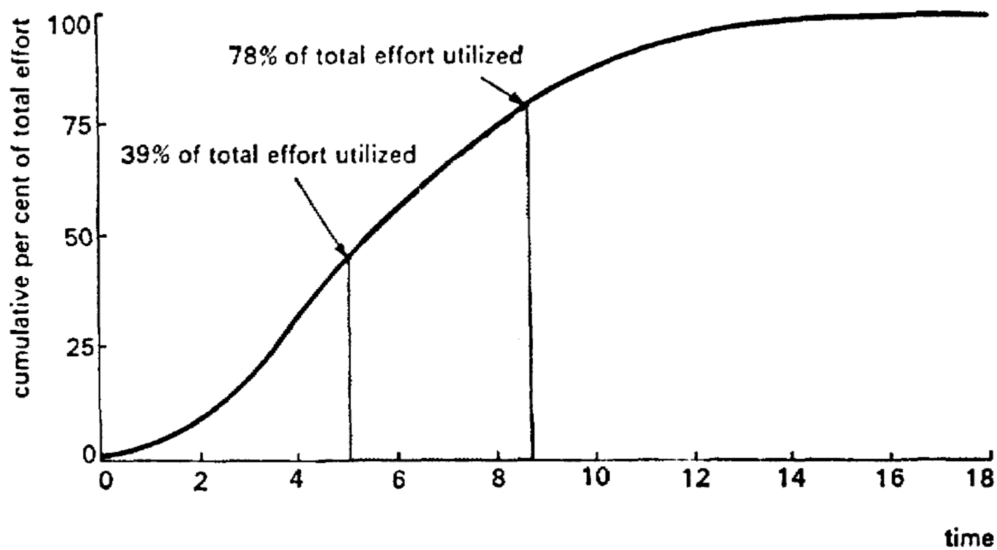

Figura 2-4: Esforço acumulado gasto no desenvolvimento do *software*
Fonte: A general empirical solution to the macro software sizing and estimating problem (Putnam, 1978)

A primeira derivada de (2-8) apresenta a forma de uma distribuição de Rayleigh e representa genericamente qualquer fase do ciclo de desenvolvimento de *software*, incluindo o próprio ciclo. A primeira derivada de (2-8) pode ser representada por (2-9) (Putnam, 1978; Tinnirello, 1999).

$$y' = 2Kate^{-at^2} \tag{2-9}$$

Onde y' representa a distribuição de esforço gasto no desenvolvimento do *software* ao longo do tempo t (Figura 2-5), K é a área da região limitada superiormente pela curva e representa o esforço total necessário durante a totalidade do ciclo de vida do *software*, a é o parâmetro que afeta a forma da curva e t é o tempo decorrido desde o início do projeto. (Basha e Ponnurangam, 2010; Putnam, 1978; Tinnirello, 1999).

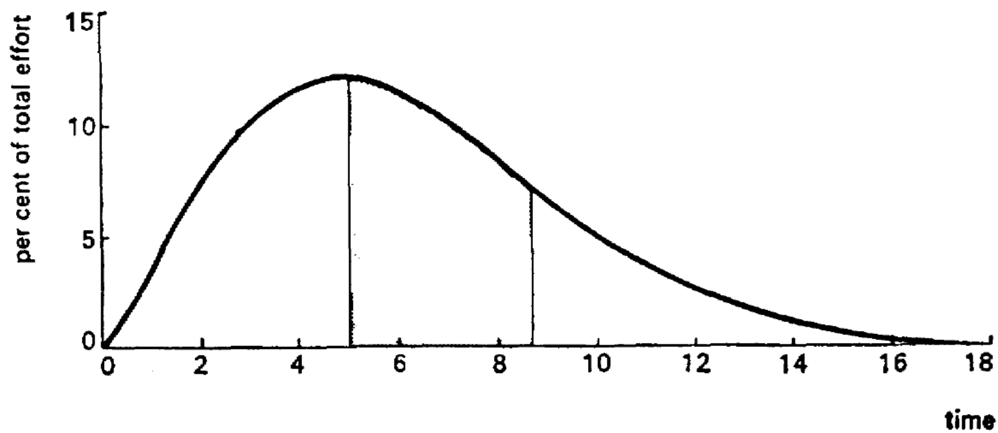

Figura 2-5: Distribuição de esforço gasto no desenvolvimento do *software*
Fonte: A general empirical solution to the macro software sizing and estimating problem (Putnam, 1978)

O parâmetro a, que afeta a forma das curvas, é obtido através da resolução da primeira derivada de (2-8) em ordem a a quando y' é máximo. Assim, a pode ser definido por (2-10) (Putnam, 1978; Tinnirello, 1999).

$$a = \frac{1}{2{t_d}^2} \tag{2-10}$$

Onde t_d é o instante de tempo t em que y' é máximo. Putnam determinou que y'_{max} ocorre quando t é aproximadamente igual ao instante de tempo em que o *software* se torna operacional, o que equivale sensivelmente ao fim da fase de desenho e desenvolvimento do *software*. Baseado na curva apresentada na Figura 2-4, Putnam concluiu que cerca de 39.35% do esforço total é gasto no desenvolvimento do *software* e o esforço restante é gasto em testes, manutenção e modificações (Barry Boehm, Abts e Chulani, 2000; Putnam, 1978; Tinnirello, 1999).

Baseado no modelo de Norden/Rayleigh e na sua análise de cerca de duzentos projetos, Putman determinou vários outros relacionamentos matemáticos que caracterizam o processo de desenvolvimento de *software*. Dois dos relacionamentos mais uteis para a estimativa de esforço de desenvolvimento de *software* são a dificuldade de um projeto e a equação do *software* (*software equation*) (Putnam, 1978; Tinnirello, 1999).

Putnam observou que a dificuldade de um projeto D podia ser representada por (2-11) (Putnam, 1978; Tinnirello, 1999).

$$D = \frac{K}{{t_d}^2} \tag{2-11}$$

Onde K é o esforço total necessário durante a totalidade do ciclo de vida do *software* e t_d é o tempo de desenvolvimento do *software*. Um valor baixo de D indica que o *software* é

relativamente fácil de desenvolver e um valor elevado de D indica que a dificuldade de desenvolver o *software* será elevada (Putnam, 1978; Tinnirello, 1999).

Equação do *software* (*software equation*) foi a denominação dada por Putnam à relação encontrada entre o tamanho do *software*, o esforço total necessário para o seu desenvolvimento, o tempo de desenvolvimento e uma constante que representa o estado atual da tecnologia (Putnam, 1978). Esta relação pode ser representada por (2-12) (Borade e Khalkar, 2013; Leung e Fan, 2002; Putnam, 1978).

$$S = C_k K^{\frac{1}{3}} t_d^{\frac{4}{3}} \qquad (2\text{-}12)$$

Onde S é o tamanho do *software*, C_k é uma constante que representa o estado atual da tecnologia, K o esforço total necessário durante a totalidade do ciclo de vida do *software* e t_d é o tempo de desenvolvimento do *software*. C_k é um fator que reflete os efeitos de vários fatores sobre a produtividade, como restrições de hardware, complexidade do programa, ambiente de programação e experiência pessoal (Basha e Ponnurangam, 2010; Putnam, 1978). Este fator pode ser escolhido recorrendo aos valores originalmente encontrados por Putnam (Arifoglu, 1993; Putnam, 1978):

- 2 para um ambiente de desenvolvimento de *software* fraco (por exemplo, desenvolvimento sem metodologias e fraca ou nenhuma documentação);
- 8 para um bom ambiente de desenvolvimento de *software* (por exemplo, utilização de uma metodologia de desenvolvimento e documentação adequada);
- 11 para um ambiente excelente (por exemplo, utilização de uma metodologia de desenvolvimento, utilização de ferramentas automatizadas e controlo de qualidade).

Porém é recomendado que este fator seja determinado por cada organização utilizando informações de projetos anteriores (Arifoglu, 1993; Borade e Khalkar, 2013; Kumari e Pushkar, 2013b; Leung e Fan, 2002; Putnam, 1978), recorrendo à formula apresentada em (2-13).

$$C_k = \frac{S}{K^{\frac{1}{3}} t_d^{\frac{4}{3}}} \qquad (2\text{-}13)$$

Dependendo dos valores disponíveis no momento da utilização da equação, esta pode ser manipulada de modo a obter não só o tamanho do *software*, mas também o esforço total necessário durante a totalidade do ciclo de vida do *software*, apresentado em (2-14), ou o tempo de desenvolvimento do *software*, apresentado em (2-15).

$$K = \left(\frac{S}{C_k t_d^{\frac{4}{3}}} \right)^3 \qquad (2\text{-}14)$$

$$t_d = \left(\frac{S}{C_k K^{\frac{1}{3}}} \right)^{\frac{3}{4}} \qquad (2\text{-}15)$$

O Modelo Putnam é principalmente utilizado para estimar custos, esforço e tempo necessário ao longo do ciclo de vida do *software*. Os vários relacionamentos do modelo são também frequentemente utilizados para avaliar as consequências de diversas limitações ou alterações no tempo ou esforço dos projetos (Tinnirello, 1999). O modelo é extremamente sensível ao tempo de desenvolvimento. A diminuição do tempo de desenvolvimento pode aumentar consideravelmente o esforço necessário para o desenvolvimento do *software* (Khuttan, Kumar e Singh, 2014; Kumari e Pushkar, 2013b; Putnam, 1978; T. N. Sharma, Bhardwaj e Sharma, 2011). Apesar de o modelo ter sido desenvolvido recorrendo à unidade de tempo anos e à medida de tamanho SLOC, o modelo pode ser utilizado com outras unidades de tempo e tamanho, como por exemplo, meses e *Function Points*.

Um dos problemas inerentes ao modelo é a sua ineficácia em estimar projetos de pequena e média dimensão, geralmente produzindo uma estimativa demasiado elevada. Outro problema resulta do facto de o modelo ser baseado no conhecimento prévio, ou na capacidade de estimar com precisão, o tamanho do *software* a ser desenvolvido. Porém, na grande maioria dos seus cenários de utilização, existe grande incerteza no tamanho do *software* a ser desenvolvido, o que pode resultar na imprecisão da estimativa de custo (Khuttan, Kumar e Singh, 2014; Kumari e Pushkar, 2013b; T. N. Sharma, Bhardwaj e Sharma, 2011).

O Modelo Putnam é frequentemente apelidado incorretamente de *Software Life Cycle Model* (SLIM). Contudo, o SLIM é um conjunto de ferramentas proprietárias da Quantitative Software Measurement que utiliza os conceitos definidos no Modelo Putnam (Tinnirello, 1999).

2.1.4 Constructive Cost Model

O *Constructive Cost Model* (COCOMO, também conhecido por COCOMO 81), desenvolvido por Barry Boehm da TRW Aerospace, foi inicialmente publicado em 1981 (B. W. Boehm, 1981). É um conjunto de modelos hierárquicos que tem por base o estudo de experiencias anteriores em projetos de *software*, mais concretamente, de 63 projetos da TRW Aerospace. Boehm propôs uma hierarquia com três níveis composta pelos modelos Basic COCOMO, Intermediate COCOMO e Detailed COCOMO (Basha e Ponnurangam, 2010; B. W. Boehm, 1981; Borade e Khalkar, 2013; Kemerer, 1987; Khatibi e Jawawi, 2011; Munialo e Muketha, 2016).

Independentemente do modelo a utilizar, deve ser atribuído um modo de desenvolvimento ao projeto de desenvolvimento de *software* a estimar com base na sua complexidade. Segundo Boehm qualquer projeto de desenvolvimento de *software* pode ser classificado num dos seguintes três modos de desenvolvimento: *organic*, *semidetached* ou *embedded*. Esta classificação é feita considerando não só as características do produto, mas também as da equipa de desenvolvimento e do ambiente de desenvolvimento (Tabela 2-3) (Arifoglu, 1993; B. W. Boehm, 1981; Menzies et al., 2006; Merlo–Schett, Glinz e Mukhija, 2002).

Tabela 2-3: Modos de desenvolvimento
Fonte: Software engineering economics (B. W. Boehm, 1981)

	Mode		
Feature	Organic	Semidetached	Embedded
Organizational understanding of product objectives	Thorough	Considerable	General
Experience in working with related software systems	Extensive	Considerable	Moderate
Need for software conformance with pre-established requirements	Basic	Considerable	Full
Need for software conformance with external interface specifications	Basic	Considerable	Full
Concurrent development of associated new hardware and operational procedures	Some	Moderate	Extensive
Need for innovative data processing architectures, algorithms	Minimal	Some	Considerable
Premium on early completion	Low	Medium	High
Product size range	< 50 KDSI	< 300 KDSI	All Size

O modelo Basic COCOMO é um modelo de valor único, que calcula o esforço de desenvolvimento de *software* em função do tamanho do *software*, expresso em milhares de linhas de código fonte entregues (*Thousand delivered source instructions* – KDSI) (B. W. Boehm, 1981; Merlo–Schett, Glinz e Mukhija, 2002; Pressman, 1997). Este cálculo pode ser representado por (2-16).

$$E = a\,(KDSI)^b \qquad (2\text{-}16)$$

Onde E é esforço total de desenvolvimento de *software* em meses e os coeficientes a e b são dados em função do modo de desenvolvimento previamente escolhido (Tabela 2-4).

Tabela 2-4: Coeficientes Basic COCOMO
Fonte: Performance Analysis of the Software Cost Estimation Methods: A Review (Kumari e Pushkar, 2013b)

Mode	a	b	c	d
Organic	2.4	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Para além do esforço total de desenvolvimento de *software*, o COCOMO permite também calcular o tempo de desenvolvimento e o número de recursos necessários para concluir o projeto de *software* (B. W. Boehm, 1981). Estes cálculos podem ser representados por (2-17) e (2-18) respetivamente.

$$D = c\,(E)^d \tag{2-17}$$

$$P = \frac{E}{D} \tag{2-18}$$

Onde D é o tempo de desenvolvimento em meses, E é esforço total de desenvolvimento de *software* em meses, os coeficientes c e d são dados em função do modo de desenvolvimento previamente escolhido (Tabela 2-4) e P é o número de recursos necessários.

O modelo Basic COCOMO é o mais simples e fácil de utilizar, porém como não são considerados fatores de custo, este pode apenas ser utilizado como uma estimativa muito grosseira (Borade e Khalkar, 2013; Leung e Fan, 2002).

A falta de precisão do Basic COCOMO deu origem ao Intermediate COCOMO. O modelo Intermediate COCOMO calcula o esforço de desenvolvimento de *software* em função do tamanho do *software* e de um conjunto de fatores de ajuste de esforço (*Effort adjustment factors* – EAF) que incluem avaliações subjetivas de atributos do produto, hardware, pessoal e projeto (Arifoglu, 1993; B. W. Boehm, 1981; Borade e Khalkar, 2013; Kumari e Pushkar, 2013b; Leung e Fan, 2002; Merlo–Schett, Glinz e Mukhija, 2002; Munialo e Muketha, 2016; Pressman, 1997).

Atributos do produto:

- RELY – *Required software reliability*: refere-se ao nível de fiabilidade do *software* e das consequências da ocorrência de uma falha, que podem variar desde uma pequena inconveniência até à perde de vidas humanas;
- DATA – *Database size*: refere-se ao tamanho da base de dados e à quantidade de dados a ser armazenada neste e noutros meios de armazenamento;

- CPLX – *Product complexity*: refere-se a uma avaliação subjetiva da complexidade de quatro tipos de função do *software*, sendo eles funções de controlo, funções de computação, funções de input/output e funções de gestão de dados.

Atributos de *hardware*:

- TIME – *Execution time constraint*: refere-se à existência de restrições relativas ao tempo de execução do *software*;
- STOR – *Main storage constraint*: refere-se à percentagem de espaço de armazenamento utilizado pelo *software* em relação ao espaço total disponível;
- VIRT – *Virtual machine volatility*: refere-se ao nível de volatilidade da máquina virtual subjacente ao *software* a ser desenvolvido. A máquina virtual é definida como o conjunto de *hardware* e *software* que o produto recorrerá para realizar suas tarefas;
- TURN – *Computer turnaround time*: refere-se ao tempo de resposta médio desde o momento em que um elemento da equipa de desenvolvimento submete uma tarefa para ser executada até que os resultados estejam disponíveis.

Atributos de pessoal:

- ACAP – *Analyst capability*: refere-se à capacidade, eficiência, rigor e capacidade de comunicação e cooperação dos analistas;
- AEXP – *Applications experience*: refere-se ao nível de experiência da equipa do projeto que desenvolve *software* em projetos semelhantes;
- PCAP – *Programmer capability*: refere-se à habilidade, eficiência, rigor e capacidade de comunicação e cooperação dos programadores;
- VEXP – *Virtual machine experience*: refere-se à experiencia da equipa no conjunto de *hardware* e *software* que o produto recorrerá para realizar suas tarefas;
- LEXP – *Programming language experience*: refere-se ao nível de experiência da equipa do projeto que desenvolve *software* com a linguagem de programação.

Atributos do projeto:

- MODP – *Use of modern programming practices*: refere-se ao nível de utilização de práticas de programação modernas no desenvolvimento do *software*;
- TOOL – *Use of software tools*: refere-se ao nível de utilização de ferramentas de *software* no desenvolvimento do *software*;
- SCED – *Required development schedule*: refere-se à restrição de tempo para o desenvolvimento do *software*. Esta classificação é feita em termos de percentagem em relação à duração típica do desenvolvimento de um projeto com esforço semelhante.

O Intermediate COCOMO tem por base a mesma equação do Basic COCOMO, porém neste modelo os quinze fatores de ajuste de custo são classificados numa escada de "muito baixo" até

"extremamente alto" de modo a obter os multiplicadores de esforço específicos de cada fator (B. W. Boehm, 1981; Merlo–Schett, Glinz e Mukhija, 2002; Pressman, 1997).

Tabela 2-5: Multiplicadores de esforço Intermediate COCOMO
Fonte: Software engineering economics (B. W. Boehm, 1981)

	Ratings					
Cost Drivers	Very Low	Low	Nominal	High	Very High	Extra High
Product attributes						
RELY	0.75	0.88	1.00	1.15	1.40	
DATA		0.94	1.00	1.08	1.16	
CPLX	0.7	0.85	1.00	1.15	1.30	1.65
Hardware attributes						
TIME			1.00	1.11	1.30	1.66
STOR			1.00	1.06	1.21	1.56
VIRT		0.87	1.00	1.15	1.30	
TURN		0.87	1.00	1.07	1.15	
Personnel attributes						
ACAP	1.46	1.19	1.00	0.86	0.71	
AEXP	1.29	1.13	1.00	0.91	0.82	
PCAP	1.42	1.17	1.00	0.86	0.70	
VEXP	1.21	1.10	1.00	0.90		
LEXP	1.14	1.07	1.00	0.95		
Project attributes						
MODP	1.24	1.10	1.00	0.91	0.82	
TOOL	1.24	1.10	1.00	0.91	0.83	
SCED	1.23	1.08	1.00	1.04	1.10	

O produto dos multiplicadores de todos os fatores é denominado de EAF total (Merlo–Schett, Glinz e Mukhija, 2002; Pressman, 1997) e pode ser representado por (2-19).

$$EAF = \prod_{i=1}^{15} EM_i \qquad (2\text{-}19)$$

Onde EM_i é o multiplicador de esforço do fator i. Valores típicos do EAF variam entre 0.9 e 1.4. O cálculo final do modelo Intermediate COCOMO pode ser representado por (2-20) (Borade e Khalkar, 2013; Kumari e Pushkar, 2013b; Leung e Fan, 2002; Merlo–Schett, Glinz e Mukhija, 2002; Munialo e Muketha, 2016; Pressman, 1997).

$$E = a\,(KDSI)^b \times EAF \qquad (2\text{-}20)$$

Além da adição do EAF, o coeficiente a é ligeiramente diferente do Basic COCOMO (Tabela 2-6). Os restantes coeficientes e o calculo o tempo de desenvolvimento e do número de recursos necessários para concluir o projeto permanecem iguais ao Basic COCOMO (Borade e Khalkar, 2013; Leung e Fan, 2002; Merlo–Schett, Glinz e Mukhija, 2002).

Tabela 2-6: Coeficientes Intermediate COCOMO
Fonte: Software engineering economics (B. W. Boehm, 1981)

Mode	a	b	c	d
Organic	3.2	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	2.8	1.20	2.5	0.32

O último modelo da hierarquia do COCOMO 81 é o Detailed COCOMO. Este modelo é uma extensão do Intermediate COCOMO e tem como objetivo aumentar a precisão da estimativa em projetos de grande dimensão. O modelo Detailed COCOMO calcula o esforço de desenvolvimento de *software* em função do tamanho do *software* e um conjunto de fatores de ajuste de esforço (EAF) ponderados de acordo com cada fase do ciclo de vida do *software*. O Detailed COCOMO aplica o modelo Intermediate COCOMO a cada componente do *software* em cada fase de desenvolvimento do mesmo (B. W. Boehm, 1981; Borade e Khalkar, 2013; Khatibi e Jawawi, 2011; Kumari e Pushkar, 2013b; Merlo–Schett, Glinz e Mukhija, 2002; Munialo e Muketha, 2016; P. Sharma, 2004):

- *Requirements planning and product design* (RPD)
- *Detailed design* (DD)
- *Code and unit test* (CUT)
- *Integration and test* (IT)

Os quinze fatores são estimados e aplicados a cada fase separadamente, em vez de ao projeto como um todo. Cada fator é dividido por fases e assume um multiplicador de esforço diferente em cada uma delas, como no exemplo mostrado na Tabela 2-7 (Arifoglu, 1993; B. W. Boehm, 1981; Kemerer, 1987; Merlo–Schett, Glinz e Mukhija, 2002; P. Sharma, 2004).

Tabela 2-7: Multiplicadores de esforço *Analyst capability Detailed* COCOMO
Fonte: Software Engineering (P. Sharma, 2004)

Cost Driver	Rating	RPD	DD	CUT	IT
ACAP	Very Low	1.80	1.35	1.35	1.50
	Low	0.85	0.85	0.85	1.20
	Nominal	1.00	1.00	1.00	1.00
	High	0.75	0.90	0.90	0.85
	Very High	0.55	0.75	0.75	0.70

Enquanto que tanto o Basic COCOMO, bem como o Intermediate COCOMO, estimam o esforço do *software* ao nível do sistema completo, o Detailed COCOMO é aplicado em cada

subsistema separadamente o que é uma grande vantagem para sistemas de grandes dimensões que contêm subsistemas não homogêneos (Borade e Khalkar, 2013; Leung e Fan, 2002).

Em 1995 (Barry Boehm et al., 1995) começou a ser desenvolvido o COCOMO II e foi finalmente publicado em 2000 (Barry Boehm, Abts, et al., 2000). O COCOMO II é o sucessor do COCOMO 81 e é mais adequado para estimar projetos modernos de desenvolvimento de *software*. Este modelo fornece um maior suporte aos processos de desenvolvimento de *software* mais modernos e tem por base um conjunto de projetos mais atualizados. A necessidade do novo modelo foi surgindo à medida que as tecnologias de desenvolvimento de *software* avançaram, com processos de desenvolvimento rápidos e não sequenciais, reengenharia, abordagens orientadas para a reutilização, abordagens orientadas a objetos, etc. (Barry Boehm, Abts e Chulani, 2000). O COCOMO II é uma extensão do Intermediate COCOMO e é composto por três variantes (Barry Boehm, Abts, et al., 2000; Barry Boehm, Clark, et al., 2000; Barry Boehm et al., 1995; Kumari e Pushkar, 2013b):

- Application Composition Model: utilizado durante a prototipagem;
- Early Design Model: utilizado quando os requisitos ainda não são todos conhecidos para obter uma estimativa de alto nível;
- Post-Architecture Model: utilizado quando os requisitos são todos conhecidos e o *software* está pronto para ser desenvolvido o que permite uma estimativa mais precisa.

Os modelos Early Design e Post-Architecture utilizam uma abordagem idêntica aos modelos COCOMO 81 para o cálculo do esforço de desenvolvimento de *software*, representada por (2-21) (Basha e Ponnurangam, 2010; Barry Boehm, Abts, et al., 2000; Barry Boehm, Clark, et al., 2000; Khatibi e Jawawi, 2011).

$$E = A\ (Size)^B\ \times\ EAF \qquad (2\text{-}21)$$

Onde A é um coeficiente cujo valor nominal segundo o método é 2.94 ou pode ser calibrado de acordo com projetos passados da organização que utiliza o método e $Size$ é o tamanho estimado do *software* em SLOC ou FP (estimativas em FP devem ser convertidas para o SLOC equivalente antes de serem utilizadas). A diferença mais significativa em relação aos modelos COCOMO 81 é que o coeficiente B deixa de ser um valor estático dependente do modo de desenvolvimento do *software* e passa a ser dependente de cinco *scale factors* (SF) (Basha e Ponnurangam, 2010; Barry Boehm, Abts, et al., 2000; Barry Boehm, Clark, et al., 2000; Barry Boehm et al., 1995; Khatibi e Jawawi, 2011; Kumari e Pushkar, 2013b; Leung e Fan, 2002):

- PREC – *Precedentedness*: refere-se à semelhança do *software* a ser desenvolvido com projetos previamente desenvolvidos;
- FLEX – *Development flexibility*: refere-se à necessidade de os desenvolvimentos seguirem normas e requisitos preestabelecidos;

- RESL – *Architecture/Risk resolution*: refere-se à identificação e resolução de itens de risco;
- TEAM – *Team cohesion*: refere-se às possíveis fontes de turbulência e perturbação do projeto devido a dificuldades na sincronização das partes interessadas do projeto, como por exemplo entre os utilizadores, o cliente e equipa de desenvolvimentos;
- PMAT – *Process maturity*: refere-se ao quão bem os comportamentos, práticas e processos de uma organização podem de forma confiável e sustentável produzir os resultados necessários (a avaliação é baseada no *Capability Maturity Model* - CMM).

Analogamente ao que acontece com os fatores de ajuste de esforço também os *scale factors* têm multiplicadores de esforço associados a cada fator (Tabela 2-8).

Tabela 2-8: Multiplicadores de esforço *scale factors* COCOMO II
Fonte: COCOMO II Model Definition Manual (Barry Boehm, Clark, et al., 2000)

	Ratings					
Scale Factors	Very Low	Low	Nominal	High	Very High	Extra High
PREC	6.20	4.96	3.72	2.48	1.24	0.00
FLEX	5.07	4.05	3.04	2.03	1.01	0.00
RESL	7.07	5.65	4.24	2.83	1.41	0.00
TEAM	5.48	4.38	3.29	2.19	1.10	0.00
PMAT	7.80	6.24	4.68	3.12	1.56	0.00

Assim o coeficiente B de (2-21) é representado por (2-22) (Basha e Ponnurangam, 2010; Barry Boehm, Abts, et al., 2000; Barry Boehm, Clark, et al., 2000; Khatibi e Jawawi, 2011).

$$B = b + 0.01 \times \sum_{j=1}^{5} SF_j \tag{2-22}$$

Onde b é um coeficiente cujo valor nominal segundo o método é 0.91 ou pode ser calibrado de acordo com projetos passados da organização que utiliza o método e SF_j é o multiplicador de esforço do *scale factor* j.

Outras diferenças incluem alterações nos fatores de ajuste de esforço (EAF) e nos seus respetivos multiplicadores de esforço que são agora baseados na análise de um conjunto de 163 de projetos mais atualizados (Barry Boehm, Abts, et al., 2000; Leung e Fan, 2002). No COCOMO II o cálculo do EAF total é representado por (2-23).

$$EAF = \prod_{i=1}^{n} EM_i \tag{2-23}$$

Onde o valor de n, que representa o número que fatores de esforço utilizados no calculo, é sete para o modelo Early Design e dezassete para o modelo Post-Architecture. Os fatores utilizados em cada modelo são apresentados na Tabela 2-9 (Barry Boehm, Abts, et al., 2000; Barry Boehm, Clark, et al., 2000; Barry Boehm et al., 1995).

Tabela 2-9: Fatores de esforço Early Design e Post-Architecture
Fonte: COCOMO II Model Definition Manual (Barry Boehm, Clark, et al., 2000)

Early Design Cost Driver	Counterpart Combined Post-Architecture Cost Drivers
PERS	ACAP, PCAP, PCON
RCPX	RELY, DATA, CPLX, DOCU
RUSE	RUSE
PDIF	TIME, STOR, PVOL
PREX	APEX, PLEX, LTEX
FCIL	TOOL, SITE
SCED	SCED

Alguns dos fatores do COCOMO II apresentados na Tabela 2-9 são provenientes do COCOMO 81 e apenas contaram com uma atualização dos seus multiplicadores de esforço, porém foram também adicionados alguns fatores novos (Barry Boehm, Abts, et al., 2000; Barry Boehm, Clark, et al., 2000; Barry Boehm et al., 1995). Ao modelo Post-Architecture foram adicionados os seguintes fatores:

Atributos do produto:

- RUSE – *Developed for reusability*: refere-se ao esforço adicional necessário para desenvolver componentes destinados à reutilização em projetos atuais ou futuros;
- DOCU – *Documentation match to life-cycle needs*: refere-se à adequação da documentação do projeto às suas necessidades.

Atributos de *hardware*:

- PVOL – *Platform Volatility*: equivalente ao fator VIRT do COCOMO 81. O termo plataforma tem o mesmo significado que máquina virtual no COCOMO 81.

Atributos de pessoal:

- PEXP – *Platform Experience*: equivalente ao fator VEXP do COCOMO 81;
- LTEX – *Language and Tool Experience*: equivalente ao fator LEXP do COCOMO 81;
- PCON – *Personnel Continuity*: refere-se à rotatividade anual de recursos do projeto.

Atributos do projeto:

- SITE – *Multisite Development*: refere-se à localização física da equipa de desenvolvimento e aos meios de comunicação existente entre estes.

A Tabela 2-10 apresenta os multiplicadores de esforço referentes aos fatores de esforço do modelo Post-Architecture.

Tabela 2-10: Multiplicadores de esforço COCOMO II Post-Architecture
Fonte: COCOMO II Model Definition Manual (Barry Boehm, Clark, et al., 2000)

	Ratings					
Cost Drivers	Very Low	Low	Nominal	High	Very High	Extra High
Product attributes						
RELY	0.82	0.92	1.00	1.10	1.26	
DATA		0.90	1.00	1.14	1.28	
CPLX	0.73	0.87	1.00	1.17	1.34	1.74
RUSE		0.95	1.00	1.07	1.15	1.24
DOCU	0.81	0.91	1.00	1.11	1.23	
Hardware attributes						
TIME			1.00	1.11	1.29	1.63
STOR			1.00	1.05	1.17	1.46
PVOL		0.87	1.00	1.15	1.30	
Personnel attributes						
ACAP	1.42	1.19	1.00	0.85	0.71	
PCAP	1.34	1.15	1.00	0.88	0.76	
AEXP	1.22	1.10	1.00	0.88	0.81	
PEXP	1.19	1.09	1.00	0.91	0.85	
LTEX	1.20	1.09	1.00	0.91	0.84	
PCON	1.34	1.15	1.00	0.88	0.76	
Project attributes						
TOOL	1.17	1.09	1.00	0.90	0.78	
SITE	1.22	1.09	1.00	0.93	0.86	0.80
SCED	1.43	1.14	1.00	1.00	1.00	

Devido à possível falta de informação na fase do projeto em que é aplicado os fatores presentes no modelo Early Design são uma combinação dos fatores do modelo Post-Architecture.

Atributos do produto:

- RCPX – *Product Reliability and Complexity*: combinação dos fatores RELY, DATA, CPLX e DOCU;
- RUSE – *Developed for reusability*: este fator é igual ao RUSE presente no modelo Post-Architecture.

Atributos de *hardware*:

- PDIF – *Platform Difficulty*: combinação dos fatores TIME, STOR e PVOL.

Atributos de pessoal:

- PERS – *Personnel Capability*: combinação dos fatores ACAP, PCAP e PCON;
- PREX – *Personnel Experience*: combinação dos fatores APEX, LTEX e PLEX.

Atributos do projeto:

- FCIL – *Facilities*: combinação dos fatores TOOL e SITE;
- SCED – *Required development schedule*: este fator é igual ao SCED presente no modelo Post-Architecture.

A Tabela 2-11 apresenta os multiplicadores de esforço referentes aos fatores de esforço do modelo Early Design.

Tabela 2-11: Multiplicadores de esforço COCOMO II Early Design
Fonte: COCOMO II Model Definition Manual (Barry Boehm, Clark, et al., 2000)

	Ratings						
Cost Drivers	Extra Low	Very Low	Low	Nominal	High	Very High	Extra High
Product attributes							
RUSE			0.95	1.00	1.07	1.15	1.24
RCPX	0.49	0.60	0.83	1.00	1.33	1.91	2.72
Hardware attributes							
PDIF			0.87	1.00	1.29	1.81	2.61
Personnel attributes							
PERS	2.12	1.62	1.26	1.00	0.83	0.63	0.50
PREX	1.69	1.33	1.22	1.00	0.87	0.74	0.62
Project attributes							
FCIL	1.43	1.30	1.10	1.00	0.87	0.73	0.62
SCED		1.43	1.14	1.00	1.00	1.00	

Tal como acontece com o esforço também o tempo de desenvolvimento segue uma abordagem idêntica aos modelos COCOMO 81, representada por (2-24). Neste caso a diferença mais significativa em relação aos modelos COCOMO 81 também é que o coeficiente d deixa de ser um valor estático dependente do modo de desenvolvimento do *software* e passa a ser dependente de cinco *scale factors* (SF) (Barry Boehm, Abts, et al., 2000; Barry Boehm, Clark, et al., 2000; Barry Boehm et al., 1995; Khatibi e Jawawi, 2011).

$$D = C\ (E)^F \qquad (2\text{-}24)$$

Onde C é um coeficiente cujo valor nominal segundo o método é 3.67 ou pode ser calibrado de acordo com projetos passados da organização que utiliza o método e F é dado por (2-25) (Barry Boehm, Abts, et al., 2000; Barry Boehm, Clark, et al., 2000; Barry Boehm et al., 1995; Khatibi e Jawawi, 2011).

$$F = D + 0.2 \times (E - B) \quad (2\text{-}25)$$

Onde D é um coeficiente cujo valor nominal segundo o método é 0.28 ou pode ser calibrado de acordo com projetos passados da organização que utiliza o método, B é dado por (2-22) e o esforço de desenvolvimento de *software* E é dado por (2-21).

Todos os métodos COCOMO recorrem a um vasto leque de parâmetros ao estimar o custo de um projeto. Apesar de serem dos métodos mais populares e com resultados mais claros, são necessários muitos dados para estimar o esforço e muitas vezes os modelos são apresentados como caixa preta para o utilizador. O uso dos métodos COCOMO requer requisitos claros e bem definidos, o que os torna desadequados a projetos em que estes não são claros e a projetos propícios a alterações levando a erros nas estimativas (Basha e Ponnurangam, 2010; Munialo e Muketha, 2016).

2.2 Modelos Não Algorítmicos

Ao contrário dos modelos algorítmicos, os modelos não algorítmicos não utilizam fórmulas matemáticas para calcular a estimativa do custo do *software* (Kumari e Pushkar, 2013b; Waghmode e Kolhe, 2014), em vez disso, estes modelos recorrem a comparações analíticas, inferências e deduções (Khatibi e Jawawi, 2011; Munialo e Muketha, 2016; Shekhar e Kumar, 2016).

Os modelos não algorítmicos são geralmente baseados na experiência e são úteis quando não existem dados empíricos disponíveis (Cuadrado-Gallego, Rodríguez-Soria e Martín-Herrera, 2010). Dependem principalmente do conhecimento adquirido na implementação de projetos anteriores (Bingamawa e Massila Kamalrudin, 2016; Cuadrado-Gallego, Rodríguez-Soria e Martín-Herrera, 2010; Nerkar e Yawalkar, 2014; J. Sharma e Singh, 2017). Por esta razão, geralmente são necessárias algumas informações sobre os projetos realizados no passado (Khatibi e Jawawi, 2011; Shekhar e Kumar, 2016) pois o processo de estimativa é realizado de acordo com a análise de estes dados (Khatibi e Jawawi, 2011; J. Sharma e Singh, 2017; Shekhar e Kumar, 2016).

Um dos problemas dos modelos não algorítmicos é que estes são baseados em opiniões subjetivas. Mesmo que estas sejam provenientes de especialistas, irão ser sempre subjetivas, isto é, não podem ser provadas cientificamente e, normalmente, não há como as corrigir se estiverem erradas. Anos de experiência podem não estar necessariamente relacionados com altos níveis de exatidão nas estimativas (Cuadrado-Gallego, Rodríguez-Soria e Martín-Herrera, 2010). Apesar da sua natureza subjetiva os modelos não algorítmicos são geralmente fáceis de usar e estão entre alguns dos métodos mais utilizados para a estimativa de esforço de desenvolvimento de *software* (Moharreri et al., 2016).

2.2.1 Expert Judgment

Desde o nascimento da indústria do *software* que a tarefa de estimar o esforço de desenvolvimento de *software* recai sobre especialistas ou engenheiros seniores. Isto faz com que a avaliação de especialistas (*Expert Judgment*) seja não só um dos métodos mais antigos, mas também um dos métodos mais utilizados (Alkoffash, Bawaneh e Rabea, 2008; Borade e Khalkar, 2013; Gandomani, Koh e Binhamid, 2014; Kumari e Pushkar, 2013b; Moharreri et al., 2016; Munialo e Muketha, 2016; N. Sharma, Bajpai e Litoriya, 2012; Shekhar e Kumar, 2016).

A avaliação de especialistas é baseada na experiência de especialistas em projetos de *software* semelhantes. A sua experiencia em contextos semelhantes serve de guia para a estimativa do esforço necessário para completar um novo projeto de *software*. A avaliação de especialistas é especialmente útil quando existe um défice de dados empíricos disponíveis ou dificuldade na recolha de requisitos concretos (Alkoffash, Bawaneh e Rabea, 2008; Borade e Khalkar, 2013; Gandomani, Koh e Binhamid, 2014; Khatibi e Jawawi, 2011; Leung e Fan, 2002; Munialo e Muketha, 2016; Nerkar e Yawalkar, 2014; J. Sharma e Singh, 2017; Shekhar e Kumar, 2016).

Geralmente a opinião de especialistas é apenas válida na organização para a qual trabalham pois esta também depende fortemente da experiencia da equipa de desenvolvimento, da complexidade do projeto e do ambiente organizacional. Para além disto, como a avaliação de especialistas é subjetiva esta pode ser tendenciosa e está sempre sujeita ao erro humano. Contudo, a avaliação de especialistas é extremamente útil para projetos pequenos ou médios, especialmente em organizações que possuem equipas e domínios de projetos que não sofram mudanças significativas em relação aos projetos anteriores (Gandomani, Koh e Binhamid, 2014; Munialo e Muketha, 2016).

2.2.2 Analogy

A estimativa de esforço de desenvolvimento de *software* baseada em analogias (*Analogy*) baseia-se no princípio de que os valores reais alcançados dentro de uma organização em projetos anteriores semelhantes são os melhores indicadores e preveem o desempenho de futuros projetos muito melhor do que uma estimativa desenvolvida a partir do zero (Chemuturi, 2011; N. Sharma, Bajpai e Litoriya, 2012).

Neste método, vários projetos semelhantes concluídos anteriormente são analisados e a estimativa de esforço e custo é realizada de acordo com o seu esforço real. A estimativa baseada em analogias é realizada tanto ao nível total do sistema como também ao nível de subsistemas deste. O nível total do projeto tem a vantagem de que todos os componentes do sistema serão considerados enquanto o nível dos subsistemas tem a vantagem de fornecer uma avaliação mais detalhada das semelhanças e diferenças entre o novo projeto e os projetos concluídos anteriormente. Ao analisar os resultados de projetos concluídos anteriormente, os dados destes podem ser extrapolados de forma a estimar o esforço necessário para novos projetos (Alkoffash,

Bawaneh e Rabea, 2008; Khatibi e Jawawi, 2011; Kumari e Pushkar, 2013b; Leung e Fan, 2002; Munialo e Muketha, 2016; Nerkar e Yawalkar, 2014; J. Sharma e Singh, 2017; N. Sharma, Bajpai e Litoriya, 2012; Shekhar e Kumar, 2016).

O processo de prever estimativas de esforço de desenvolvimento de *software* para um novo projeto pode ser dividido nas seguintes etapas:

1. Seleção de projetos para as analogias;
2. Explorar semelhanças e diferenças entre os projetos completos e o projeto a estimar;
3. Avaliar a qualidade de cada analogia;
4. Determinar uma estimativa.

Uma vantagem da estimativa de esforço por analogias é que esta é baseada em valores reais e a estimativa pode ser feita na ausência de um especialista. No entanto, não leva em consideração outros fatores de custo relevantes nos projetos anteriores, como o ambiente e as requisitos funcionais que podem diferir com os do novo projeto (Leung e Fan, 2002; Munialo e Muketha, 2016). Portanto, o aspeto mais importante para o sucesso da estimativa de esforço por analogias é a seleção do conjunto certo de projetos anteriores. Deve ser determinada a semelhança e consequentemente a confiança que pode ser depositada nas analogias (N. Sharma, Bajpai e Litoriya, 2012). Além disto, para este método ser utilizado deve existir um conjunto relativamente grande de informações sobre projetos semelhantes concluídos anteriormente o que pode nem sempre ser possível (Leung e Fan, 2002; Munialo e Muketha, 2016).

2.2.3 Price-to-win

No método *Price-to-win* (traduzido literalmente para “Preço para ganhar”), a estimativa de esforço de desenvolvimento de *software* é baseada inteiramente no orçamento do cliente em vez das funcionalidades do *software*. Neste método, apenas é considerado o orçamento do cliente, não quantos recursos são necessários para desenvolver o *software* nem quaisquer outros fatores para estimação. O custo total do *software* é acordado com base numa proposta com o orçamento do cliente e o desenvolvimento de *software* é restringido a esse custo. Por exemplo, se uma estimativa razoável para um projeto for de 100 meses de esforço, mas o orçamento do cliente só conseguir cobrir 60 meses de esforço, a estimativa é ajustada aos 60 meses de esforço para ganhar o projeto (Bingamawa e Massila Kamalrudin, 2016; Borade e Khalkar, 2013; Kumari e Pushkar, 2013b; Leung e Fan, 2002; Moharreri et al., 2016; Munialo e Muketha, 2016; Nerkar e Yawalkar, 2014; Shekhar e Kumar, 2016).

Apesar do *Price-to-win* ser apelidado de método de estimativa alguns autores acreditam que este método dificilmente pode ser considerado um método de estimativa de esforço de desenvolvimento de *software*. Além disto a maioria dos gestores não irá reportar a sua utilização como um método de estimativa dentro das suas organizações (Bingamawa e Massila Kamalrudin, 2016; Heemstra, 1992; Molokken e Jorgensen, 2003).

O *Price-to-win* ajuda imensamente a obter contratos, porém a sua utilização não é uma boa prática uma vez que é extremamente provável que origine atrasos na entrega do produto final ou obrigue a equipa de desenvolvimento a trabalhar horas extra (Bingamawa e Massila Kamalrudin, 2016; Borade e Khalkar, 2013; Kumari e Pushkar, 2013b; Leung e Fan, 2002; Moharreri et al., 2016; Munialo e Muketha, 2016; Nerkar e Yawalkar, 2014).

2.2.4 Bottom-up e Top-down

Na abordagem *bottom-up*, o esforço de cada componente é estimado pela pessoa que será responsável pelo desenvolvimento do mesmo. Os custos individuais estimados são depois somados de modo a obter a estimativa global do esforço do projeto (Bingamawa e Massila Kamalrudin, 2016; Borade e Khalkar, 2013; Heemstra, 1992; Leung e Fan, 2002; Munialo e Muketha, 2016; N. Sharma, Bajpai e Litoriya, 2012; Shekhar e Kumar, 2016). Esta abordagem visa construir a estimativa do projeto a partir do conhecimento acumulado sobre cada componente do *software* e suas interações (Kumari e Pushkar, 2013b; Moharreri et al., 2016; N. Sharma, Bajpai e Litoriya, 2012).

Para que esta abordagem possa ser aplicada à estimativa de esforço de desenvolvimento de *software* o projeto a estimar deve-se encontrar decomposto nos seus respetivos componentes (Bingamawa e Massila Kamalrudin, 2016; Borade e Khalkar, 2013; Leung e Fan, 2002; Munialo e Muketha, 2016). Muitas vezes, é difícil realizar uma estimativa *bottom-up* no início do ciclo de vida de um projeto pois a informação necessária pode ainda não estar disponível. Esta abordagem tende a ser mais demorada e pode não ser viável quando tempo ou recursos são limitados (N. Sharma, Bajpai e Litoriya, 2012; Shekhar e Kumar, 2016).

A abordagem *top-down* é o oposto da abordagem *bottom-up*. Nesta abordagem, a estimativa do projeto é derivada das características globais do mesmo. Posteriormente, o esforço total estimado é então dividido entre os vários componentes (Bingamawa e Massila Kamalrudin, 2016; Heemstra, 1992; Leung e Fan, 2002; Moharreri et al., 2016; Munialo e Muketha, 2016; Shekhar e Kumar, 2016). Esta abordagem é mais adequada para estimativas de esforço realizadas no inicio do ciclo de vida de um projeto quando somente as características globais deste são conhecidas. A adoção desta abordagem na fase inicial do desenvolvimento de *software* é muito útil pois ainda não estão disponíveis informações detalhadas sobre o mesmo (Bingamawa e Massila Kamalrudin, 2016; Borade e Khalkar, 2013; Kumari e Pushkar, 2013b; N. Sharma, Bajpai e Litoriya, 2012; Shekhar e Kumar, 2016).

Essa abordagem considera também atividades relativas ao sistema como um todo (integração, documentação, gestão de projeto, etc.), muitas das quais podem ser ignoradas em outros métodos de estimativa. A abordagem *top-down* é geralmente mais rápida, mais fácil de implementar e requer detalhes mínimos sobre o projeto. No entanto, pode ser menos precisa, tende a ignorar componentes de nível inferior e possíveis problemas técnicos e também dispõe

de pouco detalhe para justificar decisões ou estimativas (N. Sharma, Bajpai e Litoriya, 2012; Shekhar e Kumar, 2016).

2.2.5 Wideband Delphi

A técnica *Delphi* foi desenvolvida no final da década de 1940, na Rand Corporation, com o objetivo de fazer previsões sobre eventos futuros. Mais recentemente, tem sido utilizada como um meio de orientar um grupo de indivíduos informados para um consenso de opinião sobre determinada questão. O grande objetivo desta técnica é reduzir o intervalo de estimativas para um valor médio razoável (Figura 2-6). Enquanto a técnica *Delphi* original evitava discussões em grupo, uma variação desta técnica, conhecida como a técnica *Wideband Delphi*, permite discussões em grupo entre as rodadas de avaliação (Cuadrado-Gallego, Rodríguez-Soria e Martín-Herrera, 2010). Desde então, foi adaptada para a utilização em diversas indústrias para estimar vastos tipos de tarefas, desde os resultados da recolha de dados estatísticos até as previsões de vendas e marketing. Provou ser uma técnica de estimativa muito eficaz (Stellman e Greene, 2005). A *Wideband Delphi* foi introduzida na engenharia de *software* por Barry Boehm e John Farquhar na década de 1970 (Gandomani, Koh e Binhamid, 2014; Munialo e Muketha, 2016).

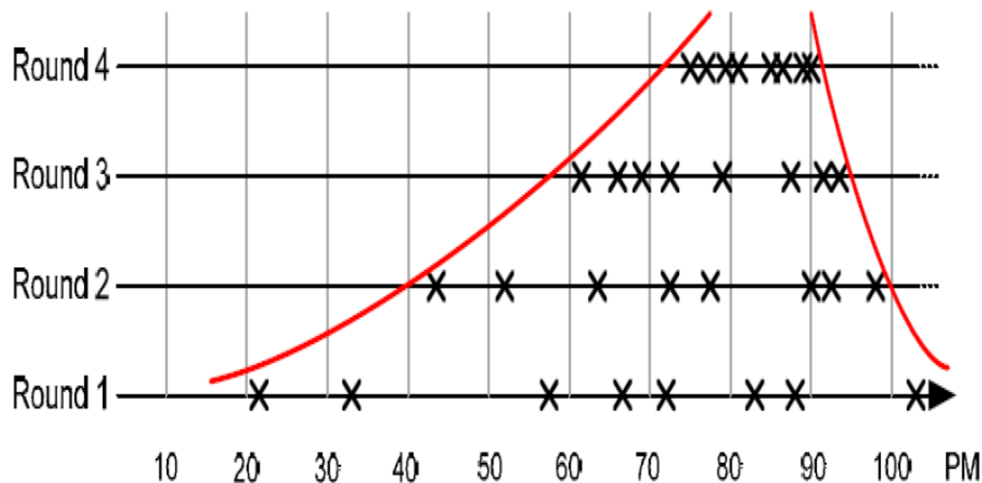

Figura 2-6: Exemplo de utilização da técnica *Wideband Delphi*
Fonte: Software Cost Estimation Methods: A Review (Khatibi e Jawawi, 2011)

Os participantes incluem não só membros da equipa que irá desenvolver o produto, mas também representantes do cliente. Cada membro estima cada tarefa sem consultar com os restantes elementos do grupo. Os membros com estimativas altas ou baixas são convidados a justificar suas estimativas perante o grupo. Depois de discutir as estimativas com o grupo, cada membro revê a sua estimativa. O ciclo repete-se até que todos os elementos do grupo concordem com as estimativas. Finalmente, o coordenador recolhe as estimativas do grupo e compila as tarefas e as suas respetivas estimativas numa única lista final de tarefas (Alkoffash, Bawaneh e Rabea,

2008; Cuadrado-Gallego, Rodríguez-Soria e Martín-Herrera, 2010; Gandomani, Koh e Binhamid, 2014; Khatibi e Jawawi, 2011; Kumari e Pushkar, 2013b; Munialo e Muketha, 2016; Nerkar e Yawalkar, 2014; N. Sharma, Bajpai e Litoriya, 2012; Stellman e Greene, 2005).

A discussão entre a equipa é uma parte muito importante do processo *Delphi*. Desta discussão tipicamente resulta a descoberta de novas prioridades, suposições e tarefas importantes do projeto (que eram desconhecidas até ao momento). A equipa fica muito mais familiarizada com o trabalho que estão prestes a realizar após completarem o processo *Wideband Delphi* (Stellman e Greene, 2005).

A técnica *Wideband Delphi* funciona principalmente porque exige que a equipa se auto corrija de uma forma que ajude a evitar erros e estimativas imprecisas. Embora a estimativa de *software* seja certamente uma competência que melhora com a experiência, o problema mais comum com estimativas é simplesmente que o indivíduo que realiza a estimativa não compreende completamente o que está a estimar. *Wideband Delphi* combate este problema através da discussão de suposições e da obtenção de consenso entre todos os elementos do grupo que está a realizar as estimativas (Stellman e Greene, 2005). Apesar disto, *Wideband Delphi* continua a depender da experiência dos membros do grupo e do consenso entre estes, sendo assim um método pouco apropriado para um projeto que não é familiar aos elementos do grupo. Embora as estimativas *Wideband Delphi* sejam baseadas no consenso de todos os elementos, continuam a ser subjetivas e consequentemente podem ser tendenciosas (otimistas ou pessimistas) (Munialo e Muketha, 2016).

A técnica *Wideband Delphi* já foi utilizada em vários estudos e atividades de estimativa de esforço e custos e tem sido bem-sucedida em combinar as vantagens da técnica de reunião em grupo e as vantagens da estimativa anônima da técnica *Delphi* original (Kumari e Pushkar, 2013b; N. Sharma, Bajpai e Litoriya, 2012).

2.3 Sumário

A combinação de diversos contextos, situações e ambientes muitas vezes determina o modelo de estimativa de esforço de *software* mais apropriado. Num contexto em que é imperativo que a estimativa comporte um elevado grau de precisão deve ser utilizado um modelo mais preciso, porém se o mais importante for ganhar um contrato então provavelmente o modelo *Price-to-win* será o mais indicado. Enquanto projetos de pequena dimensão podem ser facilmente estimados recorrendo à avaliação de um especialista, quando a dimensão dos projetos começa a aumentar começa a ser necessário recorrer a um método de estimativa mais detalhado, como a analogia ou o COMOMO (Munialo e Muketha, 2016).

Os modelos não algorítmicos são geralmente fáceis de aprender e implementar devido à forte ligação à subjetividade humana. Por outro lado, os modelos algorítmicos baseiam-se largamente

em análises estatísticas e fórmulas matemáticas. São geralmente difíceis de aprender e implementar e necessitam de uma quantidade de dados sobre o projeto a ser estimado muito mais elevada. Apesar disto, estes modelos conseguem ser muito uteis quando utilizados corretamente, isto é, quando são corretamente calibrados com base em dados históricos referentes a projetos anteriores da organização que os utiliza. Os métodos algorítmicos são geralmente complementares entre si, por exemplo, o COCOMO utiliza SLOC e FP como duas métricas de entrada e geralmente se essas duas métricas são precisas, o COCOMO também irá apresentar resultados precisos (Cuadrado-Gallego, Rodríguez-Soria e Martín-Herrera, 2010; Kumari e Pushkar, 2013b; N. Sharma, Bajpai e Litoriya, 2012).

Vários fatores de custo devem ser considerados para a estimativa do esforço e do custo do desenvolvimento de *software*. O fator de custo mais comum entre todos os métodos de estimativa é o tamanho do *software*. O esforço, e consequentemente o custo, podem ser estimados diretamente ao estimar o tamanho do *software* utilizando uma das métricas de tamanho do *software*, como SLOC ou FP. O tamanho é também utilizado em conjunto com outros fatores para estimar o esforço de desenvolvimento de *software* na maioria dos métodos de estimação algorítmica (Munialo e Muketha, 2016).

Cada método de estimativa de esforço de desenvolvimento de *software* tem vantagens e desvantagens com base nas capacidades do mesmo. Nenhum método de estimativa de esforço de desenvolvimento de *software* pode ser considerado melhor ou pior do que qualquer outro, cada um tem os seus pontos fortes e fracos que se complementam entre si. Cada método de estimativa surgiu com o objetivo de ser utilizado num tipo de projeto e método de desenvolvimento especifico. Métodos de estimativa, tais como FPA e COCOMO, são adequados para o desenvolvimento de *software* em que os requisitos são totalmente conhecidos. Por outro lado, esses métodos são desafiados quando os requisitos estão em constante mudança, como em ambientes ágeis (Cuadrado-Gallego, Rodríguez-Soria e Martín-Herrera, 2010; Munialo e Muketha, 2016; N. Sharma, Bajpai e Litoriya, 2012).

3. Estimativa de Esforço de Software em Ambientes Ágeis

As metodologias ágeis são um grupo de metodologias de desenvolvimento de *software* baseados em desenvolvimento iterativo e incremental, onde os requisitos e as soluções evoluem através da colaboração entre equipas auto-organizadas e multifuncionais. Promovem o planeamento adaptativo, o desenvolvimento e entrega iterativa, incentivam a resposta rápida e são flexíveis à mudança (Cao, 2008; Cohn, 2005; Litoriya e Kothari, 2013; Munialo e Muketha, 2016; Usman, 2015; Z. K. Zia, Tipu e Zia, 2012). O Manifesto Ágil (Beck et al., 2001) introduziu o termo em 2001.

No desenvolvimento ágil, um projeto pode consistir em uma ou mais iterações, cada uma com múltiplas *user stories* (US). Por vezes múltiplas iterações são agrupadas formando uma *release*. As iterações são curtos períodos de tempo, utilizados pelas metodologias ágeis, nos quais é dividido o trabalho do projeto. Cada iteração tem geralmente uma duração fixa de 1 semana a 4 semanas. Uma *user story* é a definição de alto nível de um requisito na qual constam informações suficientes para que a equipa de desenvolvimento possa produzir uma estimativa do esforço necessário para o seu desenvolvimento. Cada *user story* é tipicamente formulada em uma ou duas frases escritas na linguagem do cliente (Cohn, 2004, 2005, 2010; Kang, Choi e Baik, 2010; Osman e Musa, 2016).

No desenvolvimento de *software* clássico, a capacidade de carga de trabalho de um membro da equipa é determinada pelo gestor da mesma que estima quanto tempo determinadas tarefas levam e, em seguida, atribui o trabalho com base no tempo total disponível desse membro da equipa. As metodologias ágeis seguem uma abordagem consideravelmente diferente para determinar a capacidade de um membro da equipa. Em primeiro lugar, atribuem o trabalho a toda a equipa, não a um indivíduo. Filosoficamente, isto coloca ênfase no esforço coletivo. Em

segundo lugar, recusam-se a quantificar o trabalho em termos de tempo, porque isso seria prejudicial para a auto-organização das equipas. Isto é uma mudança drástica do desenvolvimento de *software* clássico. Em vez do gestor de equipa estimar o tempo em nome de outros indivíduos, os membros da equipa utilizam complexidade e grau de dificuldade para estimar seu próprio trabalho (Cohn, 2005; Litoriya e Kothari, 2013; Munialo e Muketha, 2016; Z. K. Zia, Tipu e Zia, 2012).

Num projeto ágil, é frequente iniciar uma iteração com requisitos incompletamente especificados, cujos detalhes só serão descobertos durante a iteração. As estimativas são assim muito importantes, pois servem de base para o planeamento do projeto em termos de priorização de *user stories* e alocação recursos da equipa de desenvolvimento (Osman e Musa, 2016).

As metodologias ágeis não indicam uma forma de as equipas estimarem seu trabalho. No entanto, recomendam que as equipas não estimem o esforço em termos de tempo, mas, em vez disso, que utilizem uma métrica mais abstrata para o quantificar. Alguns métodos comuns de estimativa incluem um tamanho numérico, tamanhos de t-shirt ou a sequência de Fibonacci. O importante é que a equipa compartilhe um entendimento comum da escala que é utilizada, para que cada membro da equipa esteja confortável com os valores da mesma (Cohn, 2005; Z. K. Zia, Tipu e Zia, 2012).

Os métodos de estimativa de esforço de desenvolvimento de *software* mais comuns em ambiente ágeis são a avaliação de especialistas (*Expert Judgment*), a estimativa baseada em analogias (*Analogy*) e a desagregação (*Disaggregation*) (Cohn, 2005; Usman, 2015). A avaliação de especialistas e a estimativa baseada em analogias já foram apresentadas nas secções 2.2.1 e 2.2.2 respetivamente. A desagregação refere-se à divisão de *user stories* ou funcionalidades em fragmentos mais pequenos e consequentemente mais fáceis de estimar. Por exemplo, num projeto em que a maioria das *user stories* contem com um esforço estimado de 2 a 5 dias será muito difícil estimar uma *user story* de 100 dias. Não só é extremamente difícil estimar itens muito grandes como também não haverá *user stories* equiparáveis a esta para comparações (Cohn, 2005).

Cada um destes métodos pode ser utilizado separadamente, porém os melhores resultados são obtidos quando são utilizadas em conjunto recorrendo a um método como o *Planning Poker* (Cohn, 2005; Usman, 2015).

3.1 Story Points

Os *story points* (SP) são uma unidade de medida relativa utilizada para expressar o esforço total necessário para realizar uma determinada tarefa. Na estimativa de esforço utilizando *story points* é atribuído um único valor a cada *user story*. Individualmente este valor não tem muita importância, o que importa realmente é a relação entre os vários valores de todas as estimativas

(Cohn, 2005; Santana et al., 2011). Por exemplo, uma *user story* com uma estimativa de dois *story points* irá representar sensivelmente o dobro do esforço do que uma *user story* com uma estimativa de um *story point* (Coelho e Basu, 2012).

Não existe uma fórmula formal para calcular o número de *story points* de uma *determinada user story*. A estimativa dos *story points* de uma determinada *user story* é baseada na quantidade de trabalho, na complexidade do trabalho e nos riscos e incertezas inerentes a este (Coelho e Basu, 2012; Cohn, 2005; Osman e Musa, 2016; Santana et al., 2011). Uma caraterística dos *story points*, que reforça a ideia de estes são relativos, é que cada equipa define um *story point* como achar melhor. Uma equipa pode definir um *story point* como sendo representativo de um dia ideal de trabalho (ou seja, um dia sem interrupções). Outra equipa pode definir um s*tory point* como sendo representativo de uma semana ideal de trabalho e ainda uma terceira equipa pode definir um *story point* como sendo uma qualquer medida de complexidade. Todas estas definições estão corretas desde que toda a equipa compreenda a definição utilizada (Cohn, 2004). Tipicamente existem duas abordagens para começar a estimar utilizando *story points*. A primeira abordagem consiste em escolher uma das *user stories* mais pequenas e atribuir-lhe a estimativa de um *story point*. Por outro lado, a segunda abordagem consiste em escolher uma *user story* de tamanho médio e atribuir-lhe um valor sensivelmente a meio da escala utilizada (Cohn, 2005; Osman e Musa, 2016; Santana et al., 2011).

No fim de cada iteração a equipa soma o número de *story points* das *user stories* que foram completamente desenvolvidas. Este valor, denominado de velocidade (*velocity*), pode posteriormente ser utilizado para prever quantos *story points* irão ser desenvolvidos nas próximas iterações, porém, devido à natureza relativa dos *story points*, não pode ser utilizado como uma medida de produtividade nem comparado entre equipas. (Cohn, 2004; Santana et al., 2011).

3.1.1 Story Points vs Tempo

A existência de uma relação entre *story points* e tempo é relativamente óbvia. A principal razão para se realizarem estimativas é para poder prever quantas e quais as funcionalidades que podem ser entregues ao cliente e acima de tudo, quando é que isto será possível. Se o objetivo das estimativas é determinar o "quando", então estas possuem uma perspetiva temporal. É então necessário estimar o tempo, mais precisamente, o esforço (normalmente expresso em dias ou horas de trabalho), necessário para realizar determinada tarefa. Porém, apesar de existir uma relação entre *story points* e tempo, uma afirmação como "Um *story point* = Oito horas" está incorreta. Como os *story points* são uma unidade de medida relativa, a sua relação com o tempo não pode ser formalizada desta forma (Cohn, 2005, 2014b, 2014a).

Ao produzir afirmações deste tipo está a ser ignorada a principal razão de utilizar *story points*. Os *story points* são uteis pois permitem que elementos da equipa de desenvolvimento que

trabalhem a velocidades diferentes possam comunicar e estimar em conjunto de uma forma coerente (Cohn, 2005, 2014a).

Dois elementos da equipa de desenvolvimento podem ambos estimar uma determinada *user story* em um *story point*, mesmo que as suas estimativas individuais do tempo que esta irá levar a implementar sejam diferentes. Com base nesta estimativa conseguem ambos estimar outra *user story* em dois *story points* se ambos concordarem que irá levar o dobro do esforço da primeira a implementar (Cohn, 2005, 2014a).

Se for criada uma relação explícita, que torna a conversão entre *story points* e tempo possível, todos os benefícios de estimar utilizando uma unidade de medida relativa e abstrata, como os *story points*, são perdidos. A equipa de desenvolvimento irá passar a pensar no tempo que estimam que as *user stories* demorem a implementar e posteriormente irão efetuar a conversão para os respetivos *story points*. Por exemplo, uma *user story* estimada em 16 horas irá ser convertida em uma estimativa de 2 *story points*. Esta é uma abordagem errada (Cohn, 2005, 2014a).

Os elementos da equipa desenvolvimento devem considerar o esforço necessário para a implementação de determinada *user story* em relação a outras *user stories*. Por exemplo, se dois elementos da equipa de desenvolvimento concordarem que determinada *user story* representa o dobro do esforço de outra *user story*, então esta deve ser estimada no dobro dos *story points*. Mesmo que o primeiro elemento estime que irá demorar 5 horas e o segundo elemento estime que irá demorar 10 horas, desde que o esforço seja o dobro de outra *user story* com uma estimativa de 1 *story point*, ambos podem concordar que esta deverá ser estimada em 2 *story points*. Desta forma os *story points* não possuem uma relação explícita com o tempo (esforço) (Cohn, 2005, 2014a).

A Figura 3-1 é uma representação genérica da distribuição da probabilidade implementar totalmente uma *user story* estimada em um *story point* ao longo do tempo. Pode observar-se que o tempo médio necessário para implementar um *story point* é x horas. No entanto, também mostra que algumas *user stories* podem demorar menos e outras podem demorar mais do que x horas. Até a equipa identificar todo o trabalho necessário para a implementação da *user story*, é muito difícil saber em que parte da curva esta se encontra (Cohn, 2005).

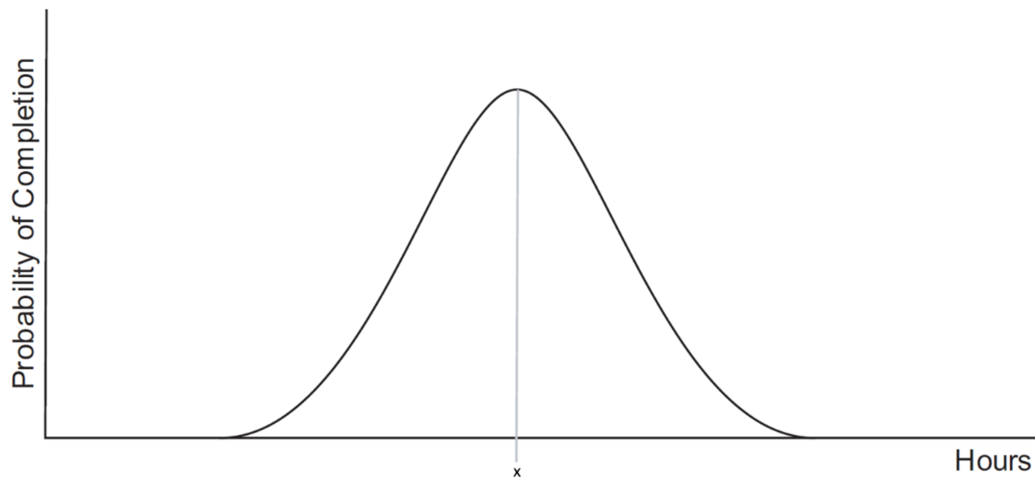

Figura 3-1: Distribuição do tempo correspondente a um *story point*
Fonte: Agile estimating and planning (Cohn, 2005)

A Figura 3-2 é uma representação genérica da distribuição da probabilidade implementar totalmente *user stories* estimadas em um, dois e três *story points* ao longo do tempo. Pode observar-se que o tempo médio necessário para implementar um *story point* continua a ser de x horas. No entanto, mostra também que podem existir casos em que uma *user story* estimada em um *story point* demora o mesmo ou mais tempo a implementar que uma *user story* estimada em dois *story point*. O mesmo acontece com *user stories* estimadas em dois e três *story points* e assim sucessivamente. Isto é uma situação normal e não será um problema pois o tempo médio necessário para implementar todas as *user stories* do projeto irá eliminar estas variações (Cohn, 2005).

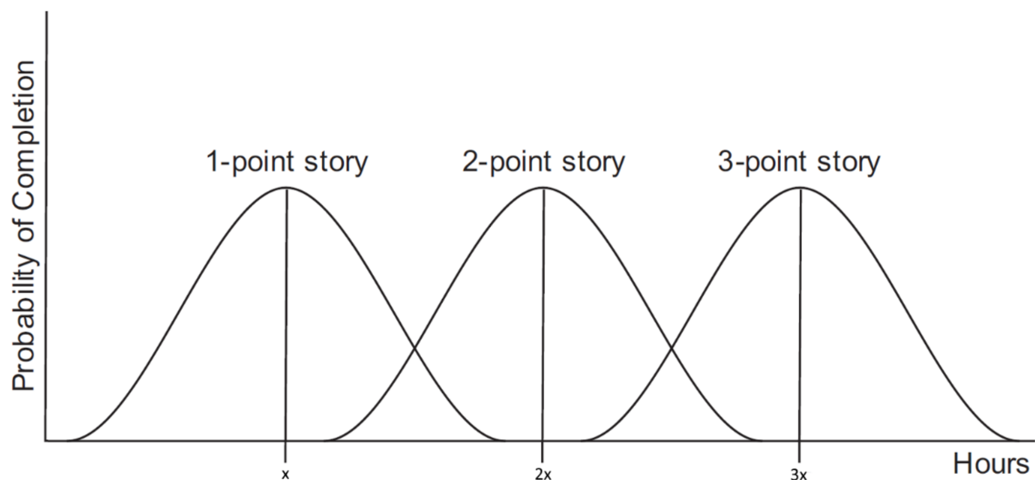

Figura 3-2: Distribuição do tempo correspondente a um, dois e três *story points*
Fonte: Agile estimating and planning (Cohn, 2005)

3.2 Ideal Days

Num projeto de desenvolvimento de *software* o tempo ideal (*ideal time*) difere do tempo decorrido devido à existência de atividades que não são diretamente contabilizadas nas atividades do projeto. Todos os dias, além do tempo gasto nas atividades planeadas do projeto, todos os elementos da equipa gastam tempo realizando outras tarefas, como responder a *e-mails* ou em chamadas ou reuniões com a restante equipa ou com o cliente (Cohn, 2005; Osman e Musa, 2016).

O tempo ideal é quantidade de tempo que determinada tarefa demora a realizar caso todo o foco de trabalho estivesse nessa tarefa e não existissem interrupções. O tempo decorrido, por outro lado, é a quantidade de tempo que passa num relógio. É quase sempre mais fácil e preciso estimar a duração de um evento em tempo ideal do que no tempo decorrido (Cohn, 2005; Osman e Musa, 2016).

A realização de uma estimativa em dias decorridos exige que sejam consideradas todas as interrupções que possam ocorrer ao trabalhar numa *user story*. Se para a estimativa forem considerados os dias ideais que uma *user story* levará a desenvolver, testar e aceitar, não é necessário considerar interrupções ou atividades secundárias referentes ao ambiente em que a equipe trabalha. Desta forma, os dias ideais são uma estimativa de tamanho, embora menos rigorosa do que *story points*. Ao estimar em dias ideais, deve apenas ser associada uma única estimativa a cada *user story*. Ao invés de estimar que uma *user story* levará quatro dias a desenvolver e dois dias a testar, estes valores devem ser somados, passando a *user story* como um todo a ter uma estimativa de seis dias ideais (Cohn, 2005; Osman e Musa, 2016).

Existem algumas diferenças entre *story points* e dias ideais. Estimar com *story points* é geralmente mais rápido do que estimar em dias ideais e, ao contrário dos dias ideais, os *story points* podem ser comparados entre os elementos da equipa. Por outro lado, os dias ideais são mais facilmente explicados a pessoas externas ao projeto, mais fáceis de começar a utilizar e é mais fácil prever a velocidade inicial (Cohn, 2005; Osman e Musa, 2016).

3.3 Planning Poker

Várias técnicas têm sido utilizadas para realizar a estimativa de esforço de desenvolvimento de *software* em ambientes ágeis. Uma das mais recentes e populares é o *Planning Poker*, introduzido por James Grenning em 2002 (Grenning, 2002) e mais tarde popularizado por Mike Cohn (Cohn, 2005). O *Planning Poker* é o método mais frequentemente utilizado em ambientes ágeis. Este combina a opinião de especialistas de diferentes áreas de desenvolvimento de *software*, a analogia entre itens e ainda torna a tarefa de obter estimativas agradável o que resulta em estimativas rápidas, mas confiáveis (Osman e Musa, 2016).

Planning Poker, tal como *Wideband Delphi*, é uma técnica de estimativa de esforço de desenvolvimento de *software* baseado na colaboração e no consenso da equipa. No desenvolvimento ágil, as sessões de *Planning Poker* são tipicamente realizadas no início de cada iteração e contam com a presença de toda a equipa de desenvolvimento do projeto (Cohn, 2005; Gandomani, Koh e Binhamid, 2014; Munialo e Muketha, 2016; Rockefeller e Hamilton, 2017).

Numa sessão de *Planning Poker*, é dado um baralho de cartas a cada elemento da equipa de desenvolvimento. O baralho contem uma sequencia de valores que representam *story points*, dias ideais ou qualquer outra unidade de medida que a equipa esteja a utilizar. Embora qualquer sequência possa ser utilizada, Cohn (Cohn, 2005) recomenda a utilização de uma sequência não linear, como por exemplo a sequência de Fibonacci (0, 1, 2, 3, 5, 8, 13, 21…). Isto deve-se ao facto de os intervalos entre os seus valores aumentarem sucessivamente com a progressão da sequencia, o que reflete um maior grau de incerteza nas estimativas mais elevadas (Calefato e Lanubile, 2011; Cohn, 2005; Gandomani, Koh e Binhamid, 2014; Munialo e Muketha, 2016; Rockefeller e Hamilton, 2017).

A equipa discute cada *user story*, colocando perguntas, conforme necessário. Quando a equipa estiver satisfeita com uma *user story*, cada elemento escolhe uma carta do seu baralho que irá representar a sua estimativa. Todas as cartas são reveladas ao mesmo tempo. Isto minimiza o efeito de ancoragem (*anchoring*), isto é, a tendência de alguns elementos influenciarem em demasia as estimativas do resto da equipa. Se todas as estimativas forem iguais, esse será o valor estimado para a *user story*. Caso contrário, a equipa realiza uma ronda de discussão, onde os elementos com o valor mais baixo e mais alto devem explicar as suas estimativas. Posteriormente, cada elemento reavalia a sua estimativa, escolhe novamente uma carta do seu baralho (pode ser igual ou diferente da sua estimativa anterior) e as cartas são novamente reveladas todas ao mesmo tempo. Este processo é repetido até a equipa chegar a um consenso ou até a equipa decidir que a estimativa desta *user story* deve ser adiada para quando estiverem disponíveis mais informações. Tipicamente quando o resultado da estimativa é superior a 21 na sequência de Fibonacci, deve ser considerado a divisão da mesma em múltiplas *user stories* mais pequenas (Calefato e Lanubile, 2011; Cohn, 2005; Gandomani, Koh e Binhamid, 2014; Munialo e Muketha, 2016; Osman e Musa, 2016; Rockefeller e Hamilton, 2017).

O *Planning Poker* funciona principalmente porque reúne as estimativas de todos os elementos da equipa multifuncional do desenvolvimento ágil. Esta colaboração tende a proporcionar estimativas mais precisas do que as de apenas um especialista. A discussão realizada durante as sessões é também benéfica para toda a equipa pois não só aumenta a sua compreensão do trabalho a realizar, mas também ajuda a identificar *user stories* com informação insuficiente e com um nível de incerteza elevado (Calefato e Lanubile, 2011; Cohn, 2005; Munialo e Muketha, 2016).

3.4 Sumário

O paradigma inerente ao desenvolvimento de *software* num ambiente ágil faz com que a aplicabilidade dos modelos de estimativa de esforço existentes seja limitada. Os requisitos dos projetos desenvolvidos num ambiente ágil não são totalmente conhecidos no arranque do projeto e podem mudar a qualquer altura do seu ciclo de vida. Isto faz com que o funcionamento das equipas seja muito diferente do que é normal quando o projeto é desenvolvido utilizando uma metodologia tradicional. As equipas ágeis assumem coletivamente o trabalho a realizar e quantificam-no recorrendo a uma unidade de medida abstrata (Cohn, 2005; Litoriya e Kothari, 2013; Munialo e Muketha, 2016; Z. K. Zia, Tipu e Zia, 2012).

A métrica mais utilizada em ambientes ágeis são os *story points* (Usman, Mendes e Börstler, 2015). Os *story points* possuem a vantagem de permitir que a equipa se abstraia do aspeto temporal das estimativas, criando assim uma forma de indivíduos com níveis de experiencia e velocidades de trabalho diferentes poderem estimar em conjunto de uma forma coerente (Cohn, 2005, 2014a).

Apesar de existirem três grandes métodos de estimativa de esforço de desenvolvimento de *software* em ambiente ágeis, os melhores resultados são obtidos quando estes são utilizados em conjunto (Cohn, 2005; Usman, 2015). O *Planning Poker* combina os vários aspetos destes métodos, o que faz com que este seja o método de estimativa mais utilizado em ambientes ágeis (Usman, Mendes e Börstler, 2015).

4. Machine Learning

Até meados da década de 90, o foco principal da investigação de estimativas de esforço de desenvolvimento de *software* eram os modelos algorítmicos. A falta de resultados consistentes levou a que investigadores analisassem técnicas alternativas, como modelos não algorítmicos e várias técnicas de *Machine Learning* (ML). Como resultado, durante esta década, as técnicas de *Machine Learning* surgiram como forma alternativa de estimar o esforço de desenvolvimento de *software* (de Barcelos Tronto, da Silva e Sant'Anna, 2008; Cuadrado-Gallego, Rodríguez-Soria e Martín-Herrera, 2010; Humayun e Gang, 2012; Mendes et al., 2003). Estas técnicas já tinham sido aplicadas com sucesso em vários domínios, como medicina, engenharia, geologia e física, geralmente para encontrar soluções para problemas de estimativa ou classificação (de Barcelos Tronto, da Silva e Sant'Anna, 2008).

Machine Learning é uma subárea da Inteligência Artificial cujo principal objetivo é dar aos computadores a capacidade de aprender sem serem explicitamente programados. As suas técnicas tentam melhorar o seu desempenho em determinada tarefa através da experiência. Os algoritmos de ML provaram ser de grande valor prático em uma variedade de domínios de aplicação, sendo particularmente úteis em (Zhang e Tsai, 2002, 2003):

- Domínios de problemas mal compreendidos onde existe pouco conhecimento para que os humanos desenvolvam algoritmos eficazes;
- Domínios em que existem grandes conjuntos de dados que potencialmente contêm relacionamentos implícitos desconhecidos, porém valiosos;
- Domínios em que os algoritmos devem adaptar-se à constante mudança das condições do mesmo.

As técnicas de ML incorporam algumas das facetas da mente humana que nos permitem resolver problemas extremamente complexos a velocidades que superam mesmo os

computadores mais rápidos (Cuadrado-Gallego, Rodríguez-Soria e Martín-Herrera, 2010). Estas técnicas oferecem uma forma de resolver problemas que envolvam muitos fatores. São especialmente boas quando existe uma grande quantidade de dados (de Barcelos Tronto, da Silva e Sant'Anna, 2008). Algumas vantagens das técnicas de ML incluem sua capacidade de modelar um complexo conjunto de relações entre as diversas variáveis em estudo e sua capacidade de aprender com dados históricos (Elish, 2009; Idri, Abran e Kjiri, 2000; Mair et al., 2000).

A engenharia de *software* tem-se revelado como sendo uma área com grande potencial para a aplicação de técnicas de ML. Muitas tarefas de desenvolvimento e manutenção de *software* podem ser formuladas como problemas de aprendizagem e consequentemente podem potencialmente ser resolvidas recorrendo a técnicas de ML (Zhang e Tsai, 2002, 2003). A Figura 4-1 mostra a relação existente entre a engenharia de *software* e ML.

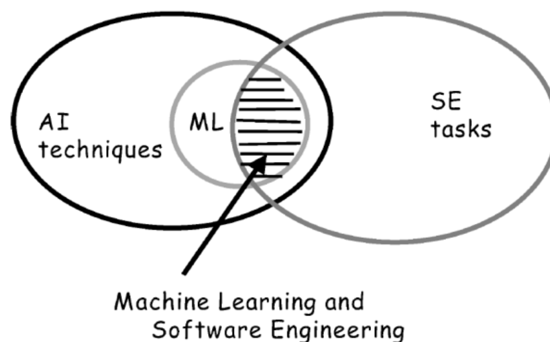

Figura 4-1: Relação entre Engenharia de *Software* e *Machine Learning*
Fonte: Machine Learning and Software Engineering (Zhang e Tsai, 2003)

Machine Learning ainda é uma área relativamente nova, porém, é uma área que investigadores pensam que poderá levar à produção, de uma forma consistente, de estimativas precisas. O sistema efetivamente "aprende" como estimar a partir de um conjunto dados históricos (Alkoffash, Bawaneh e Rabea, 2008). Em comparação a outros domínios em que as técnicas de ML foram aplicadas com sucesso, o domínio das estimativas de esforço de desenvolvimento de *software* coloca muitos desafios, como pequenos conjuntos de dados de treino, informação qualitativa e ainda a dependência de fatores humanos (Wen et al., 2012).

4.1 Aplicações na Estimativa de Esforço de Software

Dada a existência de uma grande variedade de modelos, diversos estudos tentaram determinar qual a melhor abordagem para a estimativas de esforço de desenvolvimento de *software*. Uma vez que existe um vasto número de fatores que pode variar de estudo para estudo, como por exemplo, o conjunto de dados pode apresentar características diferentes (número de variáveis, número de observações, etc.), não é surpreendente que os resultados dos diversos estudos não tenham convergido para respostas semelhantes. Isto leva a que ainda exista uma dúvida para os

profissionais da área quanto aos modelos que devem adotar (de Barcelos Tronto, da Silva e Sant'Anna, 2008; Mendes et al., 2003). Mais recentemente, estudos analisaram a utilização de abordagens baseadas em técnicas de ML como complemento ou alternativa aos modelos algorítmicos e não algorítmicos (de Barcelos Tronto, da Silva e Sant'Anna, 2008; Mendes et al., 2003). Também os resultados destes não convergiram para respostas semelhantes. A divergência dos resultados da maioria dos estudos existentes sobre modelos baseados em técnicas de ML, cujas causas ainda não são totalmente compreendidas, podem impedir que os profissionais da área adotem modelos ML na prática (Wen et al., 2012).

Em 2012, Wen et al. (Wen et al., 2012) selecionaram 84 estudos, publicados de 1991 a 2010, onde foram propostos modelos de estimativas de esforço de desenvolvimento de *software* baseados em ML. A partir dos estudos selecionados foram identificadas oito técnicas de ML já aplicadas nas estimativas de esforço de desenvolvimento de *software*.

- *Case-Based Reasoning* (CBR)
- *Artificial Neural Networks* (ANN)
- *Decision Trees* (DT)
- *Bayesian Networks* (BN)
- *Support Vector Regression* (SVR)
- *Genetic Algorithms* (GA)
- *Genetic Programming* (GP)
- *Association Rules* (AR)

Das técnicas listadas anteriormente, CBR, ANN e DT são as três mais utilizadas. Conjuntamente foram utilizadas em 80% dos estudos selecionados, como pode ser visto na Figura 4-2. É importante referir que alguns estudos contêm mais do que uma técnica (Wen et al., 2012).

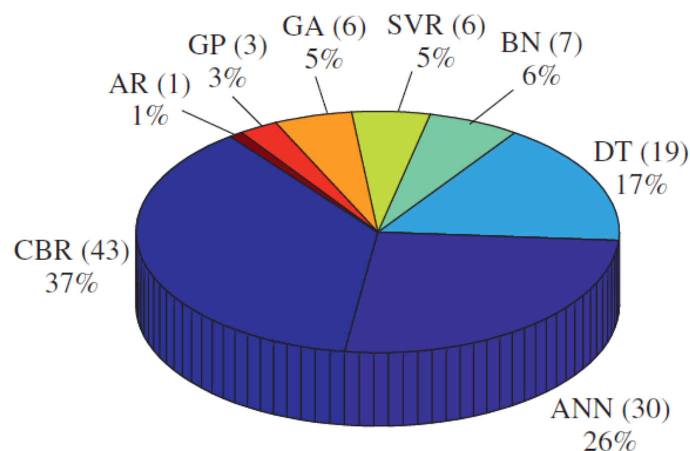

Figura 4-2: Distribuição das técnicas de ML nos estudos selecionados por Wen et al.
Fonte: Systematic literature review of machine learning based software development effort estimation models (Wen et al., 2012)

Os resultados encontrados em (Wen et al., 2012) confiram os resultados encontrados por outros autores em estudos semelhantes em anos anteriores. Por exemplo, em 2003, Zhang e Tsai (Zhang e Tsai, 2003) identificaram a utilização de cinco técnicas de ML (CBR, ANN, DT, BN e GA) e em 2007, Jørgensen e Shepperd (Jørgensen e Shepperd, 2007) identificaram a utilização de onze técnicas, das quais quatro eram técnicas de ML (CBR, ANN, DT, e BN). A popularidade de cada técnica identificada permanece consistente nos três estudos.

Para além das técnicas utilizadas, Wen et al. (Wen et al., 2012) identificaram também os conjuntos de dados históricos (*datasets*), os métodos de validação de resultados, os critérios de avaliação utilizados e os resultados dos estudos selecionados.

A Tabela 4-1 resume os *datasets* mais utilizados e apresenta algumas das suas características. É apresentado o tipo de conjunto de dados (proveniente de apenas de uma organização – *within-company* – ou de várias organizações – *cross-company*), o número e percentagem de estudos que utilizam os mesmos, a quantidade de dados (representada pelo número de projetos) e a sua publicação original (Wen et al., 2012).

Tabela 4-1: *Datasets* utilizados nos estudos selecionados por Wen et al.
Fonte: Systematic literature review of machine learning based software development effort estimation models (Wen et al., 2012)

Dataset	Type	# of Studies	Percent	# of Projects	Source
Desharnais	within-company	24	29	81	(Desharnais, 1989)
COCOMO	cross-company	19	23	63	(B. W. Boehm, 1981)
ISBSG	cross-company	17	20	>1000[a]	(ISBSG, 1997)
Albrecht	within-company	12	14	24	(Allan J. Albrecht e Gaffney, 1983)
Kemerer	within-company	11	13	15	(Kemerer, 1987)
NASA	within-company	9	11	18	(Bailey e Basili, 1981)
Tukutuku	cross-company	7	8	>100[a]	(Mendes, Mosley e Counsell, 2005)

[a] O número de projetos depende da versão do *dataset*

Em relação aos métodos de validação de resultados foram identificados os métodos *Holdout*, *Leave-One-Out Cross-Validation* (LOOCV) e *n-fold Cross-Validation* como sendo os mais utilizados. Estes métodos foram utilizados em 32 (38%), 31 (37%) e 16 (19%) dos estudos selecionados respetivamente (Wen et al., 2012). No que diz respeito a critérios de avaliação foram identificados os critérios *Mean Magnitude of Relative Error* (MMRE), *Percentage Relative Error Deviation* (PRED(x), mais concretamente PRED(25)) e *Median Magnitude of Relative Error* (MdMRE) como sendo os mais populares. Estes critérios foram utilizados em 75 (89%), 55 (65%) e 31 (37%) dos estudos selecionados respetivamente (Wen et al., 2012). Os critérios de avaliação serão explorados em mais detalhe no capítulo 5.

Os resultados dos estudos selecionados por Wen et al. (Wen et al., 2012) estão resumidos na Tabela 4-2. Um MMRE menor ou um PRED(25) mais alto indica uma estimativa mais precisa. Com exceção de BN, o resto das técnicas ML obtiveram resultados em média de 34% a 55% no MMRE e de 46% a 72% no PRED(25).

Tabela 4-2: Resumo dos resultados dos estudos selecionados por Wen et al.
Fonte: Systematic literature review of machine learning based software development effort estimation models (Wen et al., 2012)

	MMRE				PRED(25)			
Model	# of Values	Mean (%)	Min (%)	Max (%)	# of Values	Mean (%)	Min (%)	Max (%)
CBR	57	51	11	119	50	46	5	91
ANN	39	37	7	95	32	64	24	94
DT	17	55	9	156	15	56	8	89
BN	6	106	34	190	5	30	15	42
SVR	11	34	9	72	11	72	34	94
GP	11	49	26	71	14	52	16	94
AR	2	49	30	69	2	57	56	58

Estes resultados indicam que a precisão das estimativas produzidas por modelos baseados em técnicas ML está muito próxima de níveis aceitáveis. Um modelo de estimativa é geralmente considerado aceitável se conseguir resultados de MMRE ≤ 25% e PRED(25) ≥ 75% (Conte, Dunsmore e Shen, 1986; Fedotova, Teixeira e Alvelos, 2013; Wen et al., 2012).

Pode-se observar também que em média a técnicas ANN e SVR superam as restantes, porém, isto não quer dizer que estas sejam a melhor solução para qualquer problema (Wen et al., 2012). Apesar dos resultados obtidos estarem muito próxima de níveis aceitáveis estes não são consistentes, como pode ser visto pela grande variação dos resultados mínimos e máximos na Tabela 4-2.

A vasta maioria dos estudos selecionados por Wen et al. (Wen et al., 2012) são baseados em modelos tradicionais (algorítmicos e não algorítmicos) ou variações destes. Embora estes métodos sejam muitas vezes utilizados para estimar projetos ágeis, levam normalmente a estimativas imprecisas (Z. K. Zia, Tipu e Zia, 2012). Em 2012, Zia et al. (Z. K. Zia, Tipu e Zia, 2012) propuseram um modelo de estimativa de esforço de desenvolvimento de *software* para projetos ágeis (o modelo proposto não utilizava qualquer técnica de Machine Learning). O modelo proposto foi validado utilizando um *dataset* de dados de 21 projetos a partir dos quais foram obtidos resultados muito positivos (Saroha e Sahu, 2015; Z. K. Zia, Tipu e Zia, 2012). Este *dataset* é o único *dataset* encontrado que utilize *story points*.

Nos últimos anos, alguns autores tentaram melhorar os resultados obtidos por Zia et al. aplicando técnicas de *Machine Learning* ao mesmo *dataset*. Por exemplo, em 2015, Panda (Panda, 2015) e Panda et al. (Panda, Satapathy e Rath, 2015) aplicaram técnicas baseadas em

Artificial Neural Networks. Em 2016, Satapathy (Satapathy, 2016) aplicou técnicas baseadas em *Decision Trees* (mais concretamente *Random Forest*) e SVR e em 2017, Satapathy e Rath (Satapathy e Rath, 2017) aplicaram técnicas baseadas em *Decision Trees*, *Random Forest* e *Stochastic Gradient Boosting* (SGB).

Todos estes autores obtiveram resultados geralmente positivos nos seus respetivos estudos, como pode ser visto na Tabela 4-3.

Tabela 4-3: Resumo dos resultados dos estudos baseados em Story Points

Técnica	PRED(25)	Estudo
Artificial Neural Networks (melhor) [a]	96.42	(Panda, 2015)
Artificial Neural Networks (pior) [a]	77.69	(Panda, 2015)
Artificial Neural Networks (melhor) [a]	94.76	(Panda, Satapathy e Rath, 2015)
Artificial Neural Networks (pior) [a]	85.92	(Panda, Satapathy e Rath, 2015)
Random Forest	66.67	(Satapathy, 2016)
SVR (melhor) [a]	80.95	(Satapathy, 2016)
SVR (pior) [a]	38.10	(Satapathy, 2016)
Decision Trees	38.10	(Satapathy e Rath, 2017)
Gradient Boosting	85.71	(Satapathy e Rath, 2017)
Random Forest	66.67	(Satapathy e Rath, 2017)

[a] Foram implementados vários modelos pelos respetivos autores, porém apenas são apresentados os melhores e piores resultados.

Apesar disto, alguns autores (Panda, Satapathy e Rath, 2015; Satapathy, 2016; Satapathy e Rath, 2017) referiram que a quantidade de dados disponíveis, não só no que toca ao número de projetos do *dataset*, mas também ao número de atributos destes, é muito baixo. Referiram também que possuir um valor de velocidade para cada projeto antes de este ser iniciado nem sempre é possível. Estes dois pontos apresentam ameaças à validade dos resultados obtidos e é sugerido que sejam investigados com maior detalhe no futuro.

4.2 Técnicas

Machine Learning está assente na aprendizagem através da observação de um determinado conjunto de dados. A aprendizagem é um domínio muito amplo, o que levou a que *Machine Learning* se ramificasse em vários subdomínios que lidam com diferentes tipos de tarefas de aprendizagem (Abu-Mostafa, Magdon-Ismail e Lin, 2012; Bishop, 2006; Hackeling, 2014; James et al., 2013; Raschka, 2016; Shalev-Shwartz e Ben-David, 2014).

As aplicações em que o conjunto de dados utilizado contem atributos e os respetivos valores alvo são conhecidas por problemas de *Supervised Learning*. O principal objetivo deste tipo de aprendizagem é treinar um modelo utilizando dados com resultados conhecidos para que este consiga fazer previsões sobre dados nunca antes vistos (Bishop, 2006; Hackeling, 2014; James

et al., 2013; Raschka, 2016). Este tipo de tarefas de aprendizagem pode ainda ser divido pelo tipo de variável alvo. Esta divisão levou a uma convenção de nomenclatura para estass tarefas de aprendizagem: regressão, quando se tenta prever resultados quantitativos, e classificação, quando se tenta prever resultados qualitativos (Hastie, Tibshirani e Friedman, 2009). As variáveis podem ser caraterizadas como sendo quantitativas ou qualitativas (também conhecidas como categóricas). As variáveis quantitativas assumem valores numéricos, como por exemplo a idade e altura de uma pessoa ou o valor de uma casa. Por outro lado, as variáveis qualitativas assumem valores de classes ou categorias, como por exemplo o gênero de uma pessoa (masculino ou feminino) ou a marca de produto (marca A, B ou C) (James et al., 2013). Esta distinção pode ser vista na Figura 4-3.

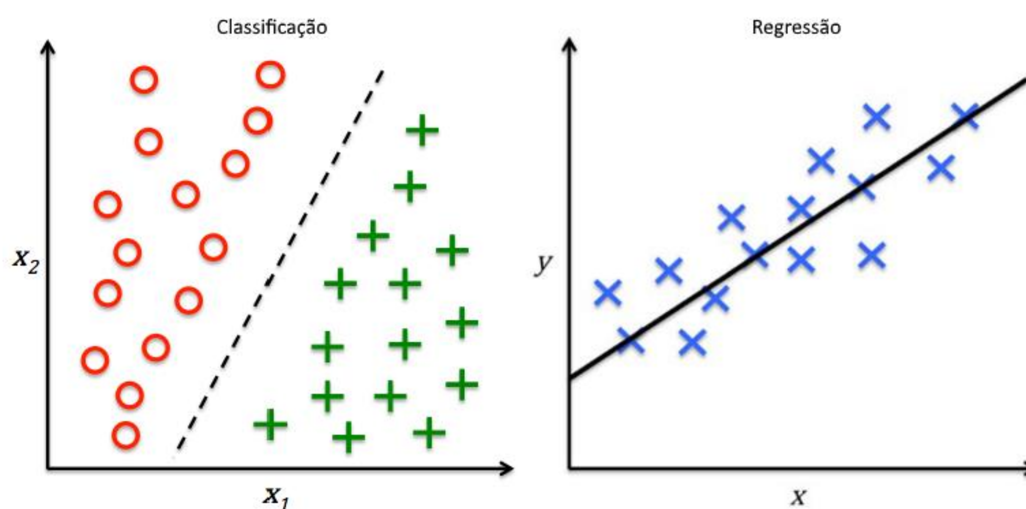

Figura 4-3: Classificação vs Regressão
Fonte: Python Machine Learning (Raschka, 2016)

Por outro lado, aplicações em que o conjunto de dados utilizado contem atributos sem qualquer valor alvo são conhecidas por problemas de *Unsupervised Learning*. Este tipo de aprendizagem pode ter vários objetivos dependendo da sua aplicação específica. Pode ser utlizado por exemplo para descobrir de grupos de dados semelhantes, chamado *Clustering*, ou para projetar dados com um elevado número de dimensões em apenas duas ou três dimensões, chamado *Dimensionality Reduction* (Bishop, 2006; Hackeling, 2014; James et al., 2013; Raschka, 2016).

Por último, existem ainda problemas de *Reinforcement Learning*. Este tipo de aprendizagem preocupa-se com problemas em que é de encontrar ações adequadas para uma determinada situação, a fim de maximizar uma recompensa. Neste caso, a aprendizagem não fornece exemplos de resultados ótimos, como acontece com *Supervised Learning*, em vez disso, tenta descobri-los utilizando um processo de tentativa e erro. Normalmente, existe uma sequência de estados e ações utilizadas para interagir com o ambiente. Em muitos casos, a ação atual não só afeta a recompensa imediata, mas também tem um impacto na recompensa em todas as etapas subsequentes (Bishop, 2006; Raschka, 2016).

As técnicas a aplicar a determinado problema tendem a ser escolhidas dependendo do tipo de aprendizagem necessária e/ou do tipo de variável alvo. Porém, a grande maioria das técnicas de Machine Learning pode ser implementada tanto como uma técnica de regressão, bem como uma técnica de classificação (James et al., 2013).

O problema em estudo nesta dissertação pode ser considerado um problema de Supervised Learning, mais concretamente um problema de regressão. Pretende-se estimar o custo, uma variável quantitativa, através da utilização de um conjunto de dados que contem atributos e seus os respetivos valores alvo.

4.2.1 Linear Regression

A regressão linear (*Linear Regression* – LR) estuda o relacionamento entre uma variável dependente, isto é, o valor a estimar e uma ou mais variáveis independentes. Se o problema em estudo apenas contiver uma variável independente é utilizada a regressão linear simples, por outro lado se contiver várias variáveis independentes é utilizada a regressão linear múltipla. A regressão linear tenta representar a variável dependente como uma combinação linear das variáveis independentes (Malhotra e Jain, 2011; Raschka, 2016; scikit-learn, 2017; Witten et al., 2016), como pode ser visto em (4-1).

$$y = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n \quad (4\text{-}1)$$

Onde y representa a variável dependente, $x_1, x_2, \dots, x_n$ representam as variáveis independentes e $w_0, w_1, w_2, \dots, w_n$ são coeficientes do modelo. Os coeficientes são aprendidos pelo modelo quando este é treinado utilizando um determinado conjunto de dados (Raschka, 2016; Witten et al., 2016)

A regressão linear é uma técnica extremamente simples que tem sido amplamente utilizada nas mais diversas aplicações estatísticas. Uma desvantagem da sua simplicidade é que os modelos lineares requerem dados lineares. Se os dados utilizados exibirem dependências não lineares, a precisão do modelo encontrado diminui rapidamente. Apesar disto, os modelos lineares são muitas vezes testados antes de outros modelos de aprendizagem mais complexos (Witten et al., 2016).

4.2.2 Decision Trees

Uma árvore, em termos computacionais, é uma estrutura que representa objetos graficamente (Braga, Oliveira e Meira, 2007). As técnicas baseadas em árvores, quer para problemas de regressão, bem como para problemas de classificação, recorrem tipicamente à subdivisão recursiva de um conjunto de valores alvo com base no conjunto de dados de treino utilizado. O conjunto de regras de divisão resultante deste processo pode ser representado recorrendo a uma

árvore de decisão (*Decision Tree* – DT) (Hackeling, 2014; Hastie, Tibshirani e Friedman, 2009; James et al., 2013; Raschka, 2016).

A estrutura de uma árvore de decisão é composta por nós (raiz, nós internos e folhas) e arestas. Os nós internos representam uma determinada condição e são ligados por arestas que especificam os possíveis resultados de cada condição. Os dados de treino são assim sucessivamente divididos em subconjuntos. Por exemplo, um nó pode testar se o valor de uma variável excede um determinado limite. As instâncias dos dados de treino que satisfaçam a condição seguem para um nó e as instâncias que não satisfaçam a condição seguem para outro. Este processo é repetido até que um critério de paragem seja atingido, como por exemplo um limite máximo de profundidade da árvore (Hackeling, 2014; James et al., 2013; Malhotra e Jain, 2011).

Em problemas de classificação as folhas da árvore representam classes ou categorias. Por outro lado, em problemas de regressão é comum calcular a média ou o a moda te todos as instâncias que compõem uma folha da árvore de modo a produzir um único valor para cada folha. Depois de construída a árvore de decisão, a obtenção de uma previsão para uma nova instância consiste na sucessiva avaliação das condições dos nós até que seja alcançada uma folha (Hackeling, 2014; James et al., 2013; Malhotra e Jain, 2011).

A Figura 4-4 apresenta um exemplo da subdivisão de um conjunto de dados utilizando uma árvore de decisão.

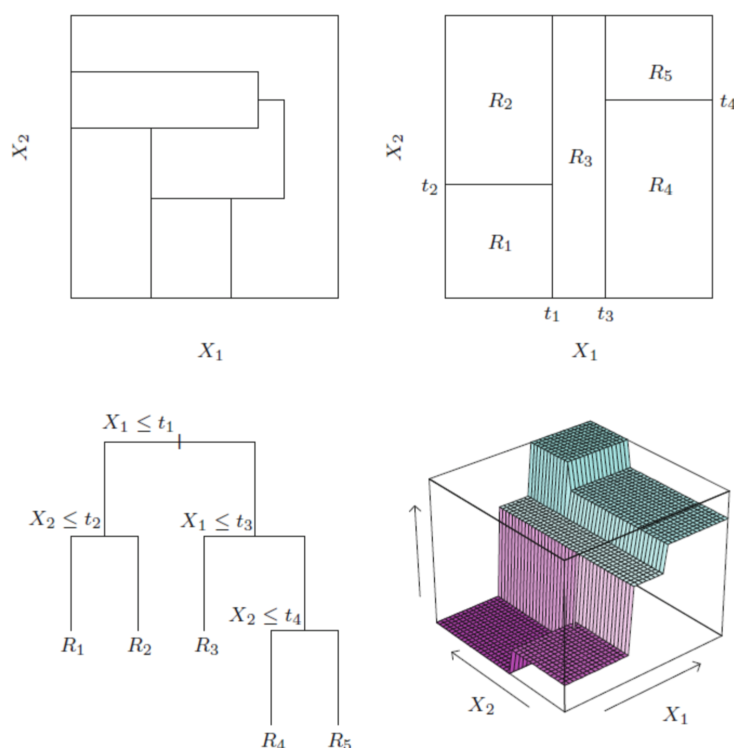

Figura 4-4: Subdivisão de um conjunto de dados utilizando uma árvore de decisão
Fonte: The Elements of Statistical Learning (Hastie, Tibshirani e Friedman, 2009)

O painel superior esquerdo da Figura 4-4 mostra uma subdivisão que não pode ser obtida a partir de divisão binária recursiva de uma árvore de decisão. O painel superior direito da Figura 4-4 mostra a subdivisão de um conjunto de dados bidimensional por uma árvore de decisão. O painel inferior esquerdo mostra a árvore de decisão correspondente à subdivisão do painel superior direito e o painel inferior direito mostra uma representação gráfica tridimensional das previsões possíveis. As técnicas baseadas em árvores de decisão fornecem uma forma de simples de realizar o processo de classificação ou regressão. Uma das suas principais vantagens é a sua fácil interpretação e compreensão pelos seres humanos. Para além disto, estas podem também ser utilizadas em conjuntos de dados que com dados em falta e ainda dados categóricos (Hastie, Tibshirani e Friedman, 2009; Malhotra e Jain, 2011; Raschka, 2016; scikit-learn, 2017).

O modelo pode ser totalmente representado por uma única árvore de decisão. A utilização de conjuntos de dados com mais de duas dimensões torna uma representação idêntica à do painel superior direito da Figura 4-4 difícil, porém a representação sob a forma de árvore funciona exatamente da mesma forma (Hastie, Tibshirani e Friedman, 2009; Raschka, 2016). Os modelos produzidos são simultaneamente conceitualmente simples e poderosos (Hastie, Tibshirani e Friedman, 2009). Os modelos baseados em árvores de decisão estão entre os mais populares e frequentemente utilizados, tendo sido aplicados com sucesso a um vasto conjunto de cenários, desde o diagnóstico médico até à avaliação de risco de crédito dos candidatos a empréstimos (Mitchell, 1997).

4.2.3 K-Nearest Neighbors

K-Nearest Neighbors (KNN) é um exemplo típico de um *lazy learner*. É apelidado desta forma porque não "aprende" a partir do conjunto de dados de treino, em vez disto, memoriza todos as instâncias de treino (Raschka, 2016). *Lazy learners* adiam toda a generalização até ao momento de realizar uma nova previsão. Isto leva a que os modelos não necessitem de ser treinados, porém, estes são geralmente lentos em comparação com outro tipo de modelos (Hackeling, 2014).

O processo de previsão de um modelo baseado em *K-Nearest Neighbors* é bastante simples e pode ser resumido pelas seguintes etapas:

1. Escolha do número de k e de uma métrica de distância;
2. Seleção das k instâncias mais próximas da instância a prever;
3. Obtenção da previsão através das instâncias selecionadas (normalmente recorrendo a votação, no caso de um problema de classificação, ou à média, no caso de um problema de regressão).

A Figura 4-5 exemplifica a previsão de uma nova instância através do processo descrito anteriormente, utilizando $k = 5$.

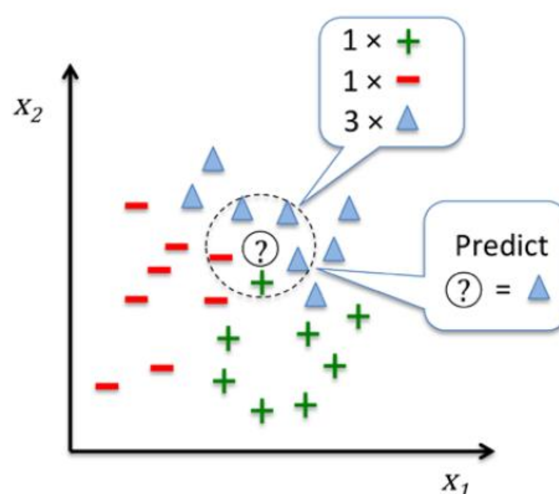

Figura 4-5: Previsão de uma nova instância utilizando *K-Nearest Neighbors*
Fonte: Python machine learning (Raschka, 2016)

Com base na métrica de distância escolhida, o modelo encontra as k instâncias do conjunto de dados de treino com maior proximidade à instância a prever. A atribuição da previsão é posteriormente realizada recorrendo a votação, no caso de um problema de classificação, ou à média, no case de um problema de regressão (Raschka, 2016).

A principal vantagem de um modelo que armazena todas instâncias de treino é que este se adapta imediatamente à adição de novas instâncias. Por outro lado, tem como desvantagem que a quantidade de memória e poder computacional necessário cresce proporcionalmente à quantidade de dados de treino (Raschka, 2016).

Um melhoramento simples que é geralmente aplicado a modelos baseados em *K-Nearest Neighbors* é a ponderação das instâncias selecionadas de acordo com as suas respetivas distâncias à instância a estimar. Instâncias mais próximas obterão uma ponderação mais elevada, e instâncias mais afastadas uma ponderação menor (Mitchell, 1997).

Os modelos baseados em *K-Nearest Neighbors* que efetuam a ponderação das distâncias são altamente eficazes em muitos problemas práticos. A sua robustez em relação a possível ruído existente nos dados de treino é uma grande vantagem deste tipo de modelos. A utilização da média ponderada das instâncias selecionadas permite eliminar o impacto de instâncias anormais do conjunto de dados de treino (Mitchell, 1997).

Para além a escolha do valor de k, também a escolha da métrica de distância é fundamental para a construção de um bom modelo. Um problema existente nos modelos baseados em *K-Nearest Neighbors* é que a distância é calculada relativamente a todos os atributos do conjunto de dados de treino. Este problema é principalmente visível, quando *K-Nearest Neighbors* é aplicado a um conjunto de dados com um grande número de atributos, onde apenas um número muito reduzido destes é realmente relevante para as previsões. Nestes casos,

instâncias que possuam um valor semelhante nos atributos realmente relevantes podem estar extremamente distantes e consequentemente, a métrica de distância utilizada irá retornar resultados enganosos (Mitchell, 1997; Raschka, 2016). Geralmente, para combater este problema é utilizada como métrica de distância a distância euclidiana e os dados são normalizados recorrendo à normalização euclidiana de modo a que todos os atributos contribuam de igual forma para o cálculo da distância (Raschka, 2016; scikit-learn, 2017).

Apesar da sua simplicidade, modelos baseados em *K-Nearest Neighbors* foram aplicados com sucesso numa grande quantidade de problemas, incluindo a classificação de dígitos manuscritos e na deteção de padrões em imagens de satélite (Hastie, Tibshirani e Friedman, 2009).

4.2.4 Support Vector Regression

A *Support Vector Regression* (SVR) é uma adaptação das *Support Vector Machines* (SVM) para problemas de regressão. As *Support Vector Machines* foram originalmente desenvolvidas para a realização de tarefas de classificação binária. Neste tipo de problemas, o objetivo é maximizar a margem do plano que separa as classes dos dados, sendo que as instâncias mais próximas da margem são chamadas de vetores de suporte (*Support Vectors*) (Basak, Pal e Patranabis, 2007; Raschka, 2016; scikit-learn, 2017; Smola e Schölkopf, 2004). A Figura 4-6 apresenta um exemplo de um problema deste tipo e a sua respetiva solução.

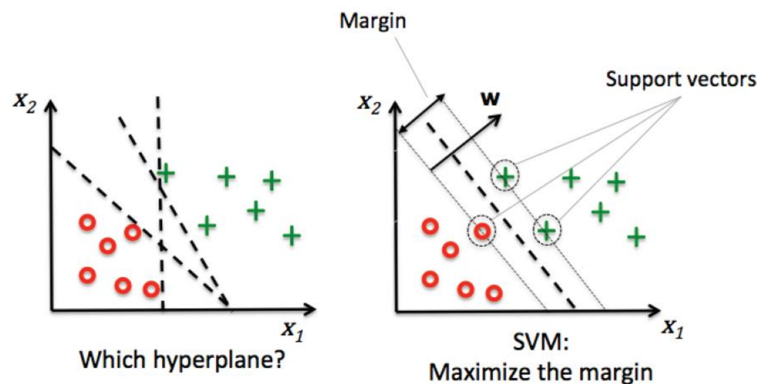

Figura 4-6: Maximização da margem do plano que separa as classes utilizando SVM
Fonte: Python machine learning (Raschka, 2016)

As *Support Vector Machines* tentam determinar o melhor plano que separa as diferentes classes, porém, isto nem sempre é possível. Nestes casos recorre-se à utilização de kernels. O principal objetivo da utilização de um *kernel* é o mapeamento dos dados para um espaço vetorial com um maior número de dimensões, aumentando o número de valores necessários para identificar um ponto. Como pode ser visto na Figura 4-7 os dados iniciais não são linearmente separáveis. Após o seu mapeamento para um espaço tridimensional, estes passam a ser linearmente separáveis. Esta abordagem pode ser aplicada tanto em problemas de classificação como em problemas de regressão (Cortez, 2010; Oliveira, 2006; Raschka, 2016).

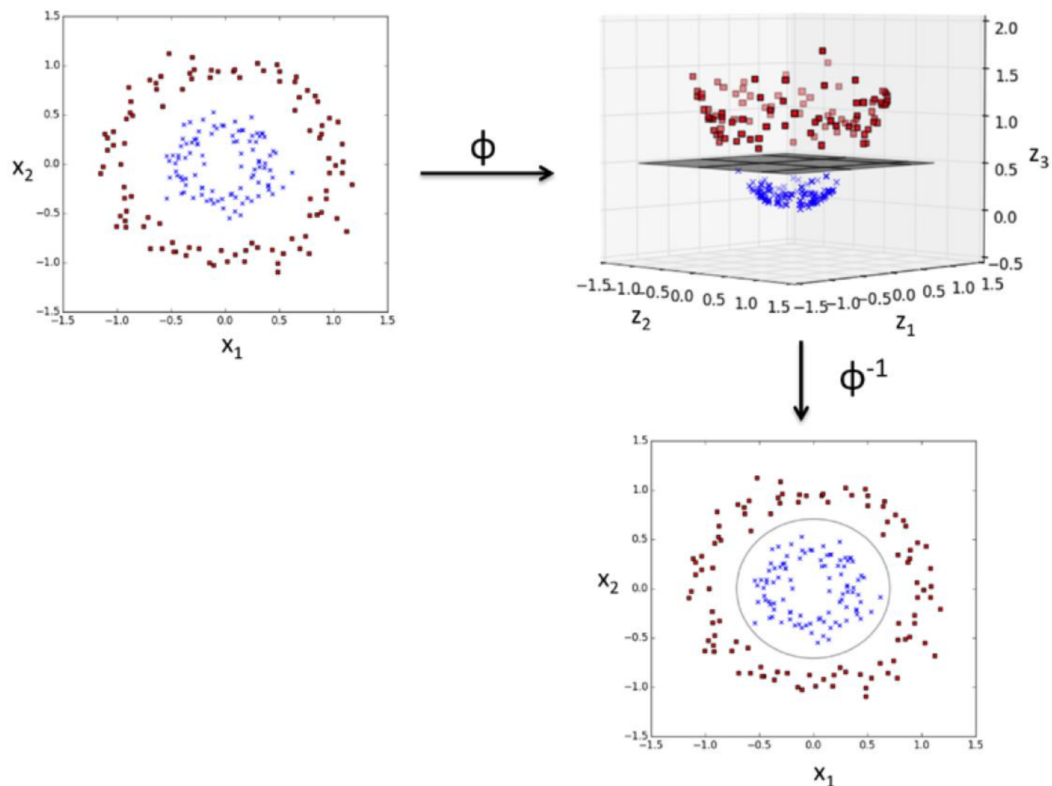

Figura 4-7: Mapeamento de atributos para um espaço vetorial com mais dimensões
Fonte: Python machine learning (Raschka, 2016)

A *Support Vector Regression* utiliza os mesmos princípios das *Support Vector Machines*, porém, neste caso, o objetivo é determinar uma função que possua no máximo um desvio de ε sobre todos as instâncias do conjunto de dados de treino. Imaginando um tubo à volta da função de regressão, procura-se determinar a função com o tubo mais fino possível (Basak, Pal e Patranabis, 2007; Raschka, 2016; Smola e Schölkopf, 2004). Este conceito é mostrado pela Figura 4-8.

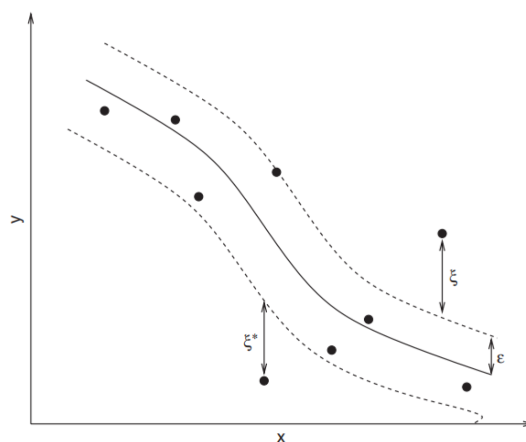

Figura 4-8: Regressão utilizando *Support Vector Regression*
Fonte: Estimation of software project effort with support vector regression (Oliveira, 2006)

Modelos baseados em *Support Vector Machines* ou em *Support Vector Regression* são modelos lineares muito bons, que por sua vez podem ser estendidos para problemas não lineares através da utilização de kernels. Apesar disto, este tipo de modelos possui um grande número de parâmetros que devem ser otimizados de modo a maximizar a precisão das suas previsões (Basak, Pal e Patranabis, 2007; Raschka, 2016; Smola e Schölkopf, 2004).

4.2.5 Multi Layer Perceptron

O cérebro humano possui um grande poder computacional e lógico. As redes neuronais artificiais (*Artificial Neural Network* – ANN) são técnicas computacionais que apresentam um modelo matemático inspirado na estrutura neuronal dos organismos inteligentes e que adquirem conhecimento através da experiência (Dave e Dutta, 2014; Prabhakar, 2013).

As redes neuronais artificiais podem ser divididas em três componentes. O primeiro é a sua arquitetura ou topologia. Esta descreve o número e a estrutura das camadas de neurónios, bem como as ligações entre as mesmas. O segundo componente é a função de ativação. Esta função é utilizada por cada neurónio de forma a gerar um sinal de saída dependendo das suas entradas. Por fim, o terceiro componente é o algoritmo de aprendizagem utilizado para treinar a rede (Hackeling, 2014; Raschka, 2016).

Existem principalmente dois tipos de redes neuronais artificiais: *feedforward neural networks* e *feedback neural networks*. As redes *feedforward* são as mais utilizadas e são definidas como sendo um grafo acíclico. Os sinais de entrada percorrem a rede apenas em uma direção (em direção à camada de saída). Por outro lado, as redes *feedback* (também conhecidas como *recurrent neural networks*) podem conter ciclos. Os ciclos podem representar o estado interno da rede. Isto pode influenciar o comportamento da rede a mudar ao longo do tempo. As redes feedforward são tipicamente utilizadas para mapear as entradas para as saídas, enquanto as redes feedback são tipicamente utilizadas para processar dados sequenciais ou temporais (Hackeling, 2014; Nassif et al., 2016; Raschka, 2016; Sehra, Brar e Kaur, 2013).

A arquitetura mais simples de uma rede neuronal artificial é o *Multi Layer Perceptron* (MLP). Esta arquitetura pode ser vista na Figura 4-9. Um *Multi Layer Perceptron* é uma rede feedforward com uma ou mais camadas ocultas entre a camada de entrada e a camada de saída (Bishop, 2006; Braga, Oliveira e Meira, 2007; Dave e Dutta, 2014; Hackeling, 2014; Nassif et al., 2016; Raschka, 2016).

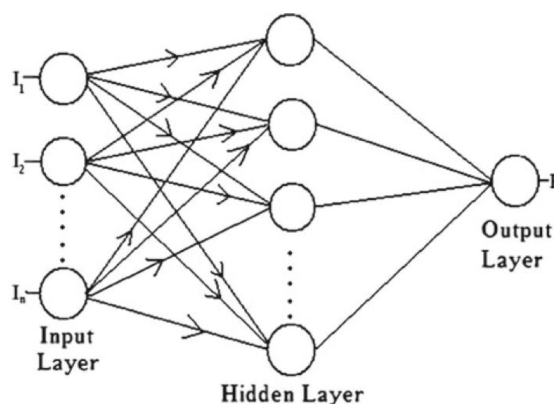

Figura 4-9: Representação de um *Multi Layer Perceptron*
Fonte: Neural network based models for software effort estimation: a review (Dave e Dutta, 2014)

A camada de entrada possui um neurónio por cada atributo do conjunto de dados utilizado. Segue-se uma ou mais camadas ocultas que podem ter um número variável de neurónios. Estas camadas podem ser parcialmente ou completamente ligadas com as camadas seguintes. Por fim, a camada de saída pode ser composta por um ou mais neurónios de saída. Para problemas de regressão a camada de saída é geralmente composta apenas por um neurónio e para problemas de classificação é geralmente composta por tantos neurónios como o número de classes existentes (Braga, Oliveira e Meira, 2007; Dave e Dutta, 2014; Nassif et al., 2016).

O algoritmo de aprendizagem mais conhecido e utilizado para treinar redes neuronais é o *backpropagation*. Este algoritmo é dividido em duas fases. Na primeira fase um sinal de entrada é propagado desde a camada de entrada até à camada de saída de modo a produzir uma previsão. Durante esta fase os pesos associados a cada neurónio são fixos. Na segunda fase é calculado o erro da previsão relativo ao valor real. Este erro é de seguida propagado no sentido inverso, da camada de saída para a camada de entrada, e os pesos de cada neurónio são ajustados de modo a minimizar o mesmo. Este processo é repetido para todas as instâncias do conjunto de dados utilizado e até a rede atingir o desempenho desejado (Bishop, 2006; Nassif et al., 2016; Prabhakar, 2013, 2013; Raschka, 2016; Srinivasan e Fisher, 1995).

Modelos baseados em *Multi Layer Perceptron*, ou em redes neuronais artificiais em geral, apresentam resultados muito positivos em uma grande variedade de problemas. São especialmente bons em problemas com grandes quantidades de dados ou atributos (Braga, Oliveira e Meira, 2007; Dave e Dutta, 2014).

4.2.6 Ensembles

As técnicas *ensemble* são técnicas de aprendizagem que combinam outras técnicas (tipicamente chamadas de técnicas base) através de um mecanismo de agregação. A previsão produzida por um modelo *ensemble* é uma combinação das previsões produzidas pelos seus modelos base, por exemplo, a média das previsões destes. O principal fundamento subjacente às técnicas *ensemble* é que se os seus modelos base forem precisos e diversos, então a precisão do modelo *ensemble*

será melhor que cada modelo base individual (Azzeh, Nassif e Minku, 2015; Dietterich, 2000; Hackeling, 2014; Hastie, Tibshirani e Friedman, 2009; Raschka, 2016; Seni e Elder, 2010).

Dois modelos são considerados diversos se produzirem erros diferentes para os mesmos dados (Azzeh, Nassif e Minku, 2015; Chandra e Yao, 2006). É esperado que uma diversidade de modelos base produza previsões fracas para diferentes subconjuntos dos dados. Assim, as fracas previsões de alguns modelos podem ser compensadas pelas previsões mais precisas de outros, o que permite que o modelo *ensemble* como um todo consiga um melhor desempenho do que os seus modelos base. Por outro lado, se o modelo *ensemble* for composto por uma baixa diversidade de modelos, o seu desempenho não será melhor do que o desempenho individual dos seus modelos base (Azzeh, Nassif e Minku, 2015; Brown et al., 2005; Kuncheva e Whitaker, 2003).

O processo de criação de um modelo *ensemble* pode ser dividido em duas tarefas. Primeiramente a criação de vários modelos base e posteriormente a combinação das previsões destes em uma única previsão (Hastie, Tibshirani e Friedman, 2009). Os modelos *ensemble* pode geralmente ser agrupados em três categorias (scikit-learn, 2017; Sewell, 2008):

- *Bagging – Bootstrap Aggregating*: este método utiliza vários subconjuntos aleatórios do conjunto de dados de treino utilizado. Cada um desses conjuntos de dados é utilizado para treinar um modelo base diferente. As previsões dos modelos base são combinadas através da sua média (no caso da regressão) ou através de votação (no caso da classificação) para criar um único resultado (scikit-learn, 2017; Sewell, 2008);
- *Boosting*: neste método primeiro é criado um modelo fraco, ou seja, um modelo cuja precisão no conjunto de dados de treino seja apenas um pouco melhor do que uma previsão completamente aleatória. De seguida uma sucessão de modelos é construída iterativamente, cada um sendo treinado recorrendo ao conjunto de dados em que o modelo anterior classificou de forma incorreta, ou no caso de regressão, que o modelo anterior não conseguiu prever com precisão suficiente. Finalmente, todos os modelos são ponderados de acordo com seu desempenho e, de seguida, as suas previsões são combinadas através da sua média ponderada (no caso da regressão) ou através de votação (no caso da classificação) para criar um único resultado (scikit-learn, 2017; Sewell, 2008);
- *Stacking*: este método é menos utilizado do que os dois métodos anteriores. Ao contrário dos métodos anteriores, o Stacking pode ser (e normalmente é) utilizado para combinar modelos de diferentes tipos. Primeiramente o conjunto de dados de treino é dividido em duas partes. Vários modelos são treinados utilizado a primeira parte dos dados e posteriormente avaliados utilizando a segunda parte. De seguida outro modelo é treinado utilizando as previsões dos vários modelos anteriores e os valores alvo corretos do conjunto de dados original. A previsão produzida por este modelo é então utilizada como previsão final do modelo (Sewell, 2008).

Em suma, em vez de propor novas técnicas de aprendizagem, os modelos *ensemble* tomam partido da combinação de técnicas já existentes. Técnicas *ensemble* têm sido largamente utilizadas na engenharia de *software* para resolver problemas de regressão e classificação. Mais especificamente na área de estimativa de esforço, Jørgensen recomenda a utilização de várias estimativas na avaliação de especialistas em vez de apenas uma. O mesmo pricipio é utilizado nas técnicas *ensemble* (Azzeh, Nassif e Minku, 2015; Jørgensen, 2004).

4.2.6.1 Bagging

Bagging, como o próprio nome indica, segue os princípios de *Bootstrap Aggregating* apresentados em 4.2.6. Este método cria várias instâncias do mesmo modelo utilizando subconjuntos aleatórios do conjunto de dados de treino original e, em seguida, combina as suas previsões individuais para formar uma previsão final (scikit-learn, 2017; Sewell, 2008).

Bagging é utilizado para melhorar o desempenho da técnica utilizada nos seus modelos base através da introdução de alguma aleatoriedade na construção dos vários modelos e posteriormente criando um modelo *ensemble* final a partir destes. Muitas vezes esta abordagem permite aumentar o desempenho em relação a um modelo único, sem a necessidade de adaptar as técnicas utilizadas nos modelos subjacentes (scikit-learn, 2017).

4.2.6.2 Random Forest

As *Random Forest* (RF) ganharam grande popularidade nas aplicações de *Machine Learning* durante a última década devido ao seu bom desempenho, escalabilidade e facilidade de uso (Raschka, 2016). A Random Forest é uma técnica *ensemble* que realiza *Bagging* sobre modelos construídos utilizando árvores de decisão (*Decision Trees*).

Cada árvore do *ensemble* é construída a partir de uma amostra aleatória (não só de instâncias, mas também de atributos) retirada do conjunto de dados de treino. Além disto, a divisão de um nó durante a construção da árvore não é realizada de forma a obter o melhor resultado sobre o conjunto de dados originais, mas de forma a obter o melhor resultado sobre a amostra aleatória utilizada (scikit-learn, 2017).

A grande diferença em relação ao que acontece no *Bagging* é que nem todos os atributos são considerados na condição de cada nó das árvores de decisão dos modelos base da *Random Forest*, aumentando assim a aleatoriedade do modelo *ensemble* resultante. Esta introdução de aleatoriedade na construção dos vários modelos permite melhorar o desempenho do modelo *ensemble* resultante (James et al., 2013; Raschka, 2016; scikit-learn, 2017).

4.2.6.3 Extremely Randomized Trees

A técnica *Extremely Randomized Trees* é idêntica às *Random Forests*, porém a aleatoriedade introduzida é maior na forma como as divisões são calculadas. Tal como acontece com as *Random Forests*, cada modelo base é construído a partir de uma amostra aleatórias do conjunto de dados de treino. A diferença estre estas duas técnicas está na forma como os nós realizam a

divisão dos dados. Enquanto nas *Random Forests* a divisão o valor exato da divisão é escolhido de forma a obter a melhor divisão possível, nas *Extremely Randomized Trees* o valor exato da divisão é escolhido de forma aleatória (Geurts, Ernst e Wehenkel, 2006; scikit-learn, 2017).

4.2.6.4 Gradient Boosting

Gradient Boosting é uma generalização de *Boosting* para funções de custo diferenciáveis. É um procedimento preciso e eficaz que pode ser utilizado tanto para problemas de regressão como para problemas de classificação. Os modelos baseados em *Gradient Boosting* são utilizados em uma variedade de áreas, incluindo ranking de pesquisas na web (scikit-learn, 2017).

O *Gradient Boosting* é uma técnica caracterizada por iterativamente fazer a aprendizagem de modelos fracos e adicioná-los de forma a obter o modelo final. A Figura 4-10 apresenta a construção de um modelo utilizando *Gradient Boosting*. Primeiramente é construído um modelo utilizando o conjunto de dados originais (pontos do primeiro painel da Figura 4-10), representado pelo segundo painel da Figura 4-10. De seguida, os modelos seguintes (terceiro e quarto painel da Figura 4-10) são construídos utilizando os resíduos do modelo completo anterior. O modelo final corresponde à soma das previsões de todos os modelos construídos (Friedman, 1999; scikit-learn, 2017).

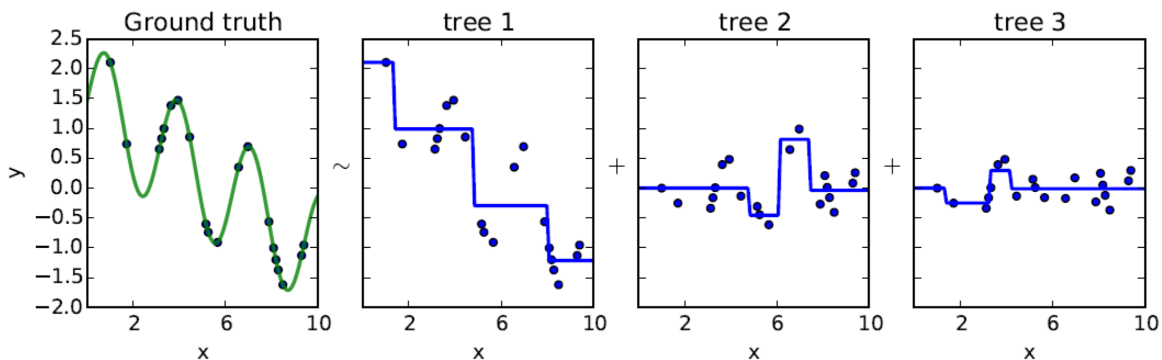

Figura 4-10: Exemplo dos modelos base de um modelo de Gradient Boosting
Fonte: Gradient boosted regression trees in scikit-learn (Prettenhofer e Louppe, 2014)

Tipicamente os modelos base de um modelo *Gradient Boosting* são baseados em árvores de decisão (*Decision Trees*) de baixa profundidade devido à sua elevada flexibilidade (Friedman, 1999; scikit-learn, 2017).

4.3 Sumário

Apesar de estudos recentes revelarem que métodos como a avaliação de especialistas continuam a ser dos mais utilizados na estimativa de esforço de desenvolvimento de *software*, a investigação de métodos baseados em técnicas de *Machine Learning* tem vindo a aumentar

(Wen et al., 2012). A escolha da melhor técnica a utilizar pode ser um dos maiores desafios práticos da aplicação de *Machine Learning* nas estimativas de esforço (James et al., 2013).

Existem uma grande variedade de fatores que devem ser considerados para a seleção de uma técnica de *Machine Learning*. Esta escolha deve ser orientada à capacidade organizacional e também à experiencia de quem as implementa. Em termos de necessidades, o objetivo mais comum é maximizar a precisão dos modelos, porém, outras abordagens devem também ser consideradas. Por exemplo, uma técnica que produz previsões ligeiramente menos precisas, mas, em geral, é mais robusta pode ser preferida, especialmente nos casos em que as organizações não têm acesso a conjuntos de dados adequados. Embora seja muito positivo que técnicas mais sofisticadas (e potencialmente mais úteis) tenham sido investigadas e utilizadas para construir modelos de estimativa de esfoço, benefícios notáveis só serão alcançados se essas técnicas forem utilizadas corretamente (de Barcelos Tronto, da Silva e Sant'Anna, 2008).

Em 1996, David Wolpert (Wolpert, 1996) afirmou que não existem "almoços grátis" na estatística ("no free lunch") e que por isso é necessário fazer cedências durante o processo de seleção (de Barcelos Tronto, da Silva e Sant'Anna, 2008). Nenhuma técnica é melhor que todas as restantes em todos os conjuntos de dados possíveis. Num determinado conjunto de dados, uma técnica específica pode obter melhores resultados, mas outra técnica pode funcionar melhor num conjunto de dados semelhantes, mas diferente (James et al., 2013). A obtenção de uma maior flexibilidade numa determinada área é normalmente indicação da introdução de suposições extras, ou da diminuição do poder de generalização do modelo noutras situações, igualmente importantes (Simon e Tibshirani, 2014). Isto faz com que seja extremamente importante selecionar a técnica que produz os melhores resultados para cada problema e conjunto de dados (James et al., 2013).

5. Critérios de Avaliação

A avaliação do desempenho dos modelos pode ser obtida através de diferentes critérios. Estes critérios são utilizados para medir o desempenho dos modelos em termos da sua precisão. Existem diversos critérios de avaliação descritos na literatura, sendo os mais frequentemente utilizados na avaliação de modelos de estimativa de esforço de desenvolvimento de *software* o *Mean Absolute Error* (MAE), o *Magnitude of Relative Error* (MRE), o *Mean Magnitude of Relative Error* (MMRE), o *Mean of Magnitude of Error Relative to the estimate* (MMER) e o *Percentage Relative Error Deviation* (PRED(x)). Estes critérios serão descritos em maior detalhe nas secções seguintes (Azzeh, Nassif e Minku, 2015; Elish, 2009; Fedotova, Teixeira e Alvelos, 2013; Menzies et al., 2006; Nassif, Ho e Capretz, 2013; Satapathy, 2016; Wen et al., 2012).

Todos estes critérios avaliam diferentes aspetos dos modelos. Por exemplo, PRED avalia o desempenho positivo enquanto o MMRE avalia o desempenho negativo. Um grande erro poderá afetar o MMRE e não afetar o PRED (Menzies et al., 2006). Shepperd e Schofield comentaram que o MMRE é bastante conservador no que toca a estimativas demasiado elevadas, enquanto que o PRED(30) irá identificar modelos que são geralmente precisos porém podem ocasionalmente ser largamente imprecisos (Shepperd e Schofield, 1997).

5.1 Mean Absolute Error

O *Mean Absolute Error* (MAE) representa a média dos erros absolutos entre o esforço atual e o esforço estimado, como é mostrado em (5-1).

$$MAE = \frac{1}{T}\sum_{i=1}^{T}|EA_i - EE_i| \quad (5\text{-}1)$$

Onde EA_i é o esforço atual observado e EE_i é o esforço estimado relativamente ao item i do *dataset* utilizado e T é o número total de itens no *dataset*.

5.2 Magnitude of Relative Error

O *Magnitude of Relative Error* (MRE) representa o erro relativo entre o valor real e o valor estimado em relação ao valor real, como é mostrado em (5-2).

$$MRE = \frac{|EA_i - EE_i|}{EA_i} \quad (5\text{-}2)$$

Onde EA_i é o esforço atual observado e EE_i é o esforço estimado relativamente ao item i do *dataset* utilizado e T é o número total de itens no *dataset*. O resultado pode ser convertido em percentagem através da multiplicação do MRE resultante por 100.

5.3 Mean Magnitude of Relative Error

O *Mean Magnitude of Relative Error* (MMRE) representa a média dos erros relativos entre o valor real e o valor estimado em relação ao valor real (MRE), sobre T observações, como é mostrado em (5-3).

$$MMRE = \frac{1}{T}\sum_{i=1}^{T}MRE_i \quad (5\text{-}3)$$

Onde MRE_i é o MRE relativamente ao item i do *dataset* utilizado e T é o número total de itens no *dataset*.

5.4 Mean of Magnitude of Error Relative to the Estimate

O *Mean of Magnitude of Error Relative to the estimate* (MMER) é um critério semelhante ao MMRE, porém as medições são relativas às estimativas, como é mostrado em (5-4). Foss et al. defenderam que o MMER pode por vezes ser mais preciso que o MMRE (Foss et al., 2003).

$$MMER = \frac{1}{T}\sum_{i=1}^{T}\frac{|EA_i - EE_i|}{EE_i} \tag{5-4}$$

Onde EA_i é o esforço atual observado e EE_i é o esforço estimado relativamente ao item i do *dataset* utilizado e T é o número total de itens no *dataset*.

5.5 Percentage Relative Error Deviation

O *Percentage Relative Error Deviation* (PRED(x)) representa a percentagem de estimativas cujos erros relativos (MRE) não são maiores que x, como é mostrado em (5-5). A precisão das estimativas é diretamente proporcional ao PRED(x) e inversamente proporcional ao MMER. A maioria dos autores recomenda a utilização de 25 como valor de x (Fedotova, Teixeira e Alvelos, 2013).

$$PRED(x) = \frac{1}{T}\sum_{i=1}^{T}\begin{cases}1 \; se \; MRE_i \leq x \\ 0 \; caso \; contrário\end{cases} \tag{5-5}$$

Onde MRE_i é o MRE relativamente ao item i do *dataset* utilizado e T é o número total de itens no *dataset*.

5.6 Sumário

Um dos principais desafios de um modelo de estimativa de esforço consiste na sua capacidade de produzir estimativas precisas (Fedotova, Teixeira e Alvelos, 2013). Para medir o desempenho dos modelos em termos da sua precisão são utilizados alguns critérios de avaliação. Os critérios de avaliação mais frequentemente utilizados na literatura são o *Mean Magnitude of Relative Error* (MMRE) e o *Percentage Relative Error Deviation* (PRED(x)). Ambos critérios são baseados no *Magnitude of Relative Error* (MRE) (Fedotova, Teixeira e Alvelos, 2013; Myrtveit, Stensrud e Shepperd, 2005; Port, Nguyen e Menzies, 2009). Um modelo de estimativa é geralmente considerado aceitável se conseguir resultados de MMRE ≤ 25% e PRED(25) ≥ 75% (Conte, Dunsmore e Shen, 1986; Fedotova, Teixeira e Alvelos, 2013; Wen et al., 2012).

Para esta dissertação foram adotados os critérios de avaliação MMRE e PRED(x) visto que são os mais utilizados na literatura. Isto permite aumentar a possibilidade de comparação dos modelos criados com modelos já existentes e também com modelos futuros. O valor de x utilizado foi 25 como é recomendado pela maioria dos autores. Para além do MMRE e PRED(25) foi também utilizado o MAE de modo a avaliar a precisão das estimativas em termos absolutos.

6. Modelo Proposto

As metodologias ágeis são utilizadas no desenvolvimento de *software* principalmente para que as organizações possam dar resposta à volatilidade dos requisitos. As organizações podem assim avaliar o foco dos desenvolvimentos ao longo do ciclo de vida do projeto (Beck et al., 2001; Satapathy, 2016). Ao promover a repetição de ciclos de trabalho o desenvolvimento torna-se aditivo e iterativo. Em vez de tentar desenvolver uma solução completa de uma vez só, as metodologias ágeis permitem que as equipas possam planear e replanear as suas iterações de modo a priorizar e otimizar o que é desenvolvido, maximizando assim o valor do produto para o cliente final (Britto, Mendes e Borstler, 2015; Satapathy, 2016; Schmietendorf, Kunz e Dumke, 2008). Esta abordagem tem permitido às organizações que a utilizam superar a sua concorrência que não utiliza metodologias ágeis (Cohen, Lindvall e Costa, 2004; Satapathy, 2016).

Uma vez que os requisitos dos projetos desenvolvidos num ambiente ágil estão em mudança constante, estimar o tamanho e a complexidade dos componentes a serem desenvolvidos torna-se um dos focos principais no processo de desenvolvimento ágil (Satapathy, 2016; Usman, 2015; Usman, Mendes e Börstler, 2015). Geralmente o planeamento de uma iteração ágil é realizada recorrendo à velocidade de iterações anteriores, porém, Cohn (Cohn, 2005) não concorda com esta abordagem. Segundo Cohn, apesar de a velocidade desempenhar um papel importante no planeamento de uma *release*, esta não deve desempenhar um papel idêntico no planeamento de iterações. Primeiro, porque a velocidade é uma medida de estimativas com uma granularidade elevada (*story points* ou *ideal days*), o que não é suficientemente precisa para planear iterações curtas. Estas estimativas de granularidade elevada podem ser utilizadas para estimar a quantidade total de trabalho que uma equipa irá completar durante uma *release*, no entanto, não devem ser utilizadas da mesma forma para planear o trabalho de uma única iteração. Em segundo lugar, uma equipa iria necessita de completar um número de *user stories*

muito elevado por iteração de modo a que os desvios da média nas estimativas se anulassem no decorrer da mesma (Cohn, 2005).

Tendo isto em conta e dada a falta de *datasets* disponíveis na literatura que utilizem *story points*, o modelo proposto utiliza dados relativos a *user stories* individuais. Em vez de utilizar dados relativos a projetos completos, como acontece na maioria dos modelos e *datasets* existentes, a utilização de *user stories* individuais permite assemelhar o modelo proposto aos processos de estimativa de esforço existentes em ambientes ágeis (como o *Planning Poker* em que uma estimativa é obtida para cada *user story*). Esta abordagem permite também obter uma granularidade de dados mais baixa, aumentando assim a quantidade de dados disponíveis para os modelos. Para além dos *story points* e velocidade foram também utilizados mais alguns dados relativos às *user stories*, à equipa que realiza os desenvolvimentos, à iteração e ao projeto. A utilização de mais dados para além de apenas uma medida de tamanho (*story points* neste caso) deve-se ao facto de modelos que utilizam apenas uma medida de tamanho obterem resultados inferiores (Ungan, Cizmeli e Demirors, 2014).

O modelo que se apresenta recorre a várias técnicas de *Machine Learning* para tentar obter estimativas mais precisas. É esperado que com a utilização de técnicas de *Machine Learning* o modelo proposto possa tomar partido da capacidade destas técnicas de conseguirem modelar relacionamentos complexos entre as diversas variáveis em estudo sem serem explicitamente programados. Os resultados obtidos utilizando as diferentes técnicas foram comparados entre si, bem como com resultados obtidos por outros autores disponíveis na literatura.

6.1 Dataset

Devido à falta generalizada de *datasets* disponíveis na literatura que utilizem *story points* procedeu-se à recolha e compilação de dados relativos a *user stories* completas num *dataset* novo. Os dados foram recolhidos de forma anónima de organizações cuja principal área de negócio é o desenvolvimento de *software*. Todos os projetos foram desenvolvidos utilizando metodologias ágeis e todas as *user stories* foram estimadas recorrendo ao método *Planning Poker*.

Foram recolhidos dados relativos a 3128 *user stories* de 11 projetos. A duração dos projetos varia entre 6 a 23 iterações de duas semanas cada. A distribuição de *user stories* e iterações pelos onze projetos pode ser vista na Tabela 6-1. Todas as *user stories* foram estimadas recorrendo ao método *Planning Poker* utilizando a sequência de Fibonacci (0, 1, 2, 3, 5, 8, 13, 21…). O número máximo de *story points* estimados é 21. *User stories* com estimativas superiores a 21 pontos são divididas em múltiplas *user stories* mais pequenas e estimadas novamente. A distribuição de *user stories* por *story points* pode ser vista na Tabela 6-2. Geralmente estimativas realizadas recorrendo a *story points* não podem ser comparadas entre equipas devido à sua natureza relativa. Porém, todas as *user stories* recolhidas foram estimadas tendo por base a mesma escala

relativa. Isto é, quando um projeto novo é iniciado as primeiras estimativas são realizadas tendo por base a experiencia da equipa em projetos passados o que leva a estimativas relativamente uniformes em todos os projetos recolhidos. De forma a minimizar ainda mais este potencial problema, foi também recolhida uma estimativa da velocidade esperada para cada iteração. Este valor é uma tentativa muito simples de normalização das estimativas.

Tabela 6-1: Distribuição de *user stories* e iterações por projeto

Projeto	Número de Iterações	Número de User Stories
P1	15	282
P2	21	385
P3	16	307
P4	6	109
P5	10	225
P6	12	262
P7	23	519
P8	13	293
P9	21	363
P10	16	261
P11	6	122
	Total:	3128

Tabela 6-2: Distribuição de *user stories* por *story points*

Story Points	Número de User Stories
1	101
2	528
3	1006
5	774
8	432
13	181
21	106
Total:	3128

Os dados recolhidos para cada *user story* podem ser agrupados em quatro categorias: atributos do projeto, atributos da iteração, atributos da equipa e atributos da *user story*. Os atributos do projeto são características transversais a todo o projeto e devem apenas ser avaliadas no início do projeto.

Atributos do projeto:

- *Project Name* – Nome do projeto: é um identificador que identifica inequivocamente o projeto, tipicamente o nome do mesmo;
- *Project Language* – Linguagem de programação do projeto: refere-se à linguagem de programação em que o projeto irá ser desenvolvido.

Os atributos de iteração e equipa são características que podem variar entre iterações e por isso devem ser avaliados no início de cada uma destas.

Atributos da iteração:

- *Iteration Number* – Numero da iteração: refere-se ao número da iteração no contexto do projeto;
- *Iteration Expected Velocity* – Velocidade esperada: refere-se à quantidade de *story points* que é esperado que a equipa consiga completar no decorrer da iteração;
- *Iteration Technical Debt* – Défice técnico: refere-se à quantidade de trabalho extra necessário que surge quando são utilizadas soluções mais rápidas de implementar a curto prazo em vez da melhor solução possível. Este atributo deve ser indicado em *story points* tendo por base uma *user story* cuja descrição é todo o trabalho necessário;
- *Iteration Duration* – Duração: refere-se à duração da iteração (em semanas).

Atributos da equipa:

- *Team Number Of Members*: Número de elementos: refere-se ao número de elementos da equipa que irão realizar os desenvolvimentos. Não devem ser contados neste atributo elementos da equipa que não participem no desenvolvimento, como por exemplo product owners;
- *Team Is Multi Site* – Múltiplas localizações: refere-se ao facto de a equipa estar ou não distribuída por várias localizações geográficas distintas;
- *Team Average Experience* – Experiencia média: refere-se à média da experiencia de todos os elementos da equipa de desenvolvimento (em anos);
- *Team Domain Knowledge* – Experiencia no domínio: refere-se ao nível de conhecimentos da equipa de desenvolvimento no domínio de conhecimentos específico em que o projeto se insere (como por exemplo banca, logística, saúde, etc.);
- *Team Architecture* – Arquitetura: refere-se ao nível de conhecimentos da equipa de desenvolvimento na arquitetura tecnológica que suporta o projeto;
- *Team Data Modeling* – Modelação de dados: refere-se ao nível de conhecimentos da equipa de desenvolvimento na modelação de dados;
- *Team Business Logic* – Lógica do negócio: refere-se ao nível de conhecimentos da equipa de desenvolvimento na lógica de negócio existente e/ou a ser desenvolvida no projeto;

- *Team Integrations* – Integrações: refere-se ao nível de conhecimentos da equipa de desenvolvimento em integrações com sistemas externos;
- *Team User Interface* – Interface: refere-se ao nível de conhecimentos da equipa de desenvolvimento no desenho de interfaces;
- *Team User Experience* – Experiencia do utilizador: refere-se ao nível de conhecimentos da equipa de desenvolvimento no desenho de sistemas de modo a proporcionar uma melhor experiencia de utilização aos seus utilizadores finais;
- *Team Performance* – Desempenho: refere-se ao nível de conhecimentos da equipa de desenvolvimento em processos e técnicas de otimização de desempenho de qualquer componente a ser desenvolvido.

Por fim, os atributos da *user story*, como o próprio nome indica, são atributos que caracterizam cada *user story*. Estes atributos devem ser avaliados durante o processo de estimativa de cada *user story*, por exemplo, durante as sessões de *Planning Poker*.

Atributos da *user story*:

- *User Story Story Points* – *Story points*: refere-se ao número de *story points* estimado para a *user story* em questão (estimados através de uma sessão de *Planning Poker*);
- *User Story Acceptance Criteria* – Critérios de aceitação: refere-se ao número de critérios de aceitação descritos na *user story* que devem ser cumpridos para que este possa ser considerada totalmente desenvolvida e consequentemente possa ser entregue ao cliente;
- *User Story Main Language* – Linguagem de programação: refere-se à linguagem de programação em que a *user story* irá ser desenvolvida. Tipicamente será igual à linguagem de programação do projeto (Project Language), porém em casos pontoais poderá ser necessário desenvolver componentes numa linguagem de programação diferente da linguagem principal do projeto;
- *User Story Internal Dependencies* – Dependências internas: número de dependências que são da responsabilidade da equipa de desenvolvimento, como por exemplo a precedência entre *user stories* (casos em que uma *user story* não pode ser desenvolvida sem antes ser desenvolvida outra *user story*);
- *User Story External Dependencies* – Dependências externas: número de dependências que são da responsabilidade de entidades externas ao projeto, como por exemplo detalhes da integração com um sistema externo;
- *User Story Is Technical* – *User story* técnica: indica se a *user story* em questão é uma *user story* técnica, isto é, se tem como principal objetivo desenvolver componentes que irão auxiliar o desenvolvimento de *user stories* funcionais (como por exemplo tarefas de modelação de dados, criação de camadas de interface com sistemas externos, alterações de arquitetura, *refactoring* de componentes existentes, etc.);

- *User Story Is Functional – User story* funcional: indica se a *user story* em questão é uma *user story* funcional, isto é, se tem como principal objetivo proporcionar uma funcionalidade aos utilizadores finais;
- *User Story Architecture* – Arquitetura: refere-se ao impacto que a *user story* em questão irá ter na arquitetura tecnológica que suporta o projeto;
- *User Story Data Modeling* – Modelação de dados: refere-se ao impacto que a *user story* em questão irá ter no modelo de dados existente;
- *User Story Business Logic* – Lógica do negócio: refere-se ao impacto que a *user story* em questão irá ter lógica de negócio existente;
- *User Story Integrations* – Integrações: refere-se ao impacto que a *user story* em questão irá ter nas integrações com sistemas externos existentes;
- *User Story User Interface* – Interface: refere-se ao impacto que a *user story* em questão irá ter na interface existente;
- *User Story User Experience* – Experiencia do utilizador: refere-se ao impacto que a *user story* em questão irá ter na experiencia de utilização dos utilizadores finais;
- *User Story Performance* – Desempenho: refere-se ao impacto que a *user story* em questão irá ter no desempenho de determinado componente do projeto.

Para além dos atributos anteriormente enunciados foi também recolhido o custo real de cada *user story* (*User Story Actual Hours*). Este valor representa o número de horas necessárias para desenvolver cada *user story*. A distribuição de *user stories* em relação ao seu custo real pode ser vista na Tabela 6-3.

Tabela 6-3: Distribuição de *user stories* por custo real

Custo Real (horas)	Número de User Stories
1-4	1173
5-8	832
9-12	403
13-16	340
17-20	186
21-24	41
25-28	45
29-32	47
33-36	30
37-40	21
41-44	9
53-56	1
Total:	3128

De modo a uniformizar o *dataset* construído a partir dos dados recolhidos cada atributo deve ser representado por um determinado tipo de valor. Os atributos *Project Name*, *Project Language* e *User Story Main Language* são representados por um texto livre (não vazio). Os atributos *Team Is Multi Site*, *User Story Is Technical* e *User Story Is Functional* são representados por um valor booleano (sim/não ou true/false tipicamente representado por um 0 ou um 1). O atributo *Team Average Experience* é representado por um valor decimal (superior ou igual a zero) e todos os restantes são representados por um valor inteiro (superior ou igual a zero).

Para auxiliar na representação de atributos que requerem a atribuição de um nível foram criadas duas escalas padronizadas. As escalas são compostas por seis níveis representados por valores inteiros a variar de 0 até 5 onde 0 é o nível mais baixo e 5 é o nível mais alto. A primeira escala representa vários níveis de experiencia que uma equipa pode possuir e deve ser utilizada nos atributos seguintes:

- *Team Domain Knowledge*
- *Team Architecture*
- *Team Data Modeling*
- *Team Business Logic*
- *Team Integrations*
- *Team User Interface*
- *Team User Experience*
- *Team Performance*

Os níveis de experiencia e as suas respetivas descrições podem ser vistas na Tabela 6-4.

Tabela 6-4: Níveis de Experiencia

Valor	Nível	Descrição
0	No Experience	Sem experiência.
1	Basic Knowledge	Conhecimento e compreensão de técnicas e conceitos básicos.
2	Limited Experiene	Experiência adquirida em sala de aula e/ou em cenários experimentais ou como estagiário.
3	Intermidiate	Consegue concluir com êxito tarefas, por vezes recorrendo à ajuda de um elemento sénior.
4	Proficient	Consegue executar tarefas de forma independente sem a necessidade de assistência.
5	Advanced	Especialista. Consegue fornecer orientação, solucionar problemas e responder questões relacionadas com a área de especialização.

A segunda escala representa os vários níveis de impacto que uma *user story* pode ter e deve ser utilizada nos atributos seguintes:

- *User Story Architecture*
- *User Story Data Modeling*
- *User Story Business Logic*
- *User Story Integrations*
- *User Story User Interface*
- *User Story User Experience*
- *User Story Performance*

Os níveis de impacto e as suas respetivas descrições podem ser vistas na Tabela 6-5.

Tabela 6-5: Níveis de impacto

Valor	Nível	Descrição
0	None	Sem impactos.
1	Insignificant	Impacto insignificante e alterações triviais.
2	Moderate	Impacto moderado e desenvolvimentos ou alterações simples. Alteração de componentes já existentes.
3	Significant	Impacto significante. Desenvolvimentos ou alterações complexas (tipicamente são a tarefas semelhantes a outras já efetuadas no contexto do projeto). Criação de componentes idênticos a outros já existentes.
4	Strong	Impacto forte. Desenvolvimentos ou alterações muito complexas e/ou em vários componentes do sistema (tipicamente são tarefas nunca antes realizadas no contexto do projeto). Criação de componentes novos.
5	System-Wide	Impactos globais, transversais a todo o sistema. Alterações necessárias em todos os componentes do sistema.

Nas primeiras utilizações da escala de impactos sugere-se a utilização de uma das seguintes abordagens como ponto de partida:

- Escolher uma tarefa mais simples e atribuir-lhe um dos valores mais baixos da escala;
- Escolher uma tarefa de dificuldade intermédia e atribuir-lhe um valor sensivelmente a meio da escala.

Este *dataset* for criado de modo a que seja de fácil interpretação e também para que a sua utilização em sessões de *Planning Poker* seja fácil e rápida. Um exemplo de um *user story* que compõe o *dataset* pode ser visto na Tabela 6-6.

Tabela 6-6: Exemplo de uma user story do dataset criado

Atributo	Valor	Atributo	Valor
ProjectName	P10	UserStoryStoryPoints	5 Story Points
ProjectLanguage	Outsystems	UserStoryAcceptanceCriteria	2
		UserStoryMainLanguage	Outsystems
IterationNumber	11	UserStoryInternalDependencies	0
IterationExpectedVelocity	110 Story Points	UserStoryExternalDependencies	0
IterationTechnicalDebt	8 Story Points	UserStoryIsTechnical	Não
IterationDuration	2 Semanas	UserStoryIsFunctional	Sim
		UserStoryArchitecture	None
TeamNumberOfMembers	3	UserStoryDataModeling	None
TeamIsMultiSite	Sim	UserStoryBusinessLogic	Significant
TeamAverageExperience	4.5 Anos	UserStoryIntegrations	None
TeamDomainKnowledge	Intermidiate	UserStoryUserInterface	Moderate
TeamArchitecture	Limited Experiene	UserStoryUserExperience	None
TeamDataModeling	Proficient	UserStoryPerformance	None
TeamBusinessLogic	Advanced	UserStoryActualHours	8 Horas
TeamIntegrations	Proficient		
TeamUserInterface	Proficient		
TeamUserExperience	Proficient		
TeamPerformance	Limited Experiene		

6.2 Detalhes Experimentais

A implementação dos modelos foi realizada recorrendo ao *Scikit Learn*. Para que o *dataset* possa ser interpretado pelo *Scikit Learn* foi necessário realizar um tratamento prévio dos dados. Após o tratamento inicial do *dataset* foram aplicadas diversas técnicas de *Machine Learning* ao mesmo. Cada técnica possui um conjunto de parâmetros (*Hyperparameters*) que devem ser definidos antes da sua aplicação. De modo a encontrar os valores ótimos dos parâmetros de cada modelo foi utilizado um dos seguintes métodos: *Grid Search* ou *Random Search*. Por fim os modelos criados foram validados de modo a avaliar o seu poder de generalização. Esta secção irá descrever com mais detalhe estes detalhes de implementação.

6.2.1 Scikit Learn

A implementação dos modelos foi realizada recorrendo ao *Scikit Learn*. O *Scikit Learn* é um módulo *Python* que disponibiliza uma vasta gama de algoritmos de *Machine Learning*. Este pacote foca-se na disponibilização de algoritmos de *Machine Learning* a indivíduos não especializados na área, recorrendo a uma linguagem de alto nível. É colocada uma enfase especial na facilidade de uso, no desempenho, na documentação e na consistência da API. O

Scikit Learn tem poucas dependências e é distribuída sob a licença BSD. O seu uso é incentivado em ambientes académicos e comerciais (Pedregosa et al., 2011).

A linguagem de programação *Python* tem vindo a estabelecer-se como uma das linguagens de programação mais populares na comunidade cientifica. Isto deve-se à sua natureza interativa de alto nível e ao seu ecossistema de bibliotecas científicas. Para além disto, é cada vez mais utilizada como uma linguagem de uso genérico, não só em contextos académicos, mas também em contextos industriais e comerciais (Pedregosa et al., 2011).

O *Scikit Learn* foi desenvolvido tento por base as bibliotecas *Python Numpy* e *Scipy*. A biblioteca *Numpy* proporciona a estrutura base utilizada para dados e parâmetros dos modelos. Os dados de entrada dos modelos são representados por matrizes *Numpy* o que torna a integração com outras bibliotecas científicas *Python* muito simples. Para além disto, a biblioteca disponibiliza também operações aritméticas básicas. A biblioteca *Scipy* disponibiliza algoritmos eficientes de álgebra linear, matrizes, e funções estatísticas básicas. O *Scipy* proporciona também ligações a vários pacotes baseados em *Fortran*. Isto é importante para a facilidade de instalação e a portabilidade pois a interação com bibliotecas *Fortran* revela-se complexo em algumas plataformas (Pedregosa et al., 2011).

É importante referir também que foi utilizada a biblioteca *Python pandas*. Também distribuído sob a licensa BSD, o pandas proporciona estruturas de dados, de alto desempenho de fácil utilização e manipulação (pandas, 2008).

De seguida são apresentadas as versões das bibliotecas anteriormente enunciadas que foram utilizadas para a implementação dos modelos:

- *Python*: v3.5.2
- *Scikit Learn*: v0.19.0
- *Numpy*: v1.11.0
- *Scipy*: v0.17.0
- *Pandas*: v0.17.1

Todo o código *Python* desenvolvido no âmbito da implementação dos modelos pode ser consultado no Anexo A.

6.2.2 Pré-Processamento de Dados

Geralmente, o *Scikit Learn* consegue trabalhar com quaisquer dados numéricos (scikit-learn, 2017), porém, muitos problemas de *Machine Learning* contêm atributos categóricos (Hackeling, 2014) que o *Scikit Learn* não consegue interpretar diretamente. Para que o *Scikit Learn* consiga interpretar corretamente este tipo de dados, estes devem ser alvos de um processamento prévio que os transforme em dados numéricos.

Existem dois tipos de dados categóricos: ordinais e nominais. Dados categóricos ordinais são dados que possuem uma ordem, como por exemplo os tamanhos de t-shirt visto que L > M > S. Por outro lado, dados categóricos nominais não possuem qualquer ordem nem faz sentido serem ordenados, como por exemplo cores. Geralmente não faz sentido afirmar que a cor A é maior que a cor B ou que a cor B é menor que a cor C (Raschka, 2016).

A maneira mais simples de tratar dados categóricos ordinais de modo a que o *Scikit Learn* os consiga interpretar corretamente é através da atribuição de um número inteiro a cada um destes de modo a preservar a sua ordem. Por exemplo, no caso dos tamanhos de t-shirt L, M e S podem ser representados por 3, 2 e 1 respetivamente. Não existe nenhuma funcionalidade do *Scikit Learn* que consiga derivar a ordenação correta dos dados e consequentemente esta tarefa terá que ser realizada manualmente (Raschka, 2016).

Quanto a dados categóricos nominais, apesar de a maioria dos algoritmos de aprendizagem existentes no *Scikit Learn* realizarem a sua conversão internamente para inteiros, é considerado uma boa prática realizar esta conversão previamente de modo a evitar problemas técnicos. Esta conversão pode ser realizada da forma apresentada anteriormente para os dados ordinais, porém neste caso a ordem não é relevante. Visto que ordem dos valores inteiros resultantes não é relevante esta conversão pode ser realizada recorrendo à classe *LabelEncoder* do *Scikit Learn* (Raschka, 2016; scikit-learn, 2017). Depois de realizar esta conversão no exemplo anterior das cores pode-se obter os seguintes dados:

- Cor A – 0
- Cor B – 1
- Cor C – 2

Apesar de os dados não possuírem qualquer tipo de ordem, ao utilizar estes dados num dos algoritmos de aprendizagem do *Scikit Learn* este irá assumir que a cor C é maior que a cor B e que a ambas são maiores que a cor A pois 2 é maior que 1 e tanto 2 como 1 são maiores que 0. Esta suposição é incorreta e poderá levar à produção de resultados incorretos (Raschka, 2016).

Para contornar este problema é geralmente utilizada uma técnica chamada *one-hot encoding*. Esta técnica consiste na utilização de uma variável binária para cada um dos valores possíveis de um determinado atributo. No exemplo anterior das cores o atributo Cor seria transformado em três novos atributos – Cor_A, Cor_B e Cor_C – e seria atribuído um valor binário (0 ou 1) a cada um destes. Por exemplo, para um registo da cor A seriam atribuídos os valore 1, 0 e 0 respetivamente. A aplicação desta técnica pode ser realizada recorrendo à classe *OneHotEncoder* do *Scikit Learn* (Hackeling, 2014; Raschka, 2016; scikit-learn, 2017).

No contexto do *dataset* utilizado na construção dos modelos (descrito na secção 6.1) foi necessário aplicar as transformações em alguns dos seus atributos. O *dataset* contem três atributos que possuem dados categóricos nominais (*Project Name*, *Project Language* e *User Story Main Language*) e quinze atributos que possuem dados categóricos ordinais (todos os

atributos que utilizem as escalas de experiencia ou de impacto). No caso dos atributos que possuem dados categóricos nominais foram aplicadas as duas técnicas referidas anteriormente por intermédio das classes *LabelEncoder* e *OneHotEncoder* do *Scikit Learn* (o código *Python* desenvolvido para este efeito pode ser pode ser consultado no Anexo A). Por outro lado, as transformações necessárias para os atributos que possuem dados categóricos ordinais já foi realizada indiretamente através da criação das escalas de experiencia e de impacto, bastando apenas utilizar o valor de cada nível em vez da sua representação escrita por extenso.

Outro passo importante no pré-processamento dos dados é a normalização dos mesmos. A grande maioria dos algoritmos implementados no *Scikit Learn* têm como requisito a utilização de dados normalizados (Raschka, 2016; scikit-learn, 2017). Se a escala de um atributo for ordens de grandeza maior do que a os restantes, este atributo pode dominar o algoritmo de aprendizagem e evitar que este aprenda com os restantes atributos. Isto pode acontecer quando por exemplo a maioria dos atributos são representados utilizando a escada de 0 a 10 e um único atributo é representado utilizando uma escala de 0 a 100 000 (Hackeling, 2014; Raschka, 2016; scikit-learn, 2017).

O termo normalização pode ter vários significados dependendo do autor e do contexto em que é utilizado (Raschka, 2016). Neste caso a realização desta tarefa é dividida nas duas abordagens seguintes: normalização euclidiana e estandardização.

A normalização euclidiana, também conhecida por normalização L2, consiste na transformação independente de cada amostra de um conjunto de dados de modo a que a sua norma euclidiana (ou norma L2) seja igual a um (Abdi, 2010; scikit-learn, 2017). A norma euclidiana de uma amostra x é representada por $\|x\|$ e é calculada através da raiz quadrada do somatório dos quadrados dos elementos da amostra (Abdi, 2010), como pode ser visto em (6-1).

$$\|x\| = \sqrt{\sum_{i=1}^{T} {x_i}^2} \qquad (6\text{-}1)$$

Onde x_i é o elemento i da amostra e T é o número total de valores da amostra. A normalização de uma amostras é obtida através da divisão de cada elemento desta pela norma euclidiana da amostra (Abdi, 2010), como pode ser visto em (6-2).

$$\widetilde{x_i} = \frac{x_i}{\|x\|} \qquad (6\text{-}2)$$

Onde x_i é o elemento i da amostra, $\|x\|$ é norma euclidiana da amostra e $\widetilde{x_i}$ é o valor normalizado do elemento i da amostra. Este tipo de normalização é especialmente útil em modelos que utilizem a distância euclidiana, como é o caso do *K-Nearest Neighbors* (scikit-learn, 2017).

A estandardização consiste na transformação para média nula e desvio padrão unitário de cada atributo de um conjunto de dados, isto é, os atributos tomam a forma de uma distribuição normal. Os valores estandardizados de um atributo podem ser obtidos através da divisão pelo desvio padrão de cada valor do atributo subtraído da média do mesmo (Abdi, 2010; Matos, 1995; Raschka, 2016; scikit-learn, 2017), como pode ser visto em (6-3).

$$z_i = \frac{x_i - \mu_x}{\sigma_x} \tag{6-3}$$

Onde x_i é o elemento i do atributo, μ_x é a média do atributo, σ_x é o desvio padrão do atributo e z_i é o valor estandardizado do elemento i do atributo.

Ambas estas abordagens (normalização euclidiana e estandardização) podem ser realizadas recorrendo às classes *Normalizer* e *StandardScaler* do *Scikit Learn* respetivamente (scikit-learn, 2017). Todas as técnicas de *Machine Learning* utilizadas na criação dos modelos foram precedidas por uma destas abordagens.

6.2.3 Hyperparameters

Os modelos de *Machine Learning* são basicamente funções matemáticas que representam os relacionamentos entre diversos aspetos dos dados. Por exemplo, um modelo de regressão linear utiliza uma reta para representar a relação entre os diversos atributos e a variável independente (Zheng, 2015). Esta relação pode ser genericamente representada por (6-4).

$$y = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n \tag{6-4}$$

Onde x_i representa os vários atributos e y é a variável independente, isto é, o valor a estimar. As variáveis w_i são os pesos de cada atributo. Estes valores são conhecidos como sendo parâmetros do modelo pois são aprendidos durante a fase de treino do mesmo. A fase de treino do modelo tem como objetivo determinar os melhores parâmetros do modelo para um determinado conjunto de dados (Raschka, 2016; Zheng, 2015).

Para além dos parâmetros do modelo, existe outro tipo de parâmetros denominados de *hyperparameters*. Os *hyperparameters* são parâmetros que não são determinados diretamente pelos modelos (Raschka, 2016; scikit-learn, 2017; Zheng, 2015). A regressão linear não tem qualquer *hyperparameter*, porém algoritmos como DT necessitam de *hyperparameters* que indiquem a profundidade e o número de folhas da árvore. O próprio processo de treino de certos modelos por vezes recorre à otimização de uma determinada função. Neste processo podem ser utilizadas técnicas de otimização matemáticas que por sua vez também podem ter *hyperparameters* (Zheng, 2015).

A definição dos *hyperparameters* pode ter um grande impacto na precisão dos modelos. Muitas vezes os valores ótimos dos *hyperparameters* variam dependendo do *dataset* utilizado e, portanto, devem ser definidos com base neste. Como o processo de treino do modelo não define *hyperparameters* existe a necessidade de um processo auxiliar que os otimize. Este processo de meta-otimização é denominado de *hyperparameter tuning*. Cada tentativa de otimização de um *hyperparameter* envolve a criação e o treino de um modelo interno para que seja possível realizar a otimização. O resultado desta otimização é o melhor valor para o *hyperparameter* em questão (scikit-learn, 2017; Zheng, 2015).

Depois de avaliar vários valores possíveis para os *hyperparameters*, o processo de meta-otimização destes devolve a configuração que produz o modelo com o melhor desempenho. Esta configuração é posteriormente utilizada para treinar o modelo final (Zheng, 2015).

Existem duas principais formas de realizar *hyperparameter tuning*: *grid search* e *random search*. Dado um conjunto de valores possíveis para os *hyperparameters* de um modelo o *grid search* irá realizar uma procura exaustiva, criando e avaliando modelos internos para todas as combinações possíveis. Por outro lado, o *random search* irá escolher n combinações aleatórias (onde n é definido à priori). Apenas serão criados e avaliados modelos para um número restrito de combinações, evitando assim a necessidade de avaliar todas as combinações possíveis. A grande vantagem do *random search* é que ao reduzir o número de combinações a avaliar consegue-se reduzir drasticamente o tempo de *hyperparameter tuning* (Bergstra e Bengio, 2012; Raschka, 2016; scikit-learn, 2017; Zheng, 2015).

Todas as técnicas de *Machine Learning* utilizadas na criação dos modelos foram precedidas por uma destas abordagens. A abordagem utilizada depende principalmente da quantidade de *hyperparameters* de cada técnica e consequentemente do tempo necessário para avaliar todas as combinações possíveis.

6.2.4 Validação dos modelos

Uma das etapas mais importantes da construção de um modelo é a avaliação do seu desempenho em dados nunca antes vistos por este, isto é, avaliar a sua capacidade de generalização (Raschka, 2016). A forma mais simples e mais utilizada de realizar esta avaliação é o *cross-validation* (Hastie, Tibshirani e Friedman, 2009).

Cross-validation é uma técnica de avaliação e comparação de modelos através da divisão do *dataset* utilizado em dois subconjuntos. Um é utilizado para treinar o modelo e o outro é utilizado para o validar. Tipicamente, os dados de treino e teste devem ser cruzados em rodas sucessivas de modo a que todos os dados sejam utilizados na validação do modelo (James et al., 2013).

Esta técnica tem principalmente dois objetivos (James et al., 2013):

- Avaliar a capacidade de generalização de um modelo;
- Comparar o desempenho de dois ou mais modelos diferentes de modo a encontrar o melhor modelo para o *dataset* utilizado ou comprar o desempenho de duas ou mais variantes do mesmo modelo que utilizem diferentes configurações de *hyperparameters*.

O *cross-validation* é maioritariamente aplicado em três contextos (James et al., 2013):

- Avaliação de desempenho do modelo;
- Seleção do modelo (*model selection* – processo de selecionar o melhor modelo ou tipo de modelo para um determinado *dataset* (Zheng, 2015));
- *Hyperparameter tuning*.

A Figura 6-1 mostra de uma maneira simplificada estes três contextos.

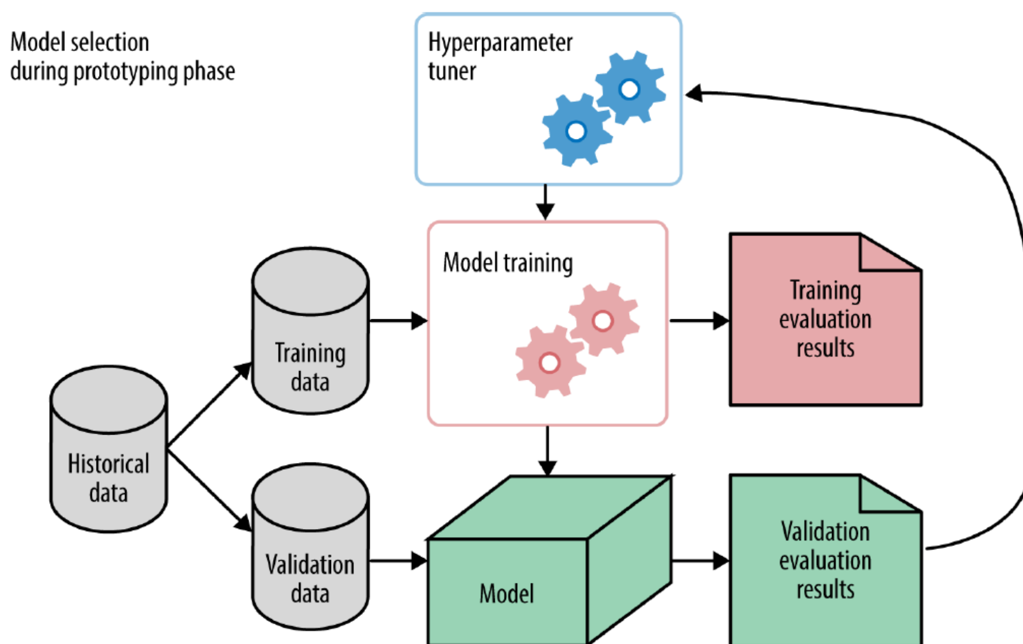

Figura 6-1: Aplicação de *cross-validation* na prototipagem de um modelo
Fonte: Evaluating Machine Learning Models (Zheng, 2015)

Pode ser visto que os dados históricos são divididos em duas partes: dados de treino e dados de teste. O processo de treino do modelo utiliza os dados de treino para produzir um modelo que é avaliado utilizando os dados de teste. Os resultados desta validação são retornados ao processo de *hyperparameter tuning* que muda a configuração dos *hyperparameters* e o processo de treino é executado novamente (Zheng, 2015).

Duas das técnicas de *cross-validation* mais utilizadas são o *holdout cross-validation* e o *k-fold cross-validation*. No *holdout cross-validation*, o mais simples dos dois, o *dataset* inicial é

dividido em dois subconjuntos. Um é utilizado para treinar o modelo e o outro é utilizado para o validar (Raschka, 2016; scikit-learn, 2017; Zheng, 2015). Porém, normalmente é também necessário avaliar diferentes configurações de *hyperparameters* de modo a maximizar o desempenho do modelo em dados novos (Raschka, 2016). Uma melhor forma de aplicar o *holdout cross-validation* é através da divisão do *dataset* inicial em três subconjuntos: dados de treino, de teste e de validação. Os dados de treino são utilizados para treinar o modelo e os dados de validação são utilizados para realizar a seleção do modelo. A vantagem de existir um terceiro subconjunto é que este possui dados nunca antes vistos pelo modelo o que permite obter uma melhor avaliação da capacidade de generalização do modelo (Raschka, 2016; scikit-learn, 2017). A Figura 6-2 mostra um exemplo genérico de utilização do *holdout cross-validation*.

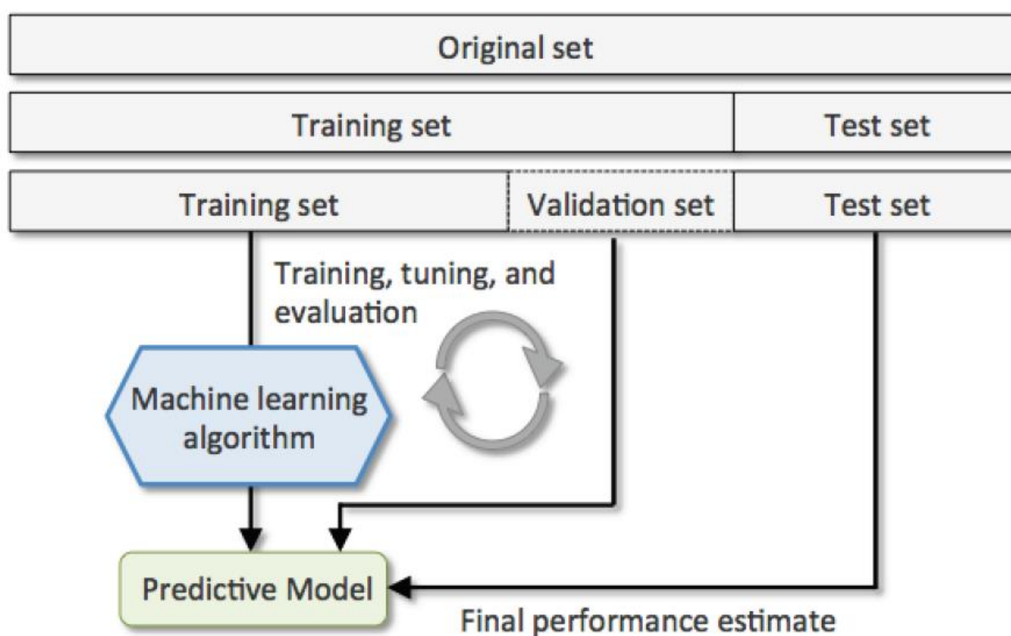

Figura 6-2: ***Holdout cross-validation***
Fonte: Python Machine Learning (Raschka, 2016)

Como pode ser visto, os dados de validação são utilizados repetidamente para avaliar o desempenho do modelo depois de este ser trainado utilizando várias configurações de *hyperparameters*. Uma vez determinados os melhores *hyperparameters* para o modelo, a sua capacidade de generalização é avaliada utilizando os dados de teste (Raschka, 2016).

No *k-fold cross-validation* o *dataset* inicial é dividido de forma aleatória em k partes (*folds*) onde $k - 1$ partes são utilizadas como dados de treino e a parte restante é utilizada como dados de teste. Este processo é repetido k vezes de modo a obter k modelos, cada um com a sua avaliação de desempenho. Posteriormente é feita a média aritmética de todas as avaliações para obter uma avaliação menos sensível à divisão do *dataset* (Raschka, 2016; scikit-learn, 2017). O valor típico de k é 10, pois este é um valor razoável para a grande maioria dos problemas

(Raschka, 2016). A Figura 6-3 mostra um exemplo genérico de utilização do *k-fold cross-validation*.

Figura 6-3: ***k-fold cross-validation***
Fonte: Python Machine Learning (Raschka, 2016)

Como pode ser visto, o *dataset* é divido em 10 partes e durante as 10 iterações, 9 destas são utilizadas para treinar o modelo e outra para avaliar o mesmo. Depois de executadas todas as iterações, os erros individuais de cada iteração são utilizados para calcular o erro médio dos modelos (Raschka, 2016; scikit-learn, 2017).

A utilização de *k-fold cross-validation* em conjunto com *grid search* ou *random search* permite aperfeiçoar o desempenho de um modelo através da otimização dos *hyperparameters* do mesmo. A esta abordagem pode-se juntar também a seleção do modelo através da comparação de várias técnicas de *Machine Learning*. Esta utilização hierárquica do *k-fold cross-validation* é denominada de *nested cross-validation* (Raschka, 2016).

No *nested cross-validation* existem dois níveis de *k-fold cross-validation*. O nível exterior irá dividir o *dataset* inicial em dados de treino e teste e o nível interior irá realizar a otimização dos hyperparameters através da aplicação de k-fold cross-validation nos dados de treino provenientes do nível exterior (Raschka, 2016; scikit-learn, 2017). A Figura 6-4 mostra um exemplo genérico de utilização de nested *k-fold cross-validation* (com 5 folds exteriores e 2 folds interiores).

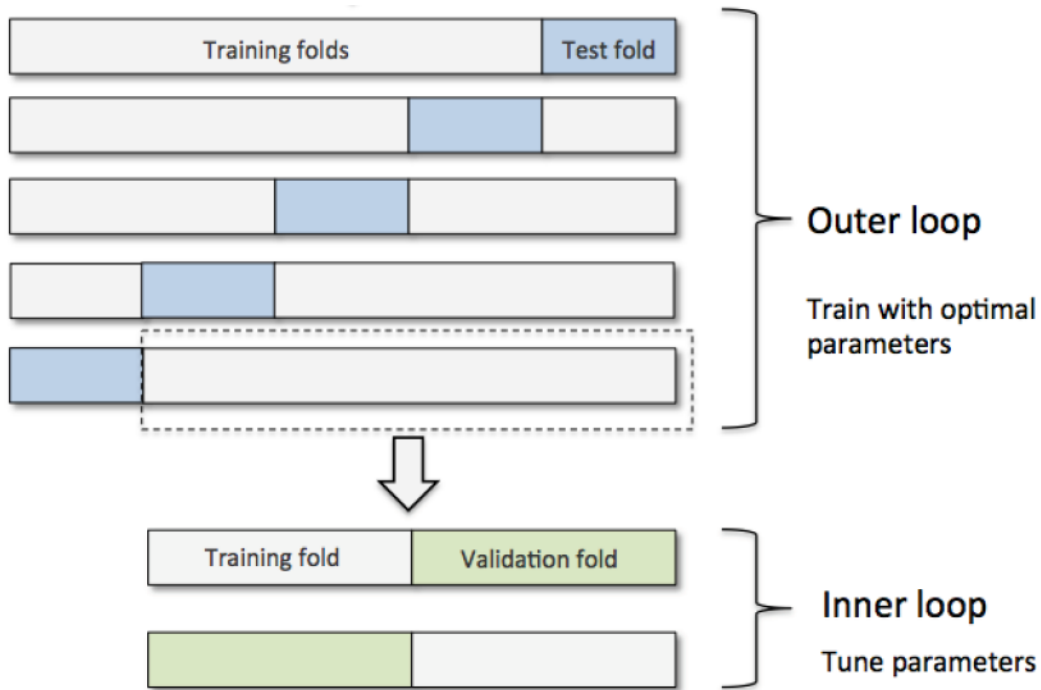

Figura 6-4: ***Nested k-fold cross-validation***
Fonte: Python Machine Learning (Raschka, 2016)

Também no caso de *nested k-fold cross-validation* os erros individuais de cada iteração são utilizados para calcular o erro médio de cada modelo. Este erro indica a capacidade de generalização de cada modelo (Raschka, 2016).

Todas as técnicas de *Machine Learning* utilizadas na criação dos modelos foram validadas utilizando *nested k-fold cross-validation*. Isto permite não só otimizar os *hyperparameters*, mas também comparar a capacidade de generalização dos diferentes modelos.

6.3 Metodologia

O modelo proposto foi implementado recorrendo às seguintes técnicas de *Machine Learning*:

- *Linear Regression*
- *Decision Trees*
- *K-Nearest Neighbors*
- *Support Vector Regression*
- *Multi Layer Perceptron*
- *Bagging*
- *Random Forest*
- *Extremely Randomized Trees*
- *Gradient Boosting*

A implementação do modelo proposto deu origem a vários modelos que foram posteriormente avaliados e comparados entre si. Esta comparação serviu fundamentalmente para aferir a viabilidade do modelo proposto para a estimativa de esforço de desenvolvimento de *software* em ambientes ágeis. O diagrama apresentado na Figura 6-5 indica as várias etapas da implementação do modelo proposto.

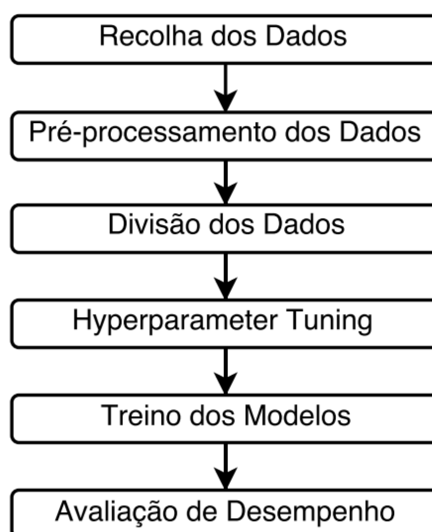

Figura 6-5: Etapas da metodologia utilizada

De seguida é apresentada uma descrição de alto nível de cada etapa da metodologia utilizada.

1. Recolha dos dados: os dados recolhidos foram utilizados na criação do *dataset* já descrito na secção 6.1. Este foi o *dataset* utilizado na construção dos modelos;

2. Pré-processamento dos dados: antes dos dados recolhidos poderem ser utilizados na construção dos modelos estes foram alvo de algumas transformações. Primeiramente os dados categóricos serão transformados em dados numéricos e por posteriormente todo o *dataset* foi normalizado por intermédio da normalização euclidiana ou da estandardização. Os dados utilizados na maioria dos modelos foram normalizados recorrendo à estandardização, sendo a única exceção os dados dos modelos baseados na técnica *K-Nearest Neighbors*. Neste caso os dados foram normalizados recorrendo à normalização euclidiana pois esta é especialmente útil em modelos que utilizem a distância euclidiana, como é o caso do *K-Nearest Neighbors*;

3. Divisão dos dados: o *dataset* utilizado foi subdividido em várias partes utilizando *k-fold cross-validation* (com $k = 10$);

4. *Hyperparameter tuning*: a otimização dos *hyperparameters* foi realizada recorrendo a *k-fold cross-validation* (com $k = 10$) em combinação com *grid search* ou *random search*, conseguindo-se assim obter *nested cross-validation*. A técnica a utilizar

depende principalmente da quantidade de *hyperparameters* e consequentemente do tempo necessário para avaliar todas as combinações possíveis. Se o número de configurações possíveis for inferior a 20 foi utilizado grid search, caso contrário foi utilizado random search com uma seleção aleatória de 20 configurações;

5. Treino do modelo: baseado nos resultados da etapa anterior foram criados modelos utilizando os melhores *hyperparameters* encontrados para cada iteração do *k-fold cross-validation* e técnica de *Machine Learning* utilizada. Estes modelos foram treinados utilizando os dados de treino provenientes da divisão do *dataset*;

6. Previsão do esforço: depois de treinado cada modelo tentou prever o esforço respeitante aos dados de treino provenientes da divisão do *dataset*;

7. Avaliação de desempenho: o esforço estimado pelos modelos foi comparado com o esforço real dos dados de treino provenientes da divisão do *dataset* de modo a avaliar o desempenho dos mesmos. A avaliação dos modelos foi realizada recorrendo aos critérios de avaliação MAE, MMRE e PRED(25).

À exceção da recolha de dados toda a implementação do modelo proposto foi desenvolvida em *Python* recorrendo ao *Scikit Learn*. O código *Python* desenvolvido para este efeito pode ser pode ser consultado no Anexo A.

6.4 Resultados

A implementação do modelo proposto, segundo a metodologia apresentada na secção anterior, produziu os resultados apresentados na Tabela 6-7.

Tabela 6-7: Resultados do modelo proposto

Técnica	MAE	MMRE	PRED(25)
Linear Regression	2,295	47,200	46,485
Decision Trees	1,453	21,404	69,566
K-Nearest Neighbors	2,027	34,242	55,274
Support Vector Regression	2,111	37,022	53,389
Multi Layer Perceptron	1,717	30,026	59,847
Bagging	1,123	18,490	76,024
Random Forest	1,214	20,154	74,521
Extremely Randomized Trees	1,147	18,698	75,320
Gradient Boosting	1,146	19,453	75,224

A Tabela 6-7 apresenta os resultados referentes à avaliação do desempenho dos modelos criados utilizando os critérios de avaliação MAE, MMRE e PRED(25). Estes valores correspondem à média aritmética das avaliações dos modelos criados em cada iteração do *k-fold*

cross-validation. Valores menores de MAE e MMRE e maiores de PRED(25) indicam estimativas mais precisas.

Em termos gerais pode observar-se que os modelos baseados em técnicas *ensemble* (*Bagging*, *Random Forest*, *Extremely Randomized Trees* e *Gradient Boosting*) superam os restantes nos três critérios de avaliação utilizados. De modo a obter uma melhor representação dos resultados, foram criados os gráficos apresentados na Figura 6-6, Figura 6-7 e Figura 6-8.

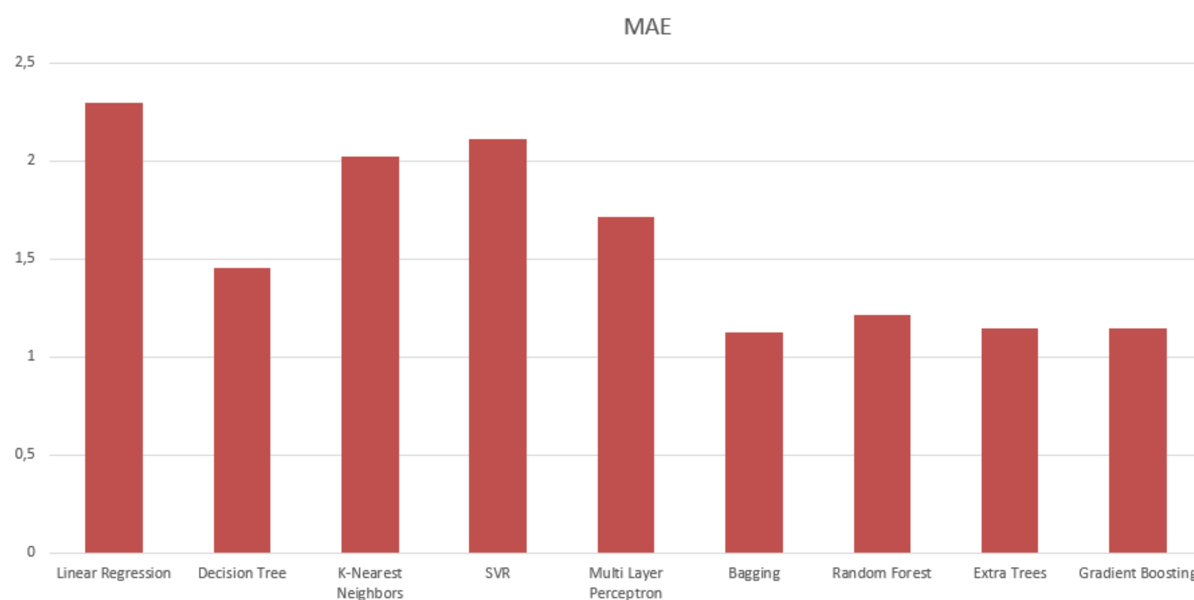

Figura 6-6: Resultados do modelo proposto – MAE

A Figura 6-6 apresenta os resultados da avaliação do desempenho dos modelos correspondestes ao critério de avaliação MAE. O erro médio absoluto dos modelos varia desde cerca de 1.1 horas até cerca de 2.3 horas, correspondendo aos modelos baseados em *Bagging* e *Linear Regression* respetivamente. Os modelos baseados em técnicas *ensemble* apresentam os melhores resultados enquanto os modelos baseados em *Linear Regression*, *Support Vector Regression* e *K-Nearest Neighbors* apresentam os piores resultados.

Dependendo do tamanho da *user story* em análise estes resultados podem ser mais ou menos positivos. Segundo a Tabela 6-3, cerca de 37.5% das *user stories* que compõem o *dataset* utilizado têm um custo real entre 1 e 4 horas. Para estas *user stories* um desvio de 2 horas na sua estimativa pode representar um erro de 50% a 200% em relação ao seu custo real. De forma a compreender melhor estes valores, o desempenho dos modelos foi de seguida avaliado utilizando dois critérios de avaliação relativos.

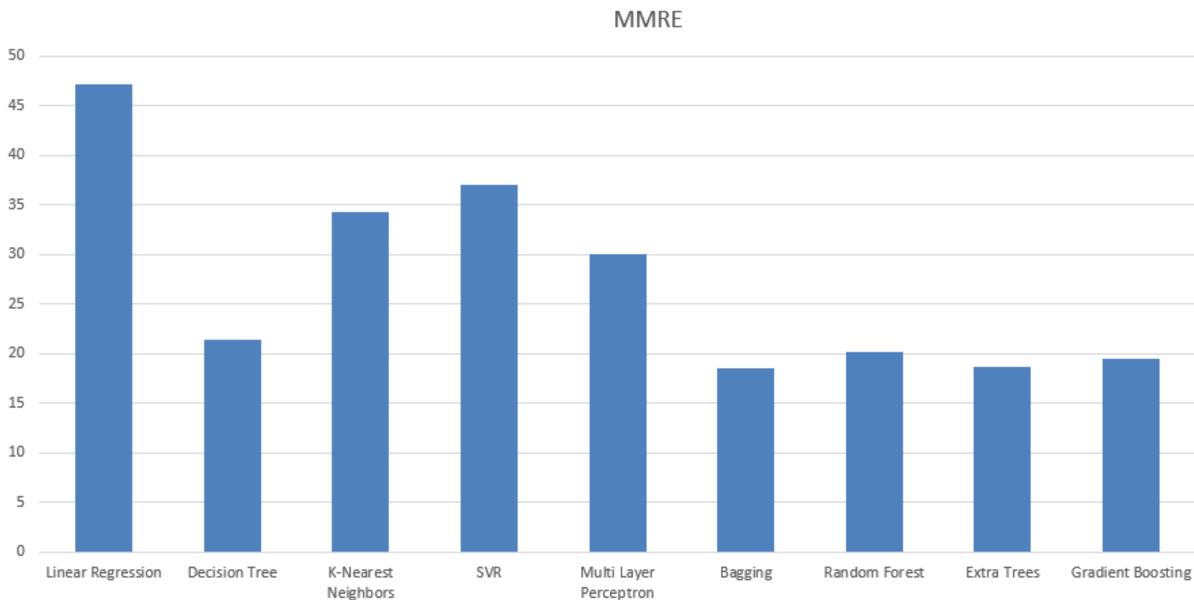

Figura 6-7: Resultados do modelo proposto – MMRE

A Figura 6-7 apresenta os resultados da avaliação do desempenho dos modelos correspondestes ao critério de avaliação MMRE. A média dos erros relativos, entre o valor real e o valor estimado, dos modelos varia desde cerca de 18.5% até cerca de 47.2%, correspondendo novamente aos modelos baseados em *Bagging* e *Linear Regression* respetivamente. Os modelos baseados em técnicas *ensemble* apresentaram novamente os melhores resultados, seguidos de perto pelos modelos baseados em *Decision Trees*. Os modelos baseados em *Linear Regression*, *Support Vector Regression* e *K-Nearest Neighbors* continuam a apresentar os piores resultados, porém com um destaque especial para os modelos baseados em *Linear Regression* que apresentaram resultados consideravelmente piores que todos os modelos restantes.

A utilização de um critério de avaliação relativo, como é o caso do MMRE, permitiu clarificar alguns dos resultados obtidos com o critério de avaliação absoluto MAE. Destaca-se o desempenho dos modelos baseados em *Linear Regression*. Este é consideravelmente pior do que era indicado pelo MAE, apresentando erros médios de cerca de 47.2% em relação ao custo real. Por outro lado, os modelos baseados em técnicas *ensemble* e ainda *Decision Trees* apresentaram resultados muito positivos tendo obtido resultados inferiores a 25% (resultado a partir do qual um modelo é geralmente considerado aceitável (Conte, Dunsmore e Shen, 1986; Fedotova, Teixeira e Alvelos, 2013; Wen et al., 2012)).

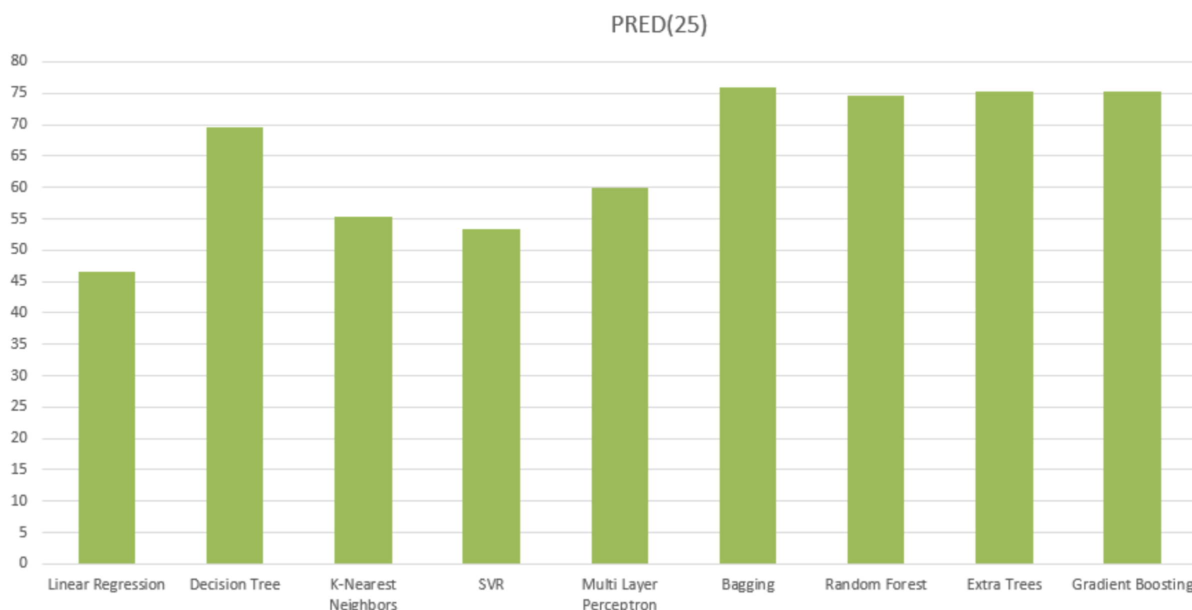

Figura 6-8: Resultados do modelo proposto – PRED(25)

A Figura 6-8 apresenta os resultados da avaliação do desempenho dos modelos correspondestes ao critério de avaliação PRED(25). A percentagem de estimativas cujos erros relativos não são maiores que 25% varia desde cerca de 46.5% até cerca de 76%, correspondendo mais uma vez aos modelos baseados em *Bagging* e *Linear Regression* respetivamente. Os modelos baseados em técnicas *ensemble* apresentaram novamente os melhores resultados, continuando a ser seguidos pelos modelos baseados em *Decision Trees*. Os modelos baseados em *Linear Regression*, *Support Vector Regression* e *K-Nearest Neighbors* apresentaram novamente os piores resultados, com um especial destaque mais uma vez para os modelos baseados em *Linear Regression* que apresentaram resultados consideravelmente piores que todos os modelos restantes.

A utilização do critério de avaliação PRED(25) permitiu confirmar os resultados obtidos anteriormente com o MMRE. Os modelos baseados em *Linear Regression* apresentaram claramente o pior desempenho de qualquer uma das técnicas utilizadas, obtendo um PRED(25) de apenas 46.5%. Por outro lado, confirmou-se os resultados muito positivos dos modelos baseados em técnicas *ensemble*, tendo três destes obtido resultados superiores a 75% (resultado a partir do qual um modelo é geralmente considerado aceitável (Conte, Dunsmore e Shen, 1986; Fedotova, Teixeira e Alvelos, 2013; Wen et al., 2012)).

6.5 Análise Comparativa

Os resultados do modelo proposto, apresentados na secção anterior, foram comparados com os resultados de estudos anteriores, apresentados na Tabela 4-3 secção 4.1. É importante relembrar que todos os estudos anteriores utilizaram um *dataset* de dados de 21 projetos (Z. K. Zia, Tipu

e Zia, 2012). Esta comparação apenas foi realizada para as técnicas *Decision Trees*, *Support Vector Regression*, *Multi Layer Perceptron*, *Random Forest* e *Gradient Boosting*. A comparação das restantes técnicas não foi possível pois não foram encontrados estudos anteriores que utilizassem as mesmas no âmbito da estimativa de esforço de desenvolvimento de *software* em ambientes ágeis.

Os modelos propostos utilizando *Decision Trees* e *Random Forest* obtiveram um desempenho superior aos modelos propostos por Satapathy (Satapathy, 2016) e Satapathy e Rath (Satapathy e Rath, 2017). Os modelos propostos utilizando *Support Vector Regression* obtiveram um desempenho superior ao pior modelo de Satapathy (Satapathy, 2016), porém inferiores ao melhor modelo deste. Por outro lado, os modelos propostos utilizando *Multi Layer Perceptron* e *Gradient Boosting* obtiveram um desempenho inferior aos modelos propostos por Panda (Panda, 2015), Panda et al. (Panda, Satapathy e Rath, 2015) e Satapathy e Rath (Satapathy e Rath, 2017).

Esta comparação pode não ser totalmente fiável pois os *datasets* utilizados são completamente diferentes. O *dataset* utilizado pelos estudos referenciados nesta comparação continha dados de 3 atributos de 21 projetos. Como já foi referido na secção 4.1, alguns dos autores (Panda, Satapathy e Rath, 2015; Satapathy, 2016; Satapathy e Rath, 2017) destes estudos manifestaram a sua preocupação com o tamanho muito reduzido do *dataset* utilizado, pois este pode influenciar o poder de generalização dos modelos propostos.

De modo a analisar o potencial problema da utilização de um *dataset* demasiado pequeno, foram criados 10 *datasets*, de 30 *user stories* cada, a partir do *dataset* apresentado na secção 6.1. O modelo proposto foi executado novamente para cada um destes 10 *datasets* mais pequenos. Um resumo dos resultados produzidos por estes 10 *datasets* é apresentado na Tabela 6-8. Os resultados completos de cada *dataset* podem ser vistos no Anexo B.

Tabela 6-8: Resultados do modelo proposto (*datasets* de dimensão reduzida)

	MAX			MMRE			PRED(25)		
Técnica	Média	Min	Max	Média	Min	Max	Média	Min	Max
Linear Regression	5,331	0,879	19,822	91,412	6,590	237,203	48,000	20,000	86,667
Decision Trees	1,732	0,850	3,838	26,529	6,111	52,774	62,500	26,667	86,667
K-Nearest Neighbors	2,007	1,084	3,372	29,848	8,611	44,434	52,167	26,667	80,000
SVR	1,771	0,926	2,721	28,052	9,805	42,429	62,333	43,333	90,000
Multi Layer Perceptron	2,255	1,417	3,391	32,946	19,199	47,614	55,333	36,667	73,333
Bagging	1,548	0,840	2,807	23,133	6,778	39,893	69,500	43,333	86,667
Random Forest	1,605	0,997	3,144	24,434	7,778	41,864	68,500	43,333	90,000
Extremely Rand. Trees	1,551	0,785	2,943	24,192	6,019	39,312	67,167	40,000	86,667
Gradient Boosting	1,469	0,736	2,917	21,228	4,875	37,298	70,167	43,333	90,000

A Tabela 6-8 apresenta um resumo dos resultados referentes à avaliação do desempenho dos modelos criados utilizando 10 *datasets* de dimensão reduzida. É apresentado o valor médio,

mínimo e máximo para cada um dos critérios de avaliação utilizados – MAE, MMRE e PRED(25).

Em termos gerais pode observar-se que a utilização de vários *datasets* pode causar uma grande variação no desempenho obtido pelos modelos criados. De modo a obter uma melhor representação dos resultados, foram criados os gráficos apresentados na Figura 6-9, Figura 6-10 e Figura 6-11.

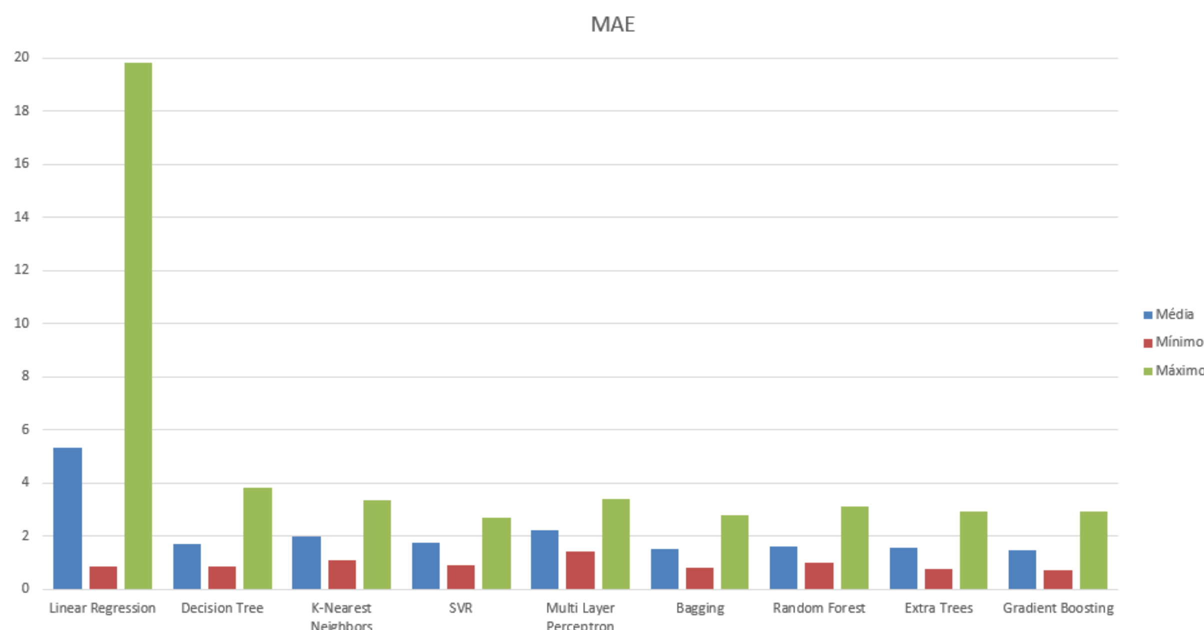

Figura 6-9: Resultados do modelo proposto (*datasets* de dimensão reduzida) – MAE

A Figura 6-9 apresenta o valor médio, mínimo e máximo respeitante aos resultados da avaliação do desempenho, dos modelos criados utilizando 10 *datasets* de dimensão reduzida, correspondestes ao critério de avaliação MAE. O erro médio absoluto dos modelos varia desde cerca de 0.7 horas até cerca de 19.8 horas. O desvio entre os valores mínimos e os valores máximos é bastante claro, especialmente no caso de *Linear Regression*, em que o erro médio absoluto pode chegar muito perto das 20 horas.

Os valores mínimos e máximos são respetivamente inferiores e superiores aos valores obtidos com o *dataset* completo. Por outro lado, os valores médios assemelham-se mais aos valores obtidos com o *dataset* completo. Como nem sempre os valores absolutos são a melhor forma de avaliar o desempenho de um modelo, de seguida os modelos foram avaliados utilizando dois critérios de avaliação relativos.

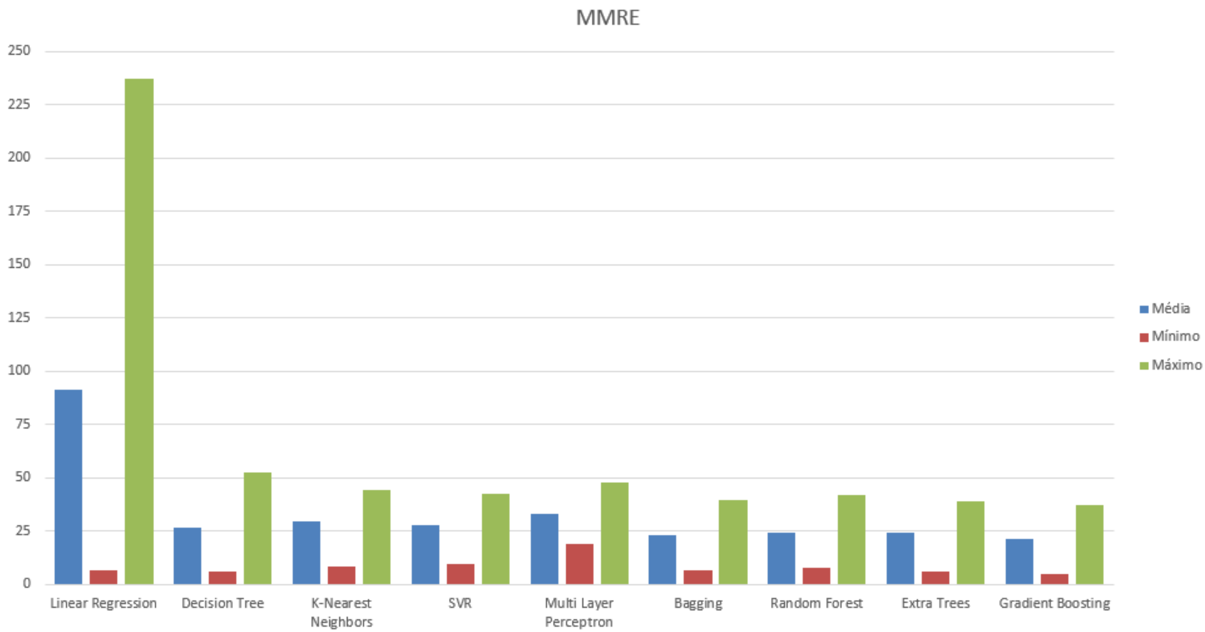

Figura 6-10: Resultados do modelo proposto (*datasets* de dimensão reduzida) – MMRE

A Figura 6-10 apresenta o valor médio, mínimo e máximo respeitante aos resultados da avaliação do desempenho, dos modelos criados utilizando 10 *datasets* de dimensão reduzida, correspondestes ao critério de avaliação MMRE. A média dos erros relativos, entre o valor real e o valor estimado, dos modelos varia desde cerca de 4.9% até cerca de 237,2%. Estes valores vão ao encontro dos valores apresentados para o critério de avaliação MAE, continuando em destaque a vasta imprecisão de alguns modelos baseados em *Linear Regression* que produziram estimativas com um erro relativo muito superior a 200%. Pode-se também observar que todos as técnicas, em pelo menos um *dataset*, conseguiram obter resultados inferiores a 25% (resultado a partir do qual um modelo é geralmente considerado aceitável (Conte, Dunsmore e Shen, 1986; Fedotova, Teixeira e Alvelos, 2013; Wen et al., 2012)).

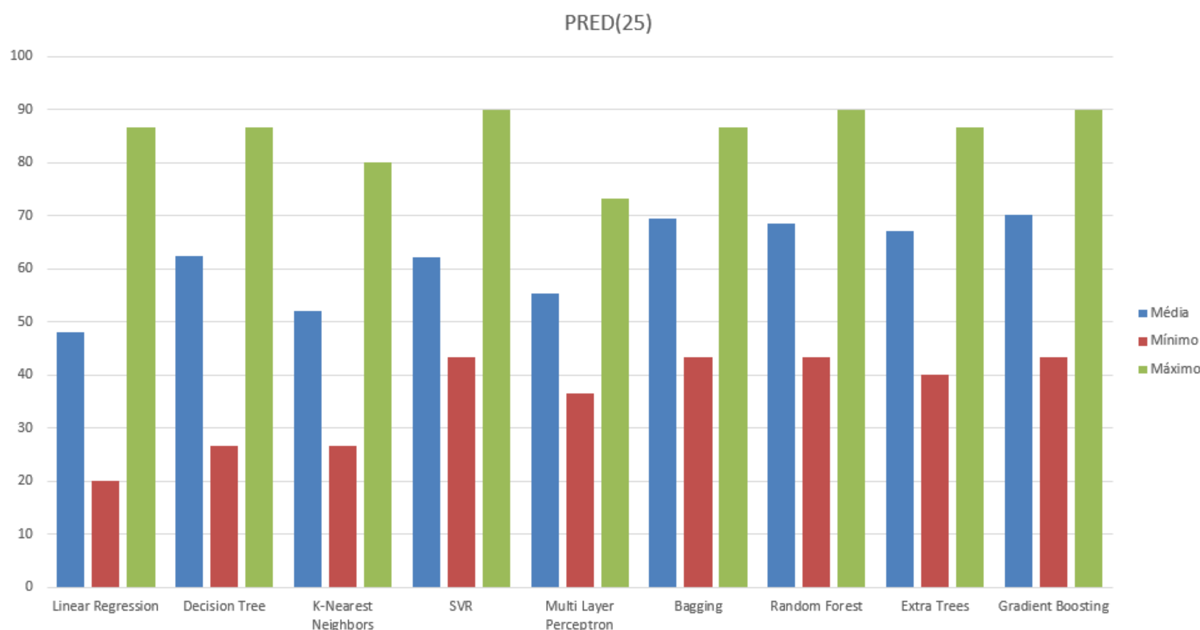

Figura 6-11: Resultados do modelo proposto (*datasets* de dimensão reduzida) – PRED(25)

A Figura 6-11 apresenta o valor médio, mínimo e máximo respeitante aos resultados da avaliação do desempenho, dos modelos criados utilizando 10 *datasets* de dimensão reduzida, correspondestes ao critério de avaliação PRED(25). A percentagem de estimativas cujos erros relativos não são maiores que 25% varia desde cerca de 20% até cerca de 90%. Pode-se também observar que todos as técnicas, exceto *Multi Layer Perceptron*, em pelo menos um *dataset*, conseguiram obter resultados superiores a 75% (resultado a partir do qual um modelo é geralmente considerado aceitável (Conte, Dunsmore e Shen, 1986; Fedotova, Teixeira e Alvelos, 2013; Wen et al., 2012)).

Utilizando o critério de avaliação PRED(25) não é tao percetível a magnitude da imprecisão identificada pelos critérios MAE e MMRE nos modelos baseados em *Linear Regression*. Isto pode ser devido ao facto do PRED(x) identificar modelos que são geralmente precisos porém podem ocasionalmente ser largamente imprecisos (Shepperd e Schofield, 1997).

6.6 Sumário

O modelo proposto propõe a utilização de nove técnicas de *Machine Learning* (*Linear Regression*, *Decision Trees*, *K-Nearest Neighbors*, *Support Vector Regression*, *Multi Layer Perceptron*, *Bagging*, *Random Forest*, *Extremely Randomized Trees* e *Gradient Boosting*) para aumentar a precisão das estimativas de esforço de desenvolvimento de *software* em ambientes ágeis. Ao contrário da maioria dos modelos já existentes na literatura o modelo proposto utiliza dados relativos a *user stories* individuais em vez de projetos completos.

O único *dataset* encontrado com dados provenientes de ambientes de desenvolvimento ágeis utiliza dados de projetos completos, criando a necessidade de criação de um novo *dataset*. Foram recolhidos de forma anónima dados relativos a 3128 *user stories* de 11 projetos desenvolvidos utilizando metodologias ágeis. Todas as *user stories* recolhidas foram estimadas em *story points* através do método *Planning Poker*. Foi colocada uma enfase especial na facilidade de interpretação e uso do *dataset*, nomeadamente em sessões de *Planning Poker*.

A implementação do modelo proposto foi desenvolvida em *Python* recorrendo ao *Scikit Learn*, tendo sido dividida em sete grandes etapas: Recolha dos dados, Pré-processamento dos dados, Divisão dos dados, *Hyperparameter tuning*, Treino do modelo, Previsão do esforço e Avaliação de desempenho. Depois de recolhidos os dados, estes foram alvo de um pré-processamento prévio de modo a que pudessem ser interpretados pelo *Scikit Learn*. De seguida foram subdivididos em várias partes com o objetivo de separar os dados utilizados no treino dos modelos e os dados utilizados na avaliação do mesmo. Isto permite uma melhor avaliação do poder de generalização dos modelos produzidos. Uma vez divididos e antes de serem treinados, foi realizada a otimização dos *hyperparameters* das várias técnicas de *Machine Learning*. Depois de treinados, cada modelo produziu uma previsão da estimativa de esforço para as *user stories* presentes no subconjunto de dados de teste. Estas previsões foram avalizadas recorrendo a três critérios de avaliação de desempenho – MAE, MMRE e PRED(25).

A partir dos resultados da avaliação do desempenho dos modelos pode-se observar que os modelos baseados em técnicas *ensemble* (*Bagging*, *Random Forest*, *Extremely Randomized Trees* e *Gradient Boosting*) superam os restantes nos três critérios de avaliação utilizados. Por outro lado, os modelos baseados e *Linear Regression* destacam-se de forma negativa, tendo obtido resultados negativos em todos os critérios utilizados.

Os modelos baseados em técnicas *ensemble* e ainda *Decision Trees* apresentaram resultados muito positivos tendo obtido resultados inferiores a 25% no critério de avaliação MMRE (resultado a partir do qual um modelo é geralmente considerado aceitável (Conte, Dunsmore e Shen, 1986; Fedotova, Teixeira e Alvelos, 2013; Wen et al., 2012)). Destes modelos, três obtiveram também resultados superiores a 75% no critério de avaliação PRED(25) (resultado a partir do qual um modelo é geralmente considerado aceitável (Conte, Dunsmore e Shen, 1986; Fedotova, Teixeira e Alvelos, 2013; Wen et al., 2012)). Em relação a outros modelos existentes na literatura, dois dos modelos propostos (*Decision Trees* e *Random Forest*) obtiveram melhores resultados e outros dois (*Multi Layer Perceptron* e *Gradient Boosting*) obtiveram piores resultados. Para além destes, os modelos baseados em *Support Vector Regression* obtiveram resultados semelhantes a outros modelos já existentes.

Alguns autores de modelos já existentes na literatura comentaram que o tamanho muito reduzido do *dataset* por eles utilizado pode influenciar os resultados dos seus respetivos modelos. Para tentar perceber melhor este potencial problema o modelo proposto foi executado novamente 10 vezes utilizando 10 subconjuntos de 30 *user stories* do *dataset* utilizado

anteriormente. Os resultados produzidos pelos modelos resultantes mostraram que apesar da quantidade de dados ser a mesma a utilização de diferentes conjuntos de dados pode causar um grande impacto no desempenho dos mesmos. Quanto menor for a dimensão do *dataset* utilizado num determinado modelo mais facilmente estes desvios podem ocorrer o que pode influenciar o poder de generalização dos modelos propostos.

7. Conclusão

A importância das estimativas de esforço de desenvolvimento de *software* é, já há alguns anos, evidente. A grande maioria dos métodos de estimativa de esforço utilizados no desenvolvimento de *software* tradicional foi desenvolvido para ser utilizado num determinado contexto. Este contexto requer geralmente que os requisitos do projeto estejam completamente definidos. Quando este não é o caso, como por exemplo em ambientes ágeis, a aplicabilidade destes métodos é limitada. Com o surgimento das metodologias ágeis surgiu também a necessidade da criação de métodos de estimativa de esforço que se adequem a este novo paradigma. Nos últimos anos, têm surgido na literatura vários métodos, porém, foram encontrados resultados inconsistentes quanto à precisão das estimativas produzidas pelos mesmos.

O modelo proposto difere dos demais existentes na literatura por utilizar dados referentes a *user stories* em vez de dados referentes a projetos completos. Num ambiente ágil esta abordagem faz perfeito sentido pois muitas vezes os requisitos do projeto não são conhecidos na totalidade e para além disto podem ser alterados em qualquer altura. A elevada volatilidade dos requisitos de um projeto ágil inviabiliza que este seja estimado como um todo. O aumento da granularidade dos dados, com a utilização de *user stories*, assemelha o modelo proposto aos métodos de estimativa já largamente utilizados nesta área. Espera-se que isto permita que o método proposto possa ser utilizado como complemento ou alternativa a métodos como o *Planning Poker*.

Uma consequência da utilização de *user stories* é o aumento do volume de dados. Este aumento é benéfico para a utilização de técnicas de *Machine Learning*. Vários autores de outros modelos existentes na literatura, que também utilizam técnicas de *Machine Learning*, alertaram para o facto da quantidade de dados utilizada por estes ser possivelmente insuficiente para obter resultados realistas. Mostrou-se que a utilização de conjuntos de dados de dimensões reduzidas

pode influenciar drasticamente os resultados obtidos. Quanto maior for a quantidade de dados disponíveis, mais realistas serão os resultados produzidos pelos modelos que utilizam técnicas de *Machine Learning*. Isto sucede-se porque os modelos treinados utilizando uma maior quantidade de dados conseguem obter uma maior capacidade de generalização.

Foi proposto a utilização e comparação de nove técnicas de *Machine Learning*. Os resultados da avaliação do desempenho dos modelos implementados mostram que os modelos baseados em técnicas *ensemble* (*Bagging*, *Random Forest*, *Extremely Randomized Trees* e *Gradient Boosting*) superam os restantes, sendo os únicos que se podem considerar desde já aceitáveis. Por outro lado, os modelos baseados em *Linear Regression* apresentam resultados muito negativos o que indica que não se adequam ao problema em estudo. Os modelos baseados em *Decision Trees* apresentam resultados ligeiramente inferiores aos dos modelos baseados em técnicas *ensemble*. Os resultados destes modelos são muito promissores e com mais alguma investigação acredita-se que possam também produzir modelos aceitáveis. Os modelos baseados nas em *K-Nearest Neighbors*, *Support Vector Regression* e *Multi Layer Perceptron* apresentam resultados inconclusivos. Isto pode dever-se a certas especificidades exististes nestas técnicas que não foram exploradas com detalhe suficiente.

7.1 Limitações

À semelhança dos modelos já existentes na literatura, o modelo proposto foi desenvolvido tendo em conta várias limitações.

O *dataset* utilizado tem por base *user stories* que foram estimadas recorrendo ao método *Planning Poker*. Apesar de este ser o método mais frequentemente utlizado em ambientes ágeis, certamente não é o único. Assim, o modelo proposto deve apenas ser considerado válido para cenários onde é utilizado o método de estimativa de esforço em ambientes ágeis *Planning Poker*.

Para a criação do *dataset* utilizado, foram recolhidos dados relativos a mais de três mil *user stories*. Isto representa um aumento substancial do volume de dados utilizado para um estudo na área da estimativa de custos de *software* em ambientes ágeis. Apesar disto, em comparação com outras áreas em que *Machine Learning* também é utilizado, este volume de dados pode ainda ser considerado como sendo insuficiente.

Os dados recolhidos são provenientes de onze projetos distintos. A natureza relativa das estimativas que utilizam *story points* faz com que estas não sejam diretamente comparáveis. Como uma técnica muito simples de normalização dos *story points*, foi incluída no *dataset* uma estimativa da velocidade esperada para cada iteração, porém não foram executadas nenhumas transformações às estimativas. Qualquer proveito destes valores será proveniente do treino realizado por cada técnica de *Machine Learning* utilizada.

As nove técnicas de *Machine Learning* utilizadas no modelo proposto foram escolhidas baseado nas técnicas disponibilizadas pelo *Scikit Learn*. Apesar da vasta utilização do *Scikit Learn*, não só em ambientes académicos, mas também em ambientes comerciais, este não disponibiliza algumas implementações mais avançadas de certos modelos. Destaca-se principalmente a ausência de implementações de algumas técnicas mais avançadas de redes neuronais artificiais, o que poderá também explicar os resultados inconclusivos obtidos relativamente aos modelos baseados em *Multi Layer Perceptron*.

7.2 Trabalhos Futuros

A área de investigação de estimativa de custos de *software* ainda está na sua infância e conta com múltiplas oportunidades de investigação. Em relação ao modelo proposto, é sugerido uma investigação mais aprofundada de alguns aspetos do mesmo.

Uma das características da utilização de *story points* é que estes não devem ser comparados entre equipas. Sugere-se que sejam investigadas formas de normalizar as estimativas de equipas distintas. Isto pode ser realizado através do pré-processamento dos dados ou idealmente pelos próprios modelos. Para além disto, sugere-se também a investigação do impacto de cada atributo presente no *dataset* utilizado. Atributos cujo impacto não é significativo podem contribuir de forma negativa para os modelos.

Os modelos implementados que foram considerados aceitáveis (*Bagging*, *Random Forest*, *Extremely Randomized Trees* e *Gradient Boosting*) devem ser avaliados no contexto de um projeto real. Sugere-se que sejam utilizados primeiramente apenas como auxílio aos métodos de estimativa de esforço normalmente utilizados e só em caso de avaliação positiva, como uma alternativa. Os modelos para os quais os resultados obtidos foram considerados inconclusivos (*K-Nearest Neighbors*, *Support Vector Regression* e *Multi Layer Perceptron*) devem ser alvo de uma investigação mais aprofundada de forma a entender as causas dos resultados obtidos e como os melhorar.

Devem também ser investigados novas técnicas de *Machine Learning* ou variações das técnicas já utilizadas. Implementações diferentes podem dar origem a resultados diferentes. É importante relembrar a importância dos *hyperparameters* nas várias técnicas de *Machine Learning*. Uma melhor otimização dos *hyperparameters* poderá também melhorar os resultados dos vários modelos.

Por fim, sugere-se que sejam recolhidos mais dados de *user stories* completas de modo a aumentar o tamanho do *dataset*. Quanto maior for a quantidade de dados disponíveis, mais precisa será a avaliação do poder de generalização dos modelos.

REFERÊNCIAS

Abdi, Hervé. 2010. «Normalizing Data». Em *Encyclopedia of research design*. Vol. 1. Sage.

Abran, Alain e Pierre N. Robillard. 1994. «Function points: a study of their measurement processes and scale transformations». *Journal of Systems and Software* 25 (2):171–184.

Abu-Mostafa, Yaser S., Malik Magdon-Ismail e Hsuan-Tien Lin. 2012. *Learning from Data: A Short Course*. S.l.: AMLbook.com.

Albrecht, A. J. 1979. «Measuring Application Development Productivity». Em *Proceedings of the Joint SHARE, GUIDE, and IBM Application Development Symposium*, 83–92. IBM Corporation.

Albrecht, Allan J. e John E. Gaffney. 1983. «Software function, source lines of code, and development effort prediction: a software science validation». *IEEE transactions on software engineering*, n. 6:639–648.

Alkoffash, Mahmud S., Mohammed J. Bawaneh e Adnan I. Al Rabea. 2008. «Which Software Cost Estimation Model to Choose in a Particular Project». *Journal of Computer Science* 4 (7):606–12. https://doi.org/10.3844/jcssp.2008.606.612.

Arifoglu, Ali. 1993. «A Methodology for Software Cost Estimation». *ACM SIGSOFT Software Engineering Notes* 18 (2):96–105. https://doi.org/10.1145/159420.155844.

Azzeh, Mohammad, Ali Bou Nassif e Leandro L. Minku. 2015. «An Empirical Evaluation of Ensemble Adjustment Methods for Analogy-Based Effort Estimation». *Journal of Systems and Software* 103 (Maio):36–52. https://doi.org/10.1016/j.jss.2015.01.028.

Bailey, John W e Victor R Basili. 1981. «A meta-model for software development resource expenditures». Em *Proceedings of the 5th international conference on Software engineering*, 107–116. IEEE Press.

Balaji, N., N. Shivakumar e V. Vignaraj Ananth. 2013. «Software cost estimation using function point with non algorithmic approach». *Global Journal of Computer Science and Technology* 13 (8).

Barcelos Tronto, Iris Fabiana de, José Demísio Simões da Silva e Nilson Sant'Anna. 2008. «An Investigation of Artificial Neural Networks Based Prediction Systems in Software Project Management». *Journal of Systems and Software* 81 (3):356–67. https://doi.org/10.1016/j.jss.2007.05.011.

Basak, Debasish, Srimanta Pal e Dipak Chandra Patranabis. 2007. «Support vector regression». *Neural Information Processing-Letters and Reviews* 11 (10):203–224.

Basha, Saleem e Dhavachelvan Ponnurangam. 2010. «Analysis of empirical software effort estimation models». *arXiv preprint arXiv:1004.1239*.

Beck, Kent, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt e Ron Jeffries. 2001. «Manifesto for agile software development».

Bergstra, James e Yoshua Bengio. 2012. «Random search for hyper-parameter optimization». *Journal of Machine Learning Research* 13 (Feb):281–305.

Bhawna e Gobind. 2015. «Research Methodology and Approaches». *IOSR Journal of Research & Method in Education (IOSR-JRME)* 5 (3). https://doi.org/10.9790/7388-05344851.

Bingamawa, Muhammad Tosan e Massila Kamalrudin. 2016. «A Review of Software Cost Estimation: Tools, Methods, and Techniques». https://doi.org/10.13140/rg.2.2.18980.48008.

Bishop, Christopher M. 2006. *Pattern recognition and machine learning*. Information science and statistics. New York: Springer.

Boehm, B. e K. Sullivan. 1999. «Software economics: status and prospects». *Information and Software Technology* 41 (14):937–946.

Boehm, Barry, Chris Abts, A Winsor Brown, Sunita Chulani, Bradford K Clark, Ellis Horowitz, Ray Madachy, Donald J Reifer e Bert Steece. 2000. «Cost estimation with COCOMO II». *ed: Upper Saddle River, NJ: Prentice-Hall*.

Boehm, Barry, Chris Abts e Sunita Chulani. 2000. «Software development cost estimation approaches—A survey». *Annals of software engineering* 10 (1–4):177–205.

Boehm, Barry, Brad Clark, Sunita D. Chulani, Ellis Horowitz, Ray Madachy, Don Reifer, Rick Selby e Bert Steece. 2000. «COCOMO II Model Definition Manual».

Boehm, Barry, Bradford Clark, Ellis Horowitz, Chris Westland, Ray Madachy e Richard Selby. 1995. «Cost models for future software life cycle processes: COCOMO 2.0». *Annals of software engineering* 1 (1):57–94.

Boehm, Barry W. 1981. *Software engineering economics*. Vol. 197. Prentice-hall Englewood Cliffs (NJ).

Borade, Jyoti G. e Vikas R. Khalkar. 2013. «Software Project Effort and Cost Estimation Techniques». *International Journal of Advanced Research in Computer Science and Software Engineering* 3 (8).

Bowling, Ann. 2014. *Research methods in health: investigating health and health services*. McGraw-Hill Education (UK).

Braga, Petronio L., Adriano L. I. Oliveira e Silvio R. L. Meira. 2007. «Software Effort Estimation using Machine Learning Techniques with Robust Confidence Intervals». Em *7th International Conference on Hybrid Intelligent Systems (HIS 2007)*, 352–57. IEEE. https://doi.org/10.1109/HIS.2007.56.

Britto, Ricardo, Emilia Mendes e Jurgen Borstler. 2015. «An Empirical Investigation on Effort Estimation in Agile Global Software Development». Em *2015 IEEE 10th International Conference on Global Software Engineering*, 38–45. IEEE. https://doi.org/10.1109/ICGSE.2015.10.

Brown, Gavin, Jeremy Wyatt, Rachel Harris e Xin Yao. 2005. «Diversity Creation Methods: A Survey and Categorisation». *Information Fusion* 6 (1):5–20. https://doi.org/10.1016/j.inffus.2004.04.004.

Calefato, Fabio e Filippo Lanubile. 2011. «A Planning Poker Tool for Supporting Collaborative Estimation in Distributed Agile Development». Em *6th International Conference on Software Engineering Advances (ICSEA 2011)*, 14–19.

Cao, Lan. 2008. «Estimating agile software project effort: an empirical study». *AMCIS 2008 Proceedings*, 401.

Chandra, Arjun e Xin Yao. 2006. «Ensemble Learning Using Multi-Objective Evolutionary Algorithms». *Journal of Mathematical Modelling and Algorithms* 5 (4):417–45. https://doi.org/10.1007/s10852-005-9020-3.

Chemuturi, Murali. 2011. «Analogy based software estimation». *Chemuturi Consultants*.

Choetkiertikul, Morakot, Hoa Khanh Dam, Truyen Tran, Trang Pham, Aditya Ghose e Tim Menzies. 2016. «A deep learning model for estimating story points». *arXiv preprint arXiv:1609.00489*.

Coelho, Evita e Anirban Basu. 2012. «Effort estimation in agile software development using story points». *International Journal of Applied Information Systems (IJAIS)* 3 (7).

Cohen, David, Mikael Lindvall e Patricia Costa. 2004. «An Introduction to Agile Methods». Em *Advances in Computers*, 62:1–66. Elsevier. https://doi.org/10.1016/S0065-2458(03)62001-2.

Cohn, Mike. 2004. *User stories applied: for agile software development*. Addison-Wesley signature series. Boston: Addison-Wesley.

———. 2005. *Agile estimating and planning*. Pearson Education.
———. 2010. *Succeeding with agile: software development using scrum*. Pearson Education India.
———. 2014a. «Don't Equate Story Points to Hours». Mountain Goat Software. 2014. https://www.mountaingoatsoftware.com/blog/dont-equate-story-points-to-hours.
———. 2014b. «Story Points Are Still About Effort». Mountain Goat Software. 2014. https://www.mountaingoatsoftware.com/blog/story-points-are-still-about-effort.
Conte, Samuel Daniel, Hubert E Dunsmore e Vincent Y Shen. 1986. *Software engineering metrics and models*. Benjamin-Cummings Publishing Co., Inc.
Cortez, Paulo. 2010. «Data mining with neural networks and support vector machines using the R/rminer tool». *Advances in data mining. Applications and theoretical aspects*, 572-583.
Creswell, John W. 2014. *Research design: qualitative, quantitative, and mixed methods approaches*. 4th ed. Thousand Oaks: SAGE Publications.
Cuadrado-Gallego, Juan J., Pablo Rodríguez-Soria e Borja Martín-Herrera. 2010. «Analogies and Differences between Machine Learning and Expert Based Software Project Effort Estimation». Em *2010 11th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, 269–76. IEEE. https://doi.org/10.1109/SNPD.2010.47.
Dave, Vachik S. e Kamlesh Dutta. 2014. «Neural Network Based Models for Software Effort Estimation: A Review». *Artificial Intelligence Review* 42 (2):295–307. https://doi.org/10.1007/s10462-012-9339-x.
Desharnais, J. 1989. «Analyse statistique de la productivitie des projets informatique a partie de la technique des point des function». *Master Thesis, University of Montreal*.
Dietterich, Thomas G. 2000. «Ensemble methods in machine learning». *Multiple classifier systems* 1857:1–15.
Elish, Mahmoud O. 2009. «Improved Estimation of Software Project Effort Using Multiple Additive Regression Trees». *Expert Systems with Applications* 36 (7):10774–78. https://doi.org/10.1016/j.eswa.2009.02.013.
Fedotova, Olga, Leonor Teixeira e Helena Alvelos. 2013. «Software Effort Estimation with Multiple Linear Regression: Review and Practical Application.» *J. Inf. Sci. Eng.* 29 (5):925–945.
Foss, Tron, Erik Stensrud, Barbara Kitchenham e Ingunn Myrtveit. 2003. «A simulation study of the model evaluation criterion MMRE». *IEEE Transactions on Software Engineering* 29 (11):985–995.
Friedman, Jerome H. 1999. «Stochastic gradient boosting». *Computational Statistics & Data Analysis*, 10.
Gandomani, Taghi Javdani, Tieng Wei Koh e Abdulelah Khaled Binhamid. 2014. «A case study research on software cost estimation using experts' estimates, Wideband Delphi, and Planning Poker technique». *International Journal of Software Engineering and its applications* 8 (11):173–182.
Geurts, Pierre, Damien Ernst e Louis Wehenkel. 2006. «Extremely Randomized Trees». *Machine Learning* 63 (1):3–42. https://doi.org/10.1007/s10994-006-6226-1.
Grenning, James. 2002. «Planning poker or how to avoid analysis paralysis while release planning». *Hawthorn Woods: Renaissance Software Consulting* 3.
Hackeling, Gavin. 2014. *Mastering Machine Learning with Scikit-Learn: Apply Effective Learning Algorithms to Real-World Problems Using Scikit-Learn*. Packt Open Source. Birmingham: Packt Publ.

Hastie, Trevor, Robert Tibshirani e Jerome Friedman. 2009. *The Elements of Statistical Learning*. Springer Series in Statistics. New York, NY: Springer New York. https://doi.org/10.1007/b94608.

Heemstra, Fred J. 1992. «Software cost estimation». *Information and software technology* 34 (10):627–639.

Humayun, Mamoona e Cui Gang. 2012. «Estimating effort in global software development projects using machine learning techniques». *International Journal of Information and Education Technology* 2 (3):208.

Idri, Ali, Alain Abran e Laila Kjiri. 2000. «COCOMO cost model using fuzzy logic». Em *7th International Conference on Fuzzy Theory & Techniques*. Vol. 27.

ISBSG. 1997. «International Software Benchmarking Standards Group (ISBSG)». ISBSG. 2017 de 1997. http://isbsg.org/.

ISPA. 2008. *Parametric estimating handbook*. 4.ª ed. International Society of Parametric Analysts.

James, Gareth, Daniela Witten, Trevor Hastie e Robert Tibshirani. 2013. *An introduction to statistical learning*. Vol. 112. Springer.

Jørgensen, Magne. 2004. «A Review of Studies on Expert Estimation of Software Development Effort». *Journal of Systems and Software* 70 (1–2):37–60. https://doi.org/10.1016/S0164-1212(02)00156-5.

Jørgensen, Magne e Martin Shepperd. 2007. «A systematic review of software development cost estimation studies». *IEEE Transactions on software engineering* 33 (1).

Kang, Sungjoo, Okjoo Choi e Jongmoon Baik. 2010. «Model-Based Dynamic Cost Estimation and Tracking Method for Agile Software Development». Em *2010 IEEE/ACIS 9th International Conference on Computer and Information Science*, 743–48. IEEE. https://doi.org/10.1109/ICIS.2010.126.

Kemerer, Chris F. 1987. «An empirical validation of software cost estimation models». *Communications of the ACM* 30 (5):416–29. https://doi.org/10.1145/22899.22906.

Khatibi, Vahid e Dayang Jawawi. 2011. «Software Cost Estimation Methods: A Review». *Journal of Emerging Trends in Computing and Information Sciences* 2 (1):21–29.

Khuttan, Anuj, Ashwini Kumar e Archana Singh. 2014. «A Survey Of Effort Estimation Techniques For The Software Development». *International Journal of Scientific & Technology Research* 3 (7):234–36.

Kumari, Sweta e Shashank Pushkar. 2013a. «Comparison and analysis of different software cost estimation methods». *International Journal of Advanced Computer Science and application (IJACSA)* 4 (1).

———. 2013b. «Performance analysis of the software cost estimation methods: a review». *International Journal of Advanced Research in Computer Science and Software Engineering* 3 (7).

Kuncheva, Ludmila e Chris Whitaker. 2003. *Measures of Diversity in Classifier Ensembles and Their Relationship with the Ensemble Accuracy*. Vol. 51. https://doi.org/10.1023/A:1022859003006.

Leung, Hareton e Zhang Fan. 2002. «Software cost estimation». *Handbook of Software Engineering, Hong Kong Polytechnic University*, 1–14.

Litoriya, Ratnesh e Abhay Kothari. 2013. «An Efficient Approach for Agile Web Based Project Estimation: AgileMOW». *Journal of Software Engineering and Applications* 06 (06):297–303. https://doi.org/10.4236/jsea.2013.66037.

Mair, Carolyn, Gada Kadoda, Martin Lefley, Keith Phalp, Chris Schofield, Martin Shepperd e Steve Webster. 2000. «An investigation of machine learning based prediction systems». *Journal of Systems and Software* 53 (1):23–29.

Malhotra, Ruchika e Ankita Jain. 2011. «Software effort prediction using statistical and machine learning methods». *International Journal of Advanced Computer Science and Applications* 2 (1):145–152.

Manifesto, CHAOS. 2013. «Think Big, Act Small». *The Standish Group International Inc* 176.

Matos, Manuel António. 1995. «Manual operacional para a regressão linear». Faculdade de Engenharia da Universidade do Porto.

Mendes, Emilia, Nile Mosley e Steve Counsell. 2005. «Investigating Web size metrics for early Web cost estimation». *Journal of Systems and Software* 77 (2):157–172.

Mendes, Emilia, Ian Watson, Chris Triggs, Nile Mosley e Steve Counsell. 2003. «A comparative study of cost estimation models for web hypermedia applications». *Empirical Software Engineering* 8 (2):163–196.

Menzies, Tim, Zhihao Chen, Jairus Hihn e Karen Lum. 2006. «Selecting best practices for effort estimation». *IEEE Transactions on Software Engineering* 32 (11):883–895.

Merlo–Schett, Nancy, Martin Glinz e Arun Mukhija. 2002. «Seminar on Software Cost Estimation WS 2002/2003». *Department of Computer Science*, 3–19.

Mitchell, Tom M. 1997. *Machine Learning*. McGraw-Hill series in computer science. New York: McGraw-Hill.

Moharreri, Kayhan, Alhad Vinayak Sapre, Jayashree Ramanathan e Rajiv Ramnath. 2016. «Cost-Effective Supervised Learning Models for Software Effort Estimation in Agile Environments». Em *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, 2:135–140. IEEE.

Molokken, Kjetil e Magne Jorgensen. 2003. «A review of software surveys on software effort estimation». Em *Empirical Software Engineering, 2003. ISESE 2003. Proceedings. 2003 International Symposium on*, 223–230. IEEE.

Munialo, Samson Wanjala e Geoffrey Muchiri Muketha. 2016. «A Review of Agile Software Effort Estimation Methods». *International Journal of Computer Applications Technology and Research* 5 (9):612–918. https://doi.org/10.7753/IJCATR0509.1009.

Myrtveit, Ingunn, Erik Stensrud e Martin Shepperd. 2005. «Reliability and validity in comparative studies of software prediction models». *IEEE Transactions on Software Engineering* 31 (5):380–391.

Nassif, Ali Bou, Mohammad Azzeh, Luiz Fernando Capretz e Danny Ho. 2016. «Neural network models for software development effort estimation: a comparative study». *Neural Computing and Applications* 27 (8):2369–2381.

Nassif, Ali Bou, Danny Ho e Luiz Fernando Capretz. 2013. «Towards an early software estimation using log-linear regression and a multilayer perceptron model». *Journal of Systems and Software* 86 (1):144–160.

Nerkar, L. R. e P. M. Yawalkar. 2014. «Software Cost Estimation using Algorithmic Model and Non-Algorithmic Model a Review». *IJCA Proceedings on Innovations and Trends in Computer and Communication Engineering*, n. 2:4–7.

Nguyen, Vu, Sophia Deeds-Rubin, Thomas Tan e Barry Boehm. 2007. «A SLOC counting standard». Em *COCOMO II Forum*.

Norden, Peter V. 1970. «Useful tools for project management». *Management of Production*, 71–101.

Oliveira, Adriano L.I. 2006. «Estimation of Software Project Effort with Support Vector Regression». *Neurocomputing* 69 (13–15):1749–53. https://doi.org/10.1016/j.neucom.2005.12.119.

Osman, Hala Hamad e Mohamed Elhafiz Musa. 2016. «A Survey of Agile Software Estimation Methods». *International Journal of Computer Science and Telecommunications* 7 (3).

Panda, Aditi. 2015. «Effort Estimation of Agile and Web-Based Software Using Artificial Neural Networks».

Panda, Aditi, Shashank Mouli Satapathy e Santanu Kumar Rath. 2015. «Empirical Validation of Neural Network Models for Agile Software Effort Estimation Based on Story Points». *Procedia Computer Science* 57:772–81. https://doi.org/10.1016/j.procs.2015.07.474.

pandas. 2008. «pandas - Python Data Analysis Library». 2017 de 2008. http://pandas.pydata.org/.

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, et al. 2011. «Scikit-learn: Machine Learning in Python». *Journal of Machine Learning Research* 12:2825–2830.

Port, Dan, Vu Nguyen e Tim Menzies. 2009. «Studies of confidence in software cost estimation research based on the criterions mmre and pred».

Prabhakar, Maitreyee Dutta. 2013. «Prediction of software effort using artificial neural network and support vector machine». *International Journal of Advanced Research in Computer Science and Software Engineering* 3 (3).

Pressman, Roger S. 1997. *Software engineering: a practitioner's approach*. 4th ed. McGraw-Hill series in computer science. McGraw Hill.

Prettenhofer, Peter e Gilles Louppe. 2014. «Gradient boosted regression trees in scikit-learn».

Putnam, Lawrence H. 1978. «A general empirical solution to the macro software sizing and estimating problem». *IEEE transactions on Software Engineering*, n. 4:345–361.

Raju, H. K. e Y. T. Krishnegowda. 2013. «Software Sizing and Productivity with Function Points». *Lecture Notes on Software Engineering* 1 (2):204–8. https://doi.org/10.7763/LNSE.2013.V1.46.

Raschka, Sebastian. 2016. *Python Machine Learning*. Community Experience Distilled. Birmingham Mumbai: Packt Publishing.

Rockefeller, John D. e Alexander Hamilton. 2017. «The Founding Fathers of Agile». *CrossTalk*, 15.

Santana, Célio, Fabiana Leoneo, Alexandre Vasconcelos e Cristine Gusmão. 2011. «Using function points in agile projects». *Agile Processes in Software Engineering and Extreme Programming*, 176–191.

Saroha, Meenakshi e Shashank Sahu. 2015. «Tools & methods for software effort estimation using use case points model—A review». Em *Computing, Communication & Automation (ICCCA), 2015 International Conference on*, 874–879. IEEE.

Satapathy, Shashank Mouli. 2016. «Effort Estimation Methods in Software Development Using Machine Learning Algorithms».

Satapathy, Shashank Mouli, Aditi Panda e Santanu Kumar Rath. 2014. «Story Point Approach based Agile Software Effort Estimation using Various SVR Kernel Methods». Em *The 26th International Conference on Software Engineering and Knowledge Engineering*, 304–7.

Satapathy, Shashank Mouli e Santanu Kumar Rath. 2017. «Empirical Assessment of Machine Learning Models for Agile Software Development Effort Estimation Using Story Points». *Innovations in Systems and Software Engineering*, Junho. https://doi.org/10.1007/s11334-017-0288-z.

Schmietendorf, Andreas, Martin Kunz e Reiner Dumke. 2008. «Effort estimation for agile software development projects». Em *5th Software Measurement European Forum*, 113-123.

scikit-learn. 2017. «scikit-learn user guide». Release 0.19.0. scikit-learn.

Sehra, Sumeet Kaur, Yadwinder Singh Brar e Navdeep Kaur. 2013. «Soft computing techniques for software effort estimation». *International Journal of Advanced Computer and Mathematical Sciences ISSN* 2 (3):160–67.

Seni, Giovanni e John F. Elder. 2010. «Ensemble Methods in Data Mining: Improving Accuracy Through Combining Predictions». *Synthesis Lectures on Data Mining and Knowledge Discovery* 2 (1):1–126. https://doi.org/10.2200/S00240ED1V01Y200912DMK002.

Sewell, Martin. 2008. «Ensemble learning». *RN* 11 (02).

Shalev-Shwartz, Shai e Shai Ben-David. 2014. *Understanding machine learning: from theory to algorithms*. New York, NY, USA: Cambridge University Press.

Sharma, Jyoti e Kulvinder Singh. 2017. «An Analysis on Software Cost Estimation Techniques». *International Journal of Engineering & Technology Innovations* 4 (1):27-30.

Sharma, Narendra, Aman Bajpai e Mr Ratnesh Litoriya. 2012. «A comparison of software cost estimation methods: A Survey». *The International Journal of Computer Science and Applications (TIJCSA)* 1 (3).

Sharma, Narendra e Ratnesh Litoriya. 2012. «Incorporating Data Mining Techniques on Software Cost Estimation: Validation and». *International Journal of Emerging Technology and Advanced Engineering*, Março de 2012.

Sharma, P. 2004. *Software Engineering*. APH Pub.

Sharma, T. N., Anil Bhardwaj e Anita Sharma. 2011. «A Comparative study of COCOMO II and Putnam models of Software Cost Estimation». *vol* 2:1–3.

Shekhar, Shivangi e Umesh Kumar. 2016. «Review of Various Software Cost Estimation Techniques». *International Journal of Computer Applications* 141 (11).

Shepperd, Martin e Chris Schofield. 1997. «Estimating software project effort using analogies». *IEEE Transactions on software engineering* 23 (11):736–743.

Simon, Noah e Robert Tibshirani. 2014. «Comment on "Detecting Novel Associations In Large Data Sets" by Reshef Et Al, Science Dec 16, 2011». *arXiv:1401.7645 [stat]*, Janeiro.

Smola, Alex J e Bernhard Schölkopf. 2004. «A tutorial on support vector regression». *Statistics and computing* 14 (3):199–222.

Srinivasan, Krishnamoorthy e Douglas Fisher. 1995. «Machine learning approaches to estimating software development effort». *IEEE Transactions on Software Engineering* 21 (2):126–137.

Stellman, Andrew e Jennifer Greene. 2005. *Applied software project management*. O'Reilly Media, Inc.

Tinnirello, Paul C. 1999. *Systems development handbook*. CRC Press.

Touesnard, Brad. 2004. «Software Cost Estimation: SLOC-based Models and the Function Points Model Version 1.1».

Ungan, Erdir, Numan Cizmeli e Onur Demirors. 2014. «Comparison of Functional Size Based Estimation and Story Points, Based on Effort Estimation Effectiveness in SCRUM Projects». Em *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, 77–80. IEEE. https://doi.org/10.1109/SEAA.2014.83.

Usman, Muhammad. 2015. «Supporting Effort Estimation in Agile Software Development». Karlskrona: Blekinge Tekniska Högskola.

Usman, Muhammad, Emilia Mendes e Jürgen Börstler. 2015. «Effort Estimation in Agile Software Development: A Survey on the State of the Practice». Em , 1–10. ACM Press. https://doi.org/10.1145/2745802.2745813.

Waghmode, Swati e K Kolhe. 2014. «A Novel Way of Cost Estimation in Software Project Development Based on Clustering Techniques». *Int. J. Innov. Res. Comput. Commun. Eng* 2 (4):3892–3899.

Wen, Jianfeng, Shixian Li, Zhiyong Lin, Yong Hu e Changqin Huang. 2012. «Systematic Literature Review of Machine Learning Based Software Development Effort Estimation Models». *Information and Software Technology* 54 (1):41–59. https://doi.org/10.1016/j.infsof.2011.09.002.

Witten, Ian H, Eibe Frank, Mark A Hall e Christopher J Pal. 2016. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.

Wolpert, David H. 1996. «The lack of a priori distinctions between learning algorithms». *Neural computation* 8 (7):1341–1390.

Zhang, Du e Jeffrey JP Tsai. 2002. «Machine learning and software engineering». Em *Proceedings-International Conference on Tools with Artificial Intelligence, TAI*, 22-29.

———. 2003. «Machine learning and software engineering». *Software Quality Journal* 11 (2):87–119.

Zheng, Alice. 2015. *Evaluating Machine Learning Models A beginner's guide to key concepts and pitfalls.* O'Reilly.

Zia, Z., Abdur Rashid e Khair uz Zaman. 2011. «Software Cost Estimation for Component-Based Fourth-Generation-Language Software Applications». *IET Software* 5 (1):103. https://doi.org/10.1049/iet-sen.2010.0027.

Zia, Ziauddin Khan, Shahid Kamal Tipu e Shahrukh Khan Zia. 2012. «An effort estimation model for agile software development». *Advances in Computer Science and its Applications (ACSA)* 2 (1):314–324.

Anexos

ANEXO A – CÓDIGO FONTE DO MODELO PROPOSTO

main.py

```
import sys
import time
import pandas   as pd
import metrics  as m

from models                      import getModels
from preprocessing               import preprocessData
from collections                 import OrderedDict
from sklearn.model_selection     import KFold
from sklearn.model_selection     import GridSearchCV
from sklearn.model_selection     import RandomizedSearchCV

# pandas settings
pd.set_option('display.max_rows', 1000)
pd.set_option('display.max_columns', 500)
pd.set_option('display.width', 1000)

# Set Start Time
startTime = time.gmtime()

# Set dataset path
datasetDir = 'Dataset/'
datasetFilePath = 'dataset.csv'

# Import data
df = pd.read_csv(datasetDir + datasetFilePath, header=0)
df_h = list(df.columns.values)

# Split data into features and target
targetColumnName = 'UserStoryActualHours'
x = df.drop(targetColumnName, axis=1)
y = df[targetColumnName]

# Preprocess data
categoricalColumns = [
    'ProjectName',
    'ProjectLanguage',
    'UserStoryMainLanguage'
]
x = preprocessData(x, df_h, categoricalColumns)

# Write output to file
sys.stdout = open('Logs/' + time.strftime('%Y-%m-%d %H%M%S', startTime) + '
' + datasetFilePath + '.txt', 'w')

# Print dataset path
print(datasetFilePath + '\n')
# Print number of samples
print('Samples: ' + str(x.shape[0]))
# Print number of features and names (after preprocessing)
print('Features: ' + str(x.shape[1]) + '\n')
print(list(x.columns.values))
```

```
# Initialize seed to be used by the random number generator. This ensures
repeatable results
rState = 1
print('\nRandom State: ' + str(rState))

# Define models
models = getModels(rState)

# Define results variables
gsResults = OrderedDict()
for model in models:
    gsResults[model[0]] = []
resultsSummary = []

# Nested Cross-Validation
# Split data into folds
nFolds = 10
print('Folds: ' + str(nFolds))

kf = KFold(n_splits=nFolds, shuffle=True, random_state=rState)

# Outer loop - For each fold train and evaluate the model using the optimal
hyperparameters
for i, [train_index, test_index] in enumerate(kf.split(x, y)):

    # Split data into training and test subsets
    x_train, x_test = x.iloc[train_index], x.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Inner loop - Find optimal Hyperparameters for each model
    for name, model, parameters, randomSearch in models:

        print('\nFold ' + str(i + 1) + ' - ' + name)

        # Grid Search or Random Search depending on the number of
        parameters to search and the time it would take
        if randomSearch:
            gs = RandomizedSearchCV(model, parameters, cv=nFolds,
                                    scoring=m.scorer_r2(), n_iter=20,
                                    random_state=rState,
                                    n_jobs=-1, verbose=True)
        else:
            gs = GridSearchCV(model, parameters, cv=nFolds,
                              scoring=m.scorer_r2(),
                              n_jobs=-1, verbose=True)

        # Train the model to find the optimal hyperparameters
        gs.fit(x_train, y_train)

        # Save Grid/Random Search results
        gsResults[name].append([gs.best_estimator_, i + 1, gs.best_params_,
        gs.best_score_])

        # Evaluate model using the test data
        y_pred = gs.best_estimator_.predict(x_test)
        mae, mmre, pred25 = m.getMetrics(y_test, y_pred, name)
```

```
        # Save model metrics
        resultsRow = [i + 1, name, mae, mmre, pred25]
        resultsSummary.append(resultsRow)

# Print grid search results
m.printGSResults(gsResults)

# Print results summary
m.printResultsSummary(resultsSummary)

# Set End Time
endTime = time.gmtime()

# Print execution time
m.printExecutionTime(startTime, endTime)

# Close output file
sys.stdout.close()
```

metrics.py

```
import time
import pandas   as pd

from sklearn.metrics    import make_scorer
from sklearn.metrics    import r2_score


def printExecutionTime(start, end):
    diff = (time.mktime(end) - time.mktime(start)) / 60
    hours, rem = divmod(diff, 3600)
    minutes, seconds = divmod(rem, 60)
    print('Duration: ')
    print("{:0>2}:{:0>2}:{:05.2f}".format(int(hours), int(minutes),
    seconds))


def pred25(test, pred):
    pred25 = 0

    for t, p in zip(test, pred):
        a = abs(t - p) / t
        if a < 0.25:
            pred25 += 1

    pred25 = pred25 / len(test) * 100

    return pred25


def mmre(test, pred):
    mmre = 0

    for t, p in zip(test, pred):
        mmre += abs(t - p) / t
```

```
    mmre = mmre / len(test)

    return mmre


def mae(test, pred):
    mae = 0

    for t, p in zip(test, pred):
        mae += abs(t - p)

    mae = mae / len(test)

    return mae


def r2(test, pred):
    return r2_score(test, pred)


def scorer_r2():
    return make_scorer(r2)


def getMetrics(test, pred, estimatorName='', printMetrics=False):
    _pred25 = pred25(test, pred)
    _mmre = mmre(test, pred)
    _mae = mae(test, pred)

    if printMetrics:
        if estimatorName != '':
            print(estimatorName)
        print("MAE: %.4fh" % _mae)
        print("MMRE: %.4f" % _mmre)
        print("PRED(25): %.4f%%" % _pred25)
        print()

    return _mae, _mmre, _pred25


def printResultsSummary(resultsSummary):
    print('############################################################\n')

    resultsColumns = [
        'Fold',
        'Estimator',
        'MAE',
        'MMRE',
        'PRED(25)'
    ]
    resultsDataFrame = pd.DataFrame(data=resultsSummary,
    columns=resultsColumns)

    print(resultsDataFrame)
    print()

    print('############################################################\n')

    print('Average Scores\n')
```

```
    print(resultsDataFrame.drop('Fold', axis=1).groupby('Estimator',
    sort=False).mean())
    print()


def printGSResults(gsResults):
    print('\n##########################################################\n')

    for key, values in gsResults.items():
        print(key)
        for v in values:
            print(v[1:])
        print()
```

models.py

```
from sklearn.pipeline        import make_pipeline
from sklearn                 import preprocessing

from sklearn.linear_model    import LinearRegression
from sklearn.tree            import DecisionTreeRegressor
from sklearn.neighbors       import KNeighborsRegressor
from sklearn.svm             import SVR
from sklearn.neural_network  import MLPRegressor

from sklearn.ensemble        import BaggingRegressor
from sklearn.ensemble        import RandomForestRegressor
from sklearn.ensemble        import ExtraTreesRegressor
from sklearn.ensemble        import GradientBoostingRegressor


# [
#     'Name',
#     make_pipeline(
#         Scaler(),
#         Model()
#     ),
#     {
#         'model__parameter': [1, 2, 3]
#     },
#     True/False (randomSearch)
# ]


def getModels(rState):

    return [
        [
            'Linear Regression',
            make_pipeline(
                preprocessing.StandardScaler(),
                LinearRegression()
            ),
            {},
            False
        ],
```

```
[
    'Decision Tree',
    make_pipeline(
        preprocessing.StandardScaler(),
        DecisionTreeRegressor(random_state=rState)
    ),
    {
        'decisiontreeregressor__criterion': ['mae', 'mse'],
        'decisiontreeregressor__max_depth': [None, 5, 10, 20, 50],
        'decisiontreeregressor__max_features':
                [None, 5, 10, 15, 20],
        'decisiontreeregressor__min_samples_split':
                [2, 10, 20, 50],
        'decisiontreeregressor__min_samples_leaf':
                [1, 5, 10, 20, 50]
    },
    True
],
[
    'K-Nearest Neighbors',
    make_pipeline(
        preprocessing.Normalizer(),
        KNeighborsRegressor()
    ),
    {
        'kneighborsregressor__n_neighbors':
                [1, 2, 3, 4, 5, 8, 10, 15, 20],
        'kneighborsregressor__weights': ['uniform', 'distance']
    },
    False
],
[
    'SVR',
    make_pipeline(
        preprocessing.StandardScaler(),
        SVR()
    ),
    {
        'svr__C': [10, 1, 0.1, 0.01, 0.001],
        'svr__epsilon': [10, 1, 0.1, 0.01, 0.001],
        'svr__kernel': ['linear', 'rbf', 'sigmoid'],
        'svr__degree': [2, 3, 5, 8, 10],
        'svr__gamma': [0.1, 0.01, 0.001, 0.0001]
    },
    True
],
[
    'Multi Layer Perceptron',
    make_pipeline(
        preprocessing.StandardScaler(),
        MLPRegressor(random_state=rState, max_iter=100000)
    ),
    {
        'mlpregressor__hidden_layer_sizes': [
            (50),
            (25),
            (50, 50),
            (25, 25),
            (50, 50, 50),
            (25, 25, 25),
```

```
                (50, 50, 50, 50),
                (25, 25, 25, 25),
                (50, 50, 50, 50, 50),
                (25, 25, 25, 25, 25)
            ],
            'mlpregressor__batch_size': ['auto', 10, 20, 50, 100],
            'mlpregressor__learning_rate':
                    ['constant', 'invscaling', 'adaptive'],
            'mlpregressor__learning_rate_init': [0.1, 0.01, 0.001]
        },
        True
    ],
    [
        'Bagging Regressor',
        make_pipeline(
            preprocessing.StandardScaler(),
            BaggingRegressor(random_state=rState)
        ),
        {
            'baggingregressor__base_estimator': [
                DecisionTreeRegressor(random_state=rState),
                RandomForestRegressor(random_state=rState),
                ExtraTreesRegressor(random_state=rState)
            ],
            'baggingregressor__n_estimators': [10, 50, 100, 200]
        },
        False
    ],
    [
        'Random Forest',
        make_pipeline(
            preprocessing.StandardScaler(),
            RandomForestRegressor(random_state=rState)
        ),
        {
            'randomforestregressor__n_estimators':
                    [70, 80, 100, 200, 500, 1000],
            'randomforestregressor__criterion': ['mae', 'mse'],
            'randomforestregressor__max_depth':
                    [None, 5, 10, 15, 20, 30, 50, 100],
            'randomforestregressor__max_features':
                    [None, 5, 10, 15, 20],
            'randomforestregressor__min_samples_split':
                    [5, 10, 20, 50],
            'randomforestregressor__min_samples_leaf':
                    [1, 2, 5, 10, 20]
        },
        True
    ],
    [
        'Extra Trees Regressor',
        make_pipeline(
            preprocessing.StandardScaler(),
            ExtraTreesRegressor(random_state=rState, n_jobs=-1)
        ),
        {
            'extratreesregressor__n_estimators': [100, 200, 500],
            'extratreesregressor__criterion': ['mae', 'mse'],
            'extratreesregressor__max_depth': [None, 10, 20, 50, 100],
            'extratreesregressor__max_features': [None, 10, 20],
```

```
                'extratreesregressor__min_samples_split': [2, 10, 20, 50],
                'extratreesregressor__min_samples_leaf': [1, 10, 20]
            },
            True
        ],
        [
            'Gradient Boosting',
            make_pipeline(
                preprocessing.StandardScaler(),
                GradientBoostingRegressor(random_state=rState)
            ),
            {
                'gradientboostingregressor__loss':
                        ['ls', 'lad', 'huber', 'quantile'],
                'gradientboostingregressor__learning_rate':
                        [0.1, 0.01, 0.001, 0.0001],
                'gradientboostingregressor__n_estimators':
                        [10, 20, 40, 60, 80, 90, 100, 500, 1000],
                'gradientboostingregressor__max_depth':
                        [3, 5, 8, 10, 20, 30, 50, 100],
                'gradientboostingregressor__max_features':
                        [None, 5, 10, 15, 20],
                'gradientboostingregressor__min_samples_split':
                        [5, 10, 20, 50],
                'gradientboostingregressor__min_samples_leaf':
                        [1, 2, 5, 10, 20]
            },
            True
        ]
    ]
```

preprocessing.py

```
import pandas   as pd

from sklearn    import preprocessing


def preprocessData(data, headers, categoricalColumns):
    for category in categoricalColumns:
        if category in headers:
            # Get category to be processed
            data_cat = data[category]

            # Encode categorical column labels with value between 0 and
            n_classes-1.
            enc_label = preprocessing.LabelEncoder()
            data_cat = enc_label.fit_transform(data_cat)

            # Encode categorical integer features using a one-hot aka
            one-of-K scheme
            enc = preprocessing.OneHotEncoder()
            enc.fit(data_cat.reshape(-1, 1))
            enc_data = enc.transform(data_cat.reshape(-1, 1)).toarray()

            # Create a dataframe with the results of the encoding
```

```
        enc_columns = [category + '_' + j.replace(' ', '_')
                       for j in enc_label.classes_]
        enc_df = pd.DataFrame(enc_data, columns=enc_columns)

        # Drop original column and join encoded equivalent
        data = data.drop(category, axis=1)
        data = data.join(enc_df)

return data
```

ANEXO B – RESULTADOS COMPLETOS DO MODELO PROPOSTO (DATASETS DE DIMENSÃO REDUZIDA)

Técnica	Dataset #	MAE	MMRE	PRED(25)
Linear Regression	1	2,580	53,818	40,000
	2	2,618	66,719	43,333
	3	2,146	26,886	60,000
	4	8,614	211,144	26,667
	5	1,693	23,603	60,000
	6	19,822	222,551	33,333
	7	0,879	6,590	86,667
	8	5,558	44,073	40,000
	9	1,657	21,534	70,000
	10	7,741	237,203	20,000
Decision Tree	1	2,433	45,645	48,333
	2	2,486	46,788	50,000
	3	3,838	52,774	26,667
	4	0,850	16,944	73,333
	5	1,333	21,667	80,000
	6	1,333	11,667	70,000
	7	0,933	6,111	86,667
	8	1,800	12,056	66,667
	9	1,139	13,061	73,333
	10	1,171	38,576	50,000
K-Nearest Neighbors	1	2,273	34,791	38,333
	2	2,639	43,795	36,667
	3	3,372	41,352	26,667
	4	1,956	44,434	30,000
	5	1,658	28,500	70,000
	6	2,341	22,360	53,333
	7	1,267	8,611	80,000
	8	1,928	13,688	66,667
	9	1,548	20,274	66,667
	10	1,084	40,674	53,333

Técnica	Dataset #	MAE	MMRE	PRED(25)
SVR	1	1,944	42,429	53,333
	2	2,070	41,627	50,000
	3	2,721	35,371	50,000
	4	1,320	27,389	60,000
	5	1,572	22,838	73,333
	6	1,834	19,530	66,667
	7	0,926	9,805	90,000
	8	2,064	16,617	83,333
	9	1,974	25,703	43,333
	10	1,286	39,216	53,333
Multi Layer Perceptron	1	2,191	36,137	53,333
	2	2,736	43,602	46,667
	3	3,391	43,238	36,667
	4	1,417	29,455	56,667
	5	2,036	30,378	73,333
	6	2,872	30,430	50,000
	7	1,908	19,199	66,667
	8	2,743	23,790	60,000
	9	1,828	25,621	60,000
	10	1,428	47,614	50,000
Bagging	1	2,081	39,893	61,667
	2	1,916	28,861	60,000
	3	2,807	36,775	43,333
	4	1,230	25,242	66,667
	5	1,090	17,717	76,667
	6	1,528	14,561	70,000
	7	0,840	6,778	86,667
	8	1,887	14,600	86,667
	9	1,146	14,245	83,333
	10	0,960	32,657	60,000
Random Forest	1	1,769	32,503	65,000
	2	1,883	33,964	56,667
	3	3,144	41,864	43,333
	4	1,401	32,001	53,333
	5	1,200	18,912	70,000
	6	1,509	14,439	80,000
	7	1,200	7,778	83,333
	8	1,827	14,661	80,000
	9	1,118	13,758	90,000
	10	0,997	34,461	63,333

Técnica	Dataset #	MAE	MMRE	PRED(25)
Extremely Randomized Trees	1	1,746	36,634	58,333
	2	2,217	39,312	50,000
	3	2,943	38,722	40,000
	4	1,377	31,431	60,000
	5	0,914	14,842	80,000
	6	1,631	14,505	73,333
	7	0,785	6,019	86,667
	8	1,713	13,876	76,667
	9	1,251	15,347	83,333
	10	0,937	31,232	63,333
Gradient Boosting	1	1,760	30,219	65,000
	2	2,010	32,568	56,667
	3	2,917	37,298	43,333
	4	1,312	24,933	63,333
	5	0,774	12,707	86,667
	6	1,594	14,410	73,333
	7	0,736	4,875	90,000
	8	1,429	11,771	80,000
	9	1,341	17,107	83,333
	10	0,822	26,391	60,000