PONTIFÍCIA UNIVERSIDADE CATÓLICA
DE SÃO PAULO

Flávio Kenji Yanai

Detecção de anomalias no funcionamento de software com Machine Learning

São Paulo - SP - Brasil

2020, v-1.0

PONTIFÍCIA UNIVERSIDADE CATÓLICA
DE SÃO PAULO
TECNOLOGIAS DA INTELIGÊNCIA E DESIGN DIGITAL
Programa de Pós-Graduação

Flávio Kenji Yanai

Detecção de anomalias no funcionamento de software com Machine Learning

Dissertação apresentada à Banca Examinadora da Pontifícia Universidade Católica de São Paulo, como exigência parcial para obtenção do título de MESTRE em Tecnologias da Inteligência e Design Digital, sob orientação do professor Dr. Daniel Couto Gatti

Orientador: Daniel Couto Gatti

São Paulo - SP - Brasil

2020, v-1.0

Flávio Kenji Yanai
Detecção de anomalias no funcionamento de software com Machine Learning - São Paulo - SP - Brasil, 2020, v-1.0-
61 páginas
Orientador: Daniel Couto Gatti

Dissertação de Mestrado – PONTIFÍCIA UNIVERSIDADE CATÓLICA
DE SÃO PAULO
TECNOLOGIAS DA INTELIGÊNCIA E DESIGN DIGITAL
Programa de Pós-Graduação, 2020, v-1.0.

1. Testes de Software. 2. Machine Learning. 3. Deep Learning 4. Detecção de Anomalias em Software I. Daniel Couto Gatti. II. PUCSP.

Errata

Flávio Kenji Yanai

Detecção de anomalias no funcionamento de software com Machine Learning

Dissertação apresentada à Banca Examinadora da Pontifícia Universidade Católica de São Paulo, como exigência parcial para obtenção do título de MESTRE em Tecnologias da Inteligência e Design Digital, sob orientação do professor Dr. Daniel Couto Gatti

---

Daniel Couto Gatti
Orientador

Professor
Demi Getschko

Professor
Renato Manzan

São Paulo - SP - Brasil
2020, v-1.0

Este trabalho é dedicado às todos os pesquisadores que mesmo frente às dificuldades de pesquisa em nosso país,
continuam seguindo em frente e tornando este país melhor.. .

Agradecimentos

Agradeço ao meu Prof. orientador Daniel Couto Gatti pela paciência e pelo ensinamentos que ministrou.

Ao Prof. Ítalo Veiga que expandiu a minha percepção sobre a computação.

Ao Prof. Diogo Cortiz que criou grandes bases para que este trabalho pudesse ser concluído.

Ao Prof. Demi Getschko que me apoiou imensamente durante o mestrado.

À Edna Conti que sempre me motivou e me auxiliou durante estes ano.

À Vera Braz que sempre esteve disposta a me auxiliar.

À minha familia que sempre me forneceu estrutura para meus estudos.

Agradecimentos

Esta pesquisa teve o suporte da CAPES / PROSUC (Coodernação de Aperfeiçoamento de Pessoal de Nível Superior / Programa de Suporte à Pós-Graduação de Instituições Comunitárias de Ensino Superior), mediante concessão de bolsa de Mestrado, modalidade II, objeto do processo no. 88887.199134/2118-00, o que permitiu a realização do curso de Mestrado e a conclusão da Dissertação que consolida a pesquisa realizada durante o curso.

"Não vos amoldeis às estruturas deste mundo,
mas transformai-vos pela renovação da mente,
a fim de distinguir qual é a vontade de Deus:
o que é bom, o que Lhe é agradável, o que é perfeito.
(Bíblia Sagrada, Romanos 12, 2)

Resumo

O teste de software é muitas vezes é relegado a um segundo plano nas empresas de desenvolvimento de software, os motivos pelos quais isso acontece são os mais diversos. A qualidade do software fica evidentemente muito prejudicada com este comportamento. Estamos acompanhando na década de 2010 uma grande quantidade de aplicações que utilizam a dita Inteligência Artificial. O objetivo deste trabalho é verificar é se podemos e como podemos utilizar a Inteligência Artificial, mais especificamente a área de Machine Learning para auxiliar na detecção precoce de erros de software e melhor consequentemente a qualidade do software.

O primeiro capítulo introduz as questões das pesquisas e suas justificativas. No segundo capítulo são apresentados conceitos básicos e o método que iremos utilizar para responder as questões dessa dissertação.

O terceiro capítulo realiza um apronfundamento bibliográfico em testes de software. O quarto capítulo trata de Machine Learning. O quinto capítulo descreve como é possível utilizar a *Machine Learning* para a detecção de anomalias em software.

Palavras-chave: machine learning. deep learning. detecção de anomalias em software. testes de software. web.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ABNT	Associação Brasileira de Normas Técnicas
ML	Machine Learning
PUCSP	Pontífica Universidade Católica de São Paulo
GUI	Graphic User Interface
IA	Inteligência Artificial
SVM	Support Vector Machine
KNN	K-Nearest Neighbors
ORD	Objetct Relation Diagram
CSS	Cascading Style Sheet
HTML	Hypertext Markup Language
OATS	Orthogonal Array Testing
IMC	Índice de Massa Corpórea
DFD	Data Flow Diagram
UML	Unified Modeling Language
CEP	Controle Estatístico do Processo
CMM	Capability Maturity Model
SEI	Software Engineering Institute
RUP	Rational Unified Process
Web	World Wide Web
	CNN Convolutional Neural Network
	CMM Capability Maturity Model

Lista de símbolos

Γ	Letra grega Gama
Λ	Lambda
ζ	Letra grega minúscula zeta
$\in$	Pertence

Sumário

1 Introdução

1.1 Motivação

A qualidade na indústria de software não seguiu uma evolução tão diversa assim da indústria automotiva. O rumo que a Qualidade de Software tomou na história se iniciou a partir daquela reunião da OTAN em 1968 onde o termo "Engenharia de Software" foi utilizado pela primeira vez por F. L. Bauer.

Naquela reunião foi utilizado também o termo "Crise do Software" para definir a situação em que a indústria do software atravessava naquele momento.

E a crise foi atribuída à complexidade de desenvolver sistemas cada vez maiores, bem como à falta de gerenciamento no processo de desenvolvimento de software.

A partir daí "engenheiros de software" passaram a imitar a engenharia convencional para resolver problemas de qualidade e falhas em sistemas de informação. Uma quantidade significativa de experiência foi obtida através de processos de garantia da qualidade praticados na indústria de manufatura, e essa adaptação para a indústria de software foi, em alguns casos, um fracasso.

Em grande parte, isso se deve ao fato de, embora esta reunião da OTAN beirasse a década de 70, o modelo inicial adotado pela engenharia de software foi orientado fortemente à melhoria do produto final de software, afinal eram neles que estavam os *bugs*. Contudo, se formos seguir a linha da qualidade, esse passo foi equivalente ao seu início quando a preocupação era única e exclusivamente com o produto final.

Ainda no fim da década de 1980, o controle de qualidade existente na indústria de software era centrado no produto final com larga utilização de métodos corretivos e inspeções no fim da linha de produção, ou seja, baseada apenas em testes. Isso se mostrava pouco efetivo para a solução de problemas gerenciais como prazos e custos.

No início da década de 1990, o próprio mercado substituía o Controle pela Garantia da Qualidade com um foco centrado no processo e que utilizava auditorias durante todo o ciclo de vida de desenvolvimento.

A partir daí, foram surgindo normas aplicadas para a área tais como a ISO 9000-3, ISO 15504, ISO 12207, padrões IEEE 1074, IEEE 1298 e Modelos SW-CMM, CMMI e MPS.BR para Qualidade de Software.

Com o advento da Garantia da Qualidade, a indústria de software passou a ser centrada em documentação e orientada a planejamento, e essa forma de desenvolvimento de software ficou conhecida como modelo tradicional [Boehm 1988]. Esta visão se tornou

aquela mais frequentemente aceita como ideal para a qualidade software. Assim, uma empresa CMMI nível 5 era tida como infalível no que tocava a qualidade.

A partir de 1994, os resultados divulgados pelo Standish Group nos relatórios conhecidos como "*Chaos Report*" que mostraram que quase 30 anos depois da Conferência da OTAN, os resultados da indústria de software ainda não são considerados um sucesso.

O relatório aponta que, ainda em 2006, apenas um terço dos projetos de software foram finalizados com sucesso, mesmo com a evolução demonstrada desde a sua primeira edição lançada em 1994, ainda existe uma sensação de pouca evolução.

Segundo (PRESSMAN, 2005) No sentido mais geral, a qualidade de software pode ser definida como: uma gestão de qualidade efetiva aplicada de modo a criar um produto útil que forneça valor mensurável para aqueles que o produzem e para aqueles que o utilizam.

Software sem qualidade

- Projetos de software difíceis de planejar e controlar; custos e prazos não são mantidos.
- A funcionalidade dos programas nem sempre resulta conforme planejado.
- Existem muitos defeitos nos sistemas.
- A imagem da empresa é denegrida no mercado, como empresa tecnologicamente atrasada.

- Projetos, prazos e custos sob controle.
- Satisfação de usuários, com necessidades atendidas na execução de suas tarefas.
- Diminuição de erros nos projetos de software.
- Melhoria da posição competitiva da empresa, como instituição capaz de acompanhar a evolução.

1.2 Questão da pesquisa

Em 1936 Alan Turing escreveu o artigo que introduziu a humanidade na era da computação digital (TURING, 1937) e proporcionou a base teórica de toda nossa computação moderna, nesta caminhada se somaram as propostas da arquitetura do computador digital de Von Neumann (NEUMANN, 1945) em 1945, depois a popularização dos computadores através do *Personal Computer* (PC) na década de 80, nos anos 90 veio a era da Internet comercial, nos anos 2000 houve o advento de uma nova era de empreendimentos baseados nos modelos ditos de cauda longa (TAPSCOTT; WILLIAMS,

2008) e na próxima década a inteligência artificial deverá antigir os níveis humanos quando será capaz de passar por um Teste de Turing válido (KURZWEIL, 2000).

Segundo uma pesquisa de mercado realizado em 2017 pela consultoria de pesquisas de mercado "Research and Markets"(LLP, 2013) o mercado de Inteligência Artificial deve movimentar cerca de US$ 23,4 bilhões de dólares até 2025. Uma cifra grande mas porém pouco representativa perto dos US$ 60 bilhões de dólares (THIBODEAU, 2012) referentes a prejuízos causados por *bugs* de software anualmente.

Atualmente as pessoas tem utilizado a Inteligência Artificial no seu dia a dia com o uso do sistema de recomendações de filmes do Netflix, com a recomendação de ofertas de compras que aparecem na tela de seus computadores baseados potentes sistemas de Machine Learning e alguns já conseguem usufruir do conforto de ter um automóvel guiado por um computador, os ditos veículos autônomos. Em paralelo a esses extraordinários acontecimentos temos presenciado a ocorrência de defeitos em diversos tipos de software, tais como o acidente com o foguete Ariane 5 (LIONS et al., 1996) que causou um prejuízo de US$ 370 milhões de dólares e alguns anos de trabalho cerca de 40s após seu lançamento e pior ainda tais bugs já causaram a morte de pessoas como o incidente com o THERAC-25 que matou 3 pessoas em 1985 (LEVESON; TURNER, 1993), certamente uma perda imensurável para os familiares.

A questão deste projeto é de como a *Machine Learning* pode ajudar na melhoria da qualidade do software, detectando falhas nos software ?

O projeto irá abranger dois campos já bastante estudadas em teoria da computação: testes de software e inteligência artificial.

Muitos autores já vem trabalhando em ambas as áreas desde a década de 30, na área de Inteligência Artificial desde Alan Turing até Peter Norvig. A área de testes também possui diversos autores que tem contribuido para seu crescimento, para citar alguns como Pressmann e Eduardo Delamaro. Na atualidade os software são testados principalmente de duas formas: automatizada e manual. O teste manual de software é feito através do emprego de pessoas interagindo com o software geralmente seguindo um roteiro previamente elaborado por desenvolvedores de testes. O teste automatizado é feito com o auxílio de outro software que automatiza os testes enviando dados de entrada e recebendo e analisando os dados de saída a partir de parâmetros previamente estabelecidos pelo programador do teste (DELAMARO; JINO; MALDONADO, 2017).

Geralmente o teste manual costuma apresentar uma baixa cobertura de teste de software, isto é , ele testa um pequeno percentual dos parâmetros que o software pode receber, isso se deve principalmente a característica do ser humano não ser tão veloz quanto um computador para inserir e ler dados. O teste automatizado aumenta o percentual de cobertura pelo fato de testar tudo o que foi programado quando necessário sem dispender

muito tempo, no entanto para a execução de novos testes é necessário que se programe novos testes, o que eventualmente custa tempo e dinheiro para ser feito.

A proposta de realizar testes feitos por uma inteligência artificial ainda é relativamente escasso, podemos apresentar alguns trabalhos como os que seguem:

Murphy foca o trabalho na questão dos parâmetros de saída de programas que utilizam Inteligência Artificial não terem um único valor de saída (MURPHY; KAISER; ARIAS, 2007).É sabido que a maioria dos programas que utilizam IA geralmente possuem uma resposta aceitável.No entanto nem sempre esta é a melhor resposta (RUSSELL; NORVIG, 1995), o foco do seu trabalho é de realizar testes que determinem qual é essa faixa de valores aceitáveis. Ferreira por sua vez trabalha focado na questão da criação de um agente inteligente para testar o software resultante de implementações de metodologia ágil (FERREIRA, 2017) As metodologias encontradas em *Artificial Intelligence Methods in Software Testing* (ABRAHAM; HORST, 2004) são voltadas ao uso de redes neurais e Lógica *Fuzzy* (ABRAHAM; HORST, 2004).

1.3 Justificativas

As notícias sobre a utilização de Inteligência Artificial são cada vez mais frequentes e as que reportam defeitos milionários em softwares infelizmente também o são. Uma das principais propostas do projeto é a de aumentar os testes de cobertura de software contribuindo para a melhoria da qualidade do software.

Alguns autores como Frederico Ferreira já trilharam o caminho de realizar testes de software com a utilização de Inteligência Artificial, no entanto raras são os que abordam a questão utilizando as técnicas de *Machine Learning*.

A linha a ser adotada nesta pesquisa é diferente das apresentadas pelos autores citados na seção anterior, pois ela trabalha com resultados provenientes de testes de software já realizados pelos métodos tradicionais (KOSCIANSKI; SOARES, 2007).

1.4 Objetivos

A proposta deste projeto de pesquisa é de desenvolver uma nova arquitetura de testes de software para o aumento de cobertura de testes.

A proposta desta pesquisa não é a de substituir a metodologia e arquitetura de testes propostos por autores como Delamaro (DELAMARO; JINO; MALDONADO, 2017), Koncianski (KOSCIANSKI; SOARES, 2007) ou pelas melhores práticas recomendadas pelo SWEVBOK (BOURQUE; FAIRLEY et al., 2014), mas sim aumentar a cobertura de testes de software.

1.5 Hipóteses

Hipótese:

A minha principal hipótese e que o uso de *Machine Learning* pode aumentar os testes funcionais de cobertura de software.

As limitações das técnicas de *Machine Learning* é que as mesmas não apresentam 100% de acurácia na maioria das vezes. A acurácia de tal técnica é melhorada incrementalmente a cada interação de aprendizado pelo qual a máquina passa, isto pode vir a apresentar como um ponto que não corrobora a hipótese.

2 Conceitos e Métodos

Foi utilizada o método de pesquisa descritiva para propor o modelo apresentado no capítulo VI, partindo do uso de uma bibliografia que aborda principalmente testes de software e Machine Learning, com a utilização de livros e artigos de vários autores da área da computação e alguns dados de sítios web como fontes secundárias.

A primeira parte da pesquisa trata dos "Testes de Software", foram utilizadas diversas fontes bibliográficas mas os autores que foram utilizados com mais profundidade foram Delamaro Maldonado(DELAMARO; JINO; MALDONADO, 2017) e Koscianski(KOSCIANSKI; SOARES, 2007). A segunda parte trata de *Machine Learnig* e as principais fonte bibliográfica utilizada foram Kubat (KUBAT, 2015) e Ethem Alpaydın (ALPAYDIN, 2017).

Na terceira parte os dois conceitos apresentados foram unidos em um modelo proposto para a detecção de anomalias durante a execução do software. No capítulo são apresentadas algumas ferramentas software que podem ser utilizados como parte do modelo proposto.

3 Testes de Software

3.1 Erro, Defeito e Falha de software

De acordo com a terminologia utilizada na Engenharia de Software do IEEE (Institute of Electrical and Electronics) (STANDARD, 1990):

- **Defeito** é um ato inconsistente cometido por um indivíduo ao tentar entender uma determinada informação, resolver um problema ou utilizar um método ou ferramenta. Exemplo: comando incorreto
- **Erro** é uma manifestação concreta de um efeito num artefato de software. Diferença entre o valor obtido e o valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução de um programa constitui um erro.
- **Falha** é o comportamento operacional do software diferente do esperado pelo usuário. Uma falha pode ter sido causada por diversos erros e alguns erros podem nunca causar uma falha.

Os prejuízos referentes a atrasos de projetos e falhas de programação de software nos acompanham pelo menos há algumas décadas (BROOKS, 1987).

Em quatro de Junho de 1996, o foguete Ariane 5 explodiu a 3700m e 40s após o seu lançamento, a causa da falha foi o que se costuma chamar de *stack overflow* (LIONS et al., 1996).Diversos testes haviam sido feitos com o Ariane 4, porém ao migrarem os testes para o software do Ariane 5 alguns erros passaram desapercebidos, no software do Ariane 4 havia uma conversão de um variável double para int que não causava problemas no Ariane 4, no entanto esse erro causou a explosão do Ariane 5 e um prejúizo estimado de 370 milhões de doláres.

Em 1993 o Dr. Nicely descobriu que o processador Pentium da Intel falhava em uma divisão longa, estima-se que este erro causou o prejuízo de cerca de meio bilhão de dólares para a Intel (ATHOW, 2014).

Os prejuízos relacionados a erros de software eram da ordem de US$ 60 bilhões de doláres anuais, em 2016 este número saltou para US$ 1,1 trilhão (COHANE, 2017).

No entanto o mais triste de tudo é que os prejuízos não são só de ordem econômica, software também matam. Entre 1985 e 1987 um equipamento de radioterapia causou a morte e lesão de diversos pacientes devido a um erro de software (LEVESON; TURNER, 1993). O Therac 25 foi o primeiro de sua série contar com controles baseados em um

software operado por um computador digital, o seu antecessor Therac 24 possuia sistemas de controle e proteção baseados em circuitos elétricos-analógicos. Em uma condição específica de operação o Therac 25 apresentava um estado que chamamos de "condição de corrida"e ocasionava uma falha operacional no qual exibia uma mensagem que informava para os seus operadores que nenhuma dose de radiação estava sendo ministrada aos pacientes, no entanto ele estava enviando doses letais de radiação.

3.2 Ciclo de Vida do Software

O ciclo de vida de um software pode ser compreendido como toda a estrutura necessária contendo processos, tarefas e atividades envolvidas no desenvolvimento, operação e manutenção de um software. O ciclo de vida completa do sistema, desde a concepção dos requisitos do sistema até o seu término de uso.

Geralmente o modelo de ciclo de vida de um software é a primeira escolha que deve ser feita no desenvolvimento do software. Existem diversos tipos de ciclo de vidas que podem ser utilizados para a criação de um software, o que as diferencia é a ordem em que suas fases ocorrerão e a ênfase dada a cada fase, alguns modelos dão mais ênfase na concepção enquanto outros procuraram dar mais ênfase na fase de testes.

Com o crescimento do mercado de software, houve uma tendência a repetirem-se os passos e as práticas que deram certo. A etapa seguinte foi a formalização em modelos de ciclo de vida.

A segui alguns exemplos que podem ser encontrados na literatura sobre os ciclos de vida do software, no entanto a escolha depende de cada caso, onde devem ser levados em conta a complexidade do negócio, o custo, o conhecimento e experiência da equipe acerca das ferramentas e do negócio.

- Cascata
- Modelo em V
- Incremental
- Evolutivo
- RAD
- Prototipagem
- Espiral
- Modelo de Ciclo de Vida Associado ao RUP

3.3 Qualidade de Software

A qualidade de software é uma área do conhecimento da engenharia de software que pode se referir a: "as características desejadas de produtos de software, a extensão em que um produto de software em particular possui essas características e aos processos, ferramentas e técnicas que são usadas para garantir essas características"(BOURQUE; FAIRLEY et al., 2014).

Os conceitos de qualidade que a industria automobilística utiliza são bem similares ao do desenvolvimento de software, a industria automobilística existe desde o inicio do seculo XX e a o desenvolvimento de software desde a década de 1950 (KOSCIANSKI; SOARES, 2007). No entanto a quantidade de defeitos encontrados em carros é da ordem de várias grandezas inferior a encontrados em software, o que vem ocasiona essa diferença ?

3.3.1 Qualidade do processo

A qualidade do processo enfoca na ferramentas e processos necessárias para como se deve construir um artefato com o menor número de defeitos possíveis, seja ele um software ou um computador. A industria de manufatura de automóveis trouxe importantes contribuições nessa área, principalmente através das ferramentas e técnicas desenvolvidas pelos japoneses da Toyota, algumas das ferramentas são utilizadas até hoje inclusive na área de desenvolvimento de software, como o Kanban (SUGIMORI et al., 1977). Na década de 80 o Departamento de defesa norte americano preocupado com a baixa previsibilidade de custos e qualidades dos softwares contratados, instituiu junto a Carnegie-Mellon University o SEI (Software Engineering Institute) (PAULK et al., 1993) que elaboraram um modelo chamado Capability Maturity Model (CMM).

O CMM pode ser definido como a soma de "melhores práticas"para diagnóstico e avaliação de maturidade do desenvolvimento de softwares em uma organização. Espera-se que quanto mais maturidade uma organização tenha em seus processos, menor será a quantidade de defeitos e falhas em seus softwares, por conseguinte melhor a qualidade do software construído.

3.3.2 Qualidade do produto

O processo de verificação da qualidade do produto foca na inspeção do produto final, se o artefato que foi fabricado atende aos requisitos de qualidades esperados dele. Geralmente as técnicas empregadas aqui são as medições e comparações do resultado com o esperado, por exemplo: Se o parafuso deveria medir entre 10.1mm e 10.2mm e está com 10.4mm , ele não atende a qualidade necessária.

Uma das principais ferramentas utilizadas no controle da qualidade do produto na manufatura é o Controle Estatístico do Processo (CEP) (SUGIMORI et al., 1977). Na área de software atualmente utilizamos principalmente os testes de software, que visam verificar que para determinados parâmetros de entrada, as saídas correspondentes estarão dentro dos valores esperados.

Neste ponto já podemos perceber que ocorrem algumas diferenças grandes entre o ambiente de manufatura industrial e desenvolvimento de software. No CEP a inspeção é feita por amostragem de peças, sendo que se temos amostras fora do padrão de qualidade estabelecido é relativamente fácil identificar que e falha está em alguma parte do processo de manufatura que geralmente já é bem conhecido da equipe de manutenção. No desenvolvimento de software a amostragem não garante com uma precisão grande que o software estará livre de defeitos e falhas, testar todos as entradas possíveis também é praticamente impossível (DELAMARO; JINO; MALDONADO, 2017).

A construção de software é uma atividade de engenharia e como tal trabalha com situações onde o projetado e o realizado são muitas vezes diferentes e para que essas diferenças não inviabilizem o produto final, são estabelecidas margens de erros que tornam o produto aceitável ou não.

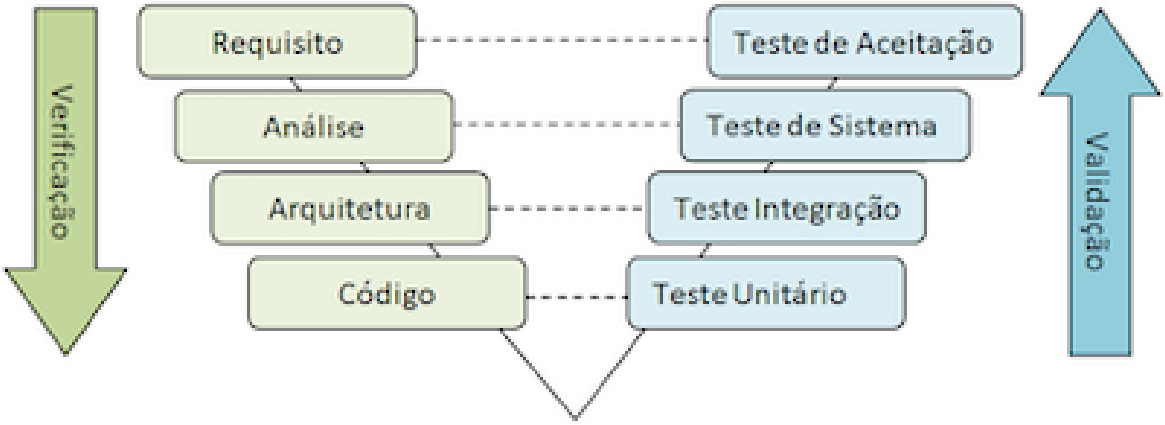
Desde a década de 60 (KOSCIANSKI; SOARES, 2007) a disciplina de Engenharia de Software tem atacado o problema de se ver livre dos defeitos e erros de software de diversas maneiras:

- Com a análise de requisitos
- Com novas técnicas de testes
- Com novas linguagens de programação
- Com novas ferramentas de qualidade
- Com novas ferramentas de gestão de projetos
- Com novas linguagens de modelagem de software (DFD, UML)
- Com ferramentas CASE

A construção de um software geralmente envolve a comunicação entre diversas pessoas, cabe geralmente ao desenvolvedor de software entender e interpretar o que os outros interlocutores desejam construir ou qual problema desejam resolver com o software, neste ponto acabam surgindo erros. Por mais que se criem metodologias de desenvolvimento de software e ferramentas para nos auxiliar a minimizar a ocorrência desses erros eles surgem e a maioria dos erros tem como causa o erro humano (DELAMARO; JINO; MALDONADO, 2017).

Existem diversos tipos de abordagem de testes de software, na Figura 1 mostra um dos modelos clássicos de ciclo de teste de software, intimamente ligado com o ciclo de desenvolvimento.

Figura 1 – Modelo V - Fonte Wikipedia



No modelo vemos claramente que os testes de software estão relacionados com os estágios de desenvolvimento de software:

- Teste de Unitário x Código: Foco na menor unidade do programa, geralmente uma parte que não pode ser subdividida. Esperado encontrar erros referentes a algoritmos incorretos, estrutura de dados incorretas ou erros simples de programação
- Teste de Integração x Arquitetura: Realizado após os testes unitários, o foco aqui é testar se todas as unidades funcionam trabalhando juntas. Esperado encontrar erros referentes a arquitetura do software e do sistema.
- Teste de Aceitação/Sistema x Requisito/Análise: Realizado após a integração de todas as partes, é o ultimo testes realizado pela equipe de desenvolvimento / testes antes da entrega para aceite. Nesta fase é esperado que todos os requisitos sejam cumpridos.

Para que erros de software não permaneçam desconhecidos e/ou sejam descobertos tarde demais, muitas vezes ocasionando vítimas fatais, existe uma série de atividades coletivamente chamadas de "Validação, Verificação e Teste"ou "VV&T", com a finalidade de garantir que tanto o processo de construção de software como o produto em si estejam conforme o que foi especificado(DELAMARO; JINO; MALDONADO, 2017).

3.3.3 Escopo do Teste

Analisando os conceitos expostos nas seções anteriores, podemos concluir que um software para não apresentar falhas e nem possuir defeitos o programador não deve cometer erros ao implementar o software.

Vamos então tentar escrever um software livre de erros e iremos considerar que ele será executado em um computador digital moderno que é uma implantação da arquitetura proposta por Von Neumann (NEUMANN, 1945), isto é o computador possui limitações físicas.

Iremos implantar a seguinte função em um computador:

$$f(x, y) = x^y \tag{3.1}$$

Para garantirmos que nosso software está livre de erros poderíamos bastar todas as entradas possíveis. Analisando a função f(x,y) produz um conjunto de cardinalidade:

$$Cardinalidade = 2^n * 2^n \tag{3.2}$$

onde n é o número de bits usado para representar um inteiro, em um computador de 32 bits:

$$Cardinalidade = 2^{32} = 18446744073709551616 \tag{3.3}$$

Se conseguíssemos efetuar cada teste em 1ms, levaríamos 5.849.424 séculos para terminar de testar tudo (DELAMARO; JINO; MALDONADO, 2017).

Apesar de demorado e inviável, matematicamente não é impossível realizar tal teste desde que tenhamos uma função matemática discreta bem definida, isto é:

- parâmetros de entradas bem definidos, isto é eles sejam claros e definidos do ponto de vista matemático, exemplo: parâmetro n, onde n é um inteiro de 0 a 10.
- para cada conjunto de parâmetros exista uma ou mais saídas aceitáveis.

Exemplo de uma função bem definida

$$f(x, y) = x^y \tag{3.4}$$

onde

$$1 \leq x \leq 3 \tag{3.5}$$

$$1 \leq y \leq 3 \tag{3.6}$$

Teremos então as seguintes entradas e saídas:

Tabela 1 – função simples f(x,y)

f(x,y)	1	2	3
1	1	2	3
2	2	4	8
3	3	9	27

Já é possível notar que é possível porém inviável tentar testar todos os parâmetros em funções mais complexas que a exposta acima. Na verdade o teste completo de software é praticamente inviável (DIJKSTRA, 1972)

Se não podemos testar todas as entradas possíveis então talvez possamos tentar definir que todas as entradas sempre obedeçam a parâmetros bem estabelecidos. No entanto isso continua sendo uma afirmação de natureza não tão simples assim. A construção de software sempre será uma tarefa árdua, pois uma das essências na construção de software é a complexidade de sua natureza (BROOKS, 1987).

Segundo (DELAMARO; JINO; MALDONADO, 2017) ,como o testes de todas as entradas possíveis é infactível devemos selecionar critérios adequados para a realização dos nossos testes, para se obter um mínimo nível de qualidade para os conjuntos adequados a um critério C para um dado programa P, deve se:

- exista um conjunto C adequado que seja finito e de preferência de baixa cardinalidade
- que o programa P execute uma função / comando de cada vez (não haja paralelismo em execução)
- do ponto de vista de utilização das variáveis , cada atribuição de valor a uma variável tenha seu valor verificado por um caso de teste que execute o trecho do programa em que esse valor é utilizado.

3.4 Tipos de Testes de Software

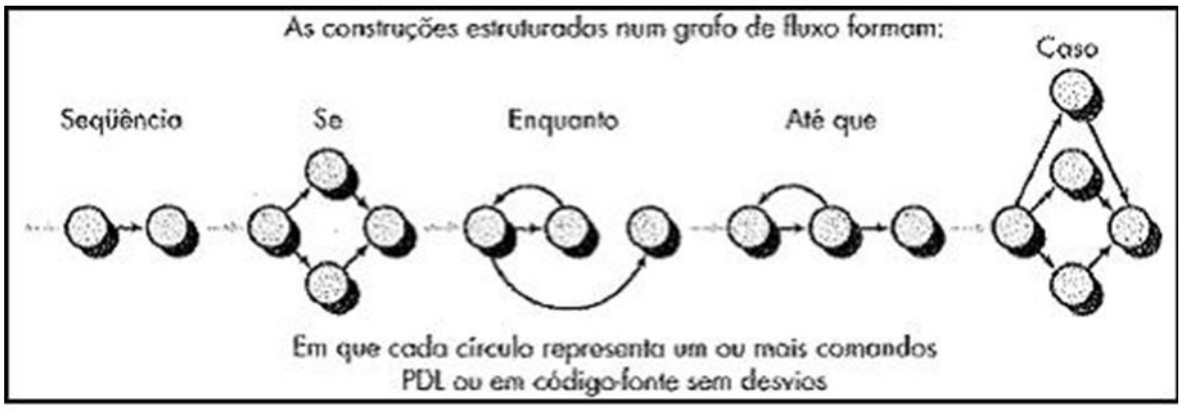
3.4.1 Introdução

Há muitas metodologias de testes e formas de abordar os testes propostas na literatura atualmente e não seria o propósito tratá-las delas aqui, abordaremos 4 tipos de testes que considero mais apropriados para a metodologia de teste desta dissertação.

Porém antes de abordarmos os tipos de testes de software é necessário expor alguns conceitos relacionados a complexidade do software.

Para um determino programa P executar uma função f(x) ele deve passar um fluxo de execução, no qual pode haver parâmetros de entrada e a saída, durante a execução um programa pode passar por diversos estados e caminhos, a figura 2 apresenta os principais caminhos e condições em um fluxo de execução de software. O fluxo no caso é apresentado por um grafo composto de nós e arestas.

Figura 2 – Caminho Básico



Fonte: (PRESSMAN; MAXIM, 2016)

A figura 3 apresenta um software um exemplo de um software de baixa complexidade, no entanto já podemos identificar que a medida que o software se torna mais complexo o número de nós e arestas cresce assim como o número de caminhos possíveis.

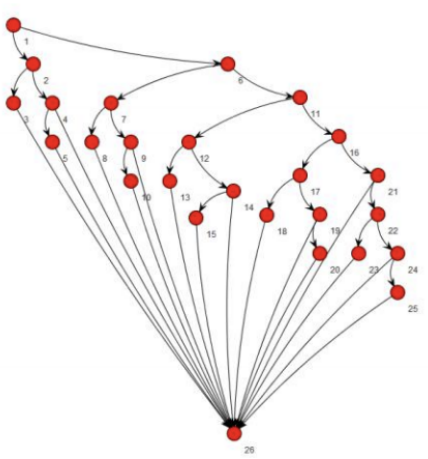
(MCCABE, 1976) propôs em 1976 uma métrica de software chamada complexidade ciclomática para mensurar a complexidade de um software. Ele mede a quantidade de caminhos de execução independentes a partir de um código fonte, isto é ele mensura a quantidade de nós e arestas que um programa pode percorrer.

3.4.2 Teste Funcional

O teste funcional é uma técnica utilizada sob o ponto de vista do usuário, o software é considerado uma caixa preta onde os detalhes da sua implementação não são considerados (DELAMARO; JINO; MALDONADO, 2017), sendo assim o que importa é se as entradas e saídas do programa estão de acordo com a expectativa do usuário.

A priori o teste funcional é capaz de detectar qualquer defeito que exista no software a partir do domínio de entrada, neste caso seria realizado o teste com todo o domínio de entrada , isto é chamado teste exaustivo, no entanto como já abordado anteriormente esta abordagem é quase sempre inviável.

Figura 3 – Incremento na complexidade de software



```
1  public float getValorPassagem(){
2          milhas = 300;
3          if (this.milhas <= 400) {   //1
4              if (this.tipoPessoa.equalsIgnoreCase("criança")) {  //2
5                  valorPassagem = 1.5F;       //3
6              } else if (this.tipoPessoa.equalsIgnoreCase("adulto")) {        //4
7                  valorPassagem = 1.8F;           //5
8              }
9          } else if (this.milhas <= 800) {            //6
10             if (this.tipoPessoa.equalsIgnoreCase("criança")) {      //7
11                 valorPassagem = 1.6F;           //8
12             } else if (this.tipoPessoa.equalsIgnoreCase("adulto")) {        //9
13                 valorPassagem = 1.9F;               //10
14             }
15         } else if (this.milhas <= 1500) {           //11
16             if (this.tipoPessoa.equalsIgnoreCase("criança")) {          //12
17                 valorPassagem = 1.7F;               //13
18             } else if (this.tipoPessoa.equalsIgnoreCase("adulto")) {            //14
19                 valorPassagem = 2F;         //15
20             }
21         } else if (this.milhas <= 2500) {               //16
22             if (this.tipoPessoa.equalsIgnoreCase("criança")) {          //17
23                 valorPassagem = 1.9F;               //18
24             } else if (this.tipoPessoa.equalsIgnoreCase("adulto")) {            //19
25                 valorPassagem = 2.15F;              //20
26             }
27         } else if (this.milhas > 2500) {            //21
28             if (this.tipoPessoa.equalsIgnoreCase("criança")) {          //22
29                 valorPassagem = 2F;         //23
30             } else if (this.tipoPessoa.equalsIgnoreCase("adulto")) {            //24
31                 valorPassagem = 2.5F;           //25
32             }
33         }
34         return valorPassagem * this.milhas;  //26
35 }
```

Fonte: (JUNIOR; PRADO; ARAÚJO, 2016)

3.4.2.1 Critérios

Os critérios mais conhecidos na técnica de teste funcional são:

- Particionamento em classes de Equivalência

  Uma vez que o teste exaustivo é inviável uma das estratégias e particionar o domínio de entrada e/ou de saída em partições que são tratadas da mesma forma pelo programa, isto é são equivalentes. Dessa forma o critério reduz o domínio para um tamanho passível de ser tratado durante os testes.

  Um exemplo pode ser dado através de um programa que calcula o Índice de Massa Corpórea (IMC) de uma pessoa:

$$IMC = Peso(Kg)/Altura(m)^2 \tag{3.7}$$

Vamos criar uma função que calcula o IMC:

```
1   #include<stdio.h>
2   #include<stdlib.h>
3   int main(void) {
4       float P, H, IMC;
5
6       printf("Digite o seu peso:\n");
7       scanf("%f", &P);
8
9       printf("Digite a sua altura:\n");
10      scanf("%f", &H);
```

```
11              IMC = P / (H*H);
12              if (IMC < 26){
13                  printf("O seu Indice de Massa Corporea e %.2f e voce
                        esta Normal\n", IMC );
14              }
15              if (IMC >=26<30){
16                 printf ("O seu Indice de Massa Corporea e %.2f e voce
                        esta Obeso\n", IMC);
17              }
18              if (IMC >=30){
19                 printf("O seu Indice de Massa Corp rea e %.2f e voce
                        esta com Obesidade M rbida\n", IMC);
20              }
21              system("pause");
22      }
```

Analisando o programa acima podemos verificar que o domínio de entrada pertencem ao conjunto dos números inteiros, neste caso poderíamos dizer que o domínio de entrada é:

$$-\infty < Peso < +\infty \tag{3.8}$$

$$-\infty < Altura < +\infty \tag{3.9}$$

Vale notar que aqui o infinito é teórico, pois o limite é dado pelo tamanho do "*float*"computável pelo programa e hardware disponível.

Como não é possível testar o domínio de entrada na sua totalidade, podemos particioná-lo em 2 grupos:

Dados que podem ser considerados válidos:

$$4 \leq Peso(Kg) \leq 300 \tag{3.10}$$

$$0.5 \leq Altura(m) \leq 2.5 \tag{3.11}$$

Dados que podem ser considerados inválidos:

$$Peso < 4, Peso(Kg) > 300 \tag{3.12}$$

$$Altura < 0.5, Altura(m) > 2.5 \tag{3.13}$$

Para realizar o teste neste caso poderíamos criar 2 grupos de equivalência:

  - Grupo de valores de Peso(Kg) situados entre 4Kg e 300Kg com Alturas(m) entre 0.5m e 2.5m, cujo IMCs fiquem abaixo de 26, entre 26 e 30, e acima de 30, desta maneira podemos testar as arestas do fluxo de execução do programa.
  - Grupo de valores abaixo de 4Kg, acima de 300Kg e Alturas abaixo de 0.5m e acima de 2.5m, com Pesos e Alturas negativas também.

Analisando essas entradas propostas podemos afirmar que o programa irá apresentar defeitos, como alturas e pesos negativos e se colocarmos a altura em zero, irá ocorrer uma divisão por zero.

Um dos pontos fortes de critério é a grande redução do domínio de entrada, por outro lado seu ponto fraco é a subjetividade da análise para a criação dos grupos de equivalência (DELAMARO; JINO; MALDONADO, 2017).

- Análise do Valor Limite

  Segundo (MYERS; SANDLER; BADGETT, 2011) a experiência mostra que os os casos de testes que exploram as condições dos limites dos parâmetros do programa tem maior probabilidade de encontrar defeitos.

  No exemplo mostrado acima , poderíamos testar os limites de peso e altura, bem como os IMCs 26 e 30. Pois estes são considerados os pontos limites do programa, vamos supor que o programador houvesse se esquecido de testar a condição igual a 26 e/ou igual a 30, isso iria incorrer em um erro detectável.

- Teste Funcional Sistemático

  Neste teste é realizado o Teste de Grupo de equivalência junto ao teste de Análise de Valor limite, pois ao mesclarem os 2 a chance de cobrir falhas se torna muito maior (DELAMARO; JINO; MALDONADO, 2017).

- Teste Combinatorial

  O teste combinatorial procura analisar as combinações de parâmetros de entrada e procura testar essas diversas combinações. O trabalho de (KUHN; WALLACE; GALLO, 2004) é um dos primeiros trabalhos a apresentar a relação entre as combinação de variáveis e as falhas, sendo que foi concluído que grande parte das falhas era ocasionada pela combinação de duas variáveis e progressivamente menos entre 3,4,5 e no máximo 6 variáveis.

  Uma forma de implantação do teste combinatorial é a utilização do Orthogonal Array Testing(OATS) que é uma técnica especial de teste funcional , fundamentada de forma estatística e sistemática que procura criar diversas combinações de entrada em um *array* e submeter o programa a estas entradas. Por meio do emprego de OATS é possível maximizar os testes de cobertura e minimizar o número de caso de testes a serem combinados.

- Error Guessing

  Esta é uma técnica bastante subjetiva e que depende muito da experiência do analista de teste que procura através de sua intuição e experiência identificar situações onde erros prováveis podem ocorrer e montar casos de testes para detectá-los.

3.4.3 Teste Estrutural

A técnica estrutural é também chamada teste de caixa branca, pois estabelece requisito de testes baseados na implantação do programa.

Esta técnica procura modularizar os componentes do software e testar cada parte deles individualmente ou testar o fluxo de dados em tempo de execução.

Esta técnica apresenta uma série de limitações e desvantagens com relação ao teste funcional, pois não há um requisito geral que possa ser utilizado para provar a existência de um defeito no software.

Os testes são feitos em partes do software, decompondo-o e apresentando entradas e comparando com saídas esperadas para cada entrada no módulo decomposto.

Segundo (DELAMARO; JINO; MALDONADO, 2017), há ainda as seguintes limitações inerentes a técnica estrutural:

- caminhos ausentes: se o programa não implementa algumas funções, nunca haverá um caminho que percorra esta função para testá-la.
- correção coincidente: o programa pode apresentar um falso positivo, uma vez que para uma determinada entrada ele pode apresentar um resultado verdadeiro que com outra entrada seria falsa.

Para a metodologia de pesquisa que será apresentado nos próximos capítulos veremos que a técnica estrutural não é interessante de ser utilizada, pois não iremos querer os detalhes de implementação do software.

3.4.4 Teste de Mutação

Como já exposto anteriormente um bom teste é aquele que revela defeitos, já vimos que a divisão dos domínios de entrada em subdomínios nos fornece uma boa maneira de revelarmos defeitos no entanto não existe uma relação direta entre um subdomínio e a capacidade de um caso de teste dele extraído revelar defeitos (DELAMARO; JINO; MALDONADO, 2017).

O Teste de mutação é uma técnica baseada em defeitos, nele são utilizados defeitos típicos do processo de programação para que sejam derivados casos de testes.

A teoria define que :

Um programa **P** é correto com relação a uma função **F**, se **P** computa **F**.

E para provar a correção de um programa, um conjunto de teste deve ser confiável de acordo com a definição de (HOWDEN, 1976):

> Um conjunto de teste **T** é confiável para um programa **P** e uma função **F**, dado que **F** e **P** coincidem em **T**, se e somente se **P** computa **F**, Se **P** não computa **F**, então, **T** deve conter um ponto t tal que:

$$F(t) \neq P(t) \tag{3.14}$$

A teoria mais aprofundada que fundamenta os testes de mutação pode ser encontrada em (DELAMARO; JINO; MALDONADO, 2017).

A proposta dos testes de mutação é de produzir variações no código fonte próximas ao programa original onde há variações, em seguida gera se um teste T para cada novo programa P.

O teste de mutação é feito em 4 etapas:

- geração dos mutantes,

  Os mutantes são gerados a partir dos operadores de mutação, exemplos:

  - SSDL: eliminação de comandos
  - ORRN: troca de operador relacional
  - STRI: armadilha em condição de comando if
  - Vsrr: troca de variáveis escalares

- execução do programa em teste

  Nesta etapa deve se executar o programa **P** com o testes de casos selecionados e verificar se o valor corresponde ao esperado, caso sim devemos prosseguir para a próxima etapa, caso contrário já detectamos um erro e devemos corrigir o programa **P**.

- execução dos mutantes

  Nesta etapa cada um dos mutantes é executado através dos casos de testes **T**. Se um mutante **M** apresenta resultado diferente de **P**, isto significa que o teste expôs a diferença entre **P** e **M** então ele é descartado e é considerado morto, caso contrário ele continua vivo e pode indicar um sintoma de falhas de software.

Após a execução dos mutantes é feito o cálculo do "escore de mutação":

$$ms(P,T) = \frac{DM(P,T)}{M((P) - EM(P)} \tag{3.15}$$

onde:

DM(P,T): número de mutantes mortos pelo conjunto de casos de teste T;

M(P): número total de mutantes gerados a partir do programa P e;

EM(P): número de mutantes gerados que são equivalentes a P.

- análise dos mutantes

  Nesta etapa são executados todos os mutantes e verifica-se o resultado de cada um e calcula-se o escore de mutação, o score de mutação varia de 0 a 1.

  Caso o score seja perto perto de 1 significa que o programa está perto de estar livre de erros, quanto mais próximo de zero maior a probabilidade do programa conter erros.

  Caso a primeira interação não resulte em um score satisfatório outra interação é feita com os mutantes vivos e um novo caso de teste, e depois é calculado um novo score. São realizadas a quantidade de interações que o testador julgar necessário para alcançar um score satisfatório.

3.4.5 Teste de Aplicações Web

Uma aplicação *Web* é em geral composta por uma base de dados, página *Web* geralmente com alguma linguagem sendo executada que disponibiliza páginas em HTML, com os quais os usuários interagem através de um *Web browser*.

As aplicações Web costuma ser muito heterogêneas, eles frequentemente utilizam diversos componentes de diversas tecnologias, como: Servidores Web, Linguagens de Programação no Back End (Java, PHP, Python, Ruby, C, Go, etc.), Diferente Banco de Dados (Posgtresql, Mysql, MariaDB, MongoDB,etc.) , Javascript, CSS e HTML. Isto faz com que atributos de compatibilidade e interoperabilidade sejam fundamentais em aplicações *Web*.

O dinamismo que as aplicações *Web* traz dificuldades as atividades de testes, pois a sequência de operações realizadas pelo usuário pode trazer à tona erros inesperados de execução do programa.

Outra característica dos programas *Web* é que eles podem ser executados no *web browser* do usuário alterando dados não esperados.As aplicações também são alteradas constantemente devido as características de atendimento de requisitos do usuário, devido a isso o tempo de desenvolvimento e manutenção deve ser o menor possível.

As diferentes tecnologias e *framerworks* que compõem uma aplicação Web também estão sempre em constante evolução e muitas vezes são substituídas rapidamente, algumas vezes várias vezes durante o dia (BASS; WEBER; ZHU, 2015).

Muitas proposta para o teste de aplicações *Web* são encontradas na literatura, a seguir são apresentados alguns dos principais testes utilizados mais recentemente:

3.4.5.1 Teste Estrutural

Alguns trabalhos de autores procuram dividir as aplicações web em estruturas que possibilitam o teste da caixa branca. Os principais elementos envolvidos nesta divisão são: página cliente, página servidora e componente (template HTML, ActiveX Control ou qualquer módulo de programa que interaja com uma página cliente, com uma página servidora ou outro componente.

Em uma modelagem orientada a objeto, os elementos da aplicação Web são considerados objetos com relações entre elas, a representação de tal modelo é chamada *Object Relation Diagram* - ORD. Neste modelo os elementos acimas são divididos e relacionados de acordo com seu tipo de interação. Para mapeamento e captura de informações do fluxo de controle e de dados que ocorrem durante a execução do programa (LIU et al., 2000) propõe quatro grafos:

- Grafo de Fluxo de Controle, utilizada um grafo para cada função
- Grafo de Fluxo de Controle Interprocedimental, utilizada para fluxo de dados que utilizam mais de uma função
- Grafo de Fluxo de Controle de Objeto, representa as associações de diferente sequências de invocação de funções no objeto,
- Grafo de Fluxo de Controle Composto, descreve o fluxo de informação entre as páginas Web,

(RICCA; TONELLA, 2001) propõe a divisão dos testes de aplicações *Web* em dos grupos:

3.4.5.2 Realização de teste estático

Neste teste procura-se encontrar defeitos relacionados a:

- Páginas que não podem ser acessadas por falta de *link* para elas
- Páginas cruzadas - páginas não acessíveis a partir da página inicial
- Páginas quebradas - url referenciadas que são inexistentes

3.4.5.3 Realização de teste dinâmico

Neste teste várias páginas são percorridas segundo o fluxo de teste estabelecido pelo teste, carregando váriaveis de uma página para outra, (RICCA; TONELLA, 2001) propõem os seguintes critérios:

- Teste de página - cada página deve ser acessada ao menos uma vez
- Teste de link - cada link deve ser percorrido ao menos uma vez
- Teste de definição-uso - todos os caminhos que que constam nos requisitos devem ser feitos ao menos uma vez
- Teste de todos-usos - ao menos um caminho que constam nos requisitos devem feitos uma vez.
- Teste de todos-caminhos - cada caminho na aplicação deve ser percorrido ao menos uma vez.

Os testes proposto por acima por (RICCA; TONELLA, 2001) são amplamente utilizados atualmente em ferramentas como o SEMRush, que procura exatamente testar a parte estática de sítios de Internet.

3.4.5.4 Testes baseados na Interface Gráfica do Usuário *GUI*

Outra abordagem bastante utilizada atualmente é a de interação diretamente com a Interface Gráfica que o usuário vê em seu dispositivo , seja ele desktop , celular ou tablet.

Com o aumento nas últimas décadas no diversos números de dispositivos que acessam a *Web* com diferentes formatos de tela, este tipo de teste tem se demonstrado amplamente vantajoso.

Segundo (NGUYEN et al., 2014) pesquisadores e desenvolvedores concordam que para o teste baseado em GUI ser efetivo é necessário o uso de várias técnicas combinadas como "teste baseado em modelo", *scripts* manuais e captura de telas. O autor em seu estudo propõe a criação de um software de teste chamado GUITAR que se baseia em plugins que utilizam uma técnica de análise estrutural de fluxo para desenvolver os testes.

Na figura 4 é demmonstrado o comparativo entre algumas ferramentas de teste de software e a necessidade que se tem de escrever *script* manuais para cada caso de teste, repare que para o GUITAR não é necessário escrever *scripts*.

No mercado atualmente já existem algumas ferramentas baseadas em *Deep Learning* que identificam também desformatação mo leiaute de páginas *Web*, avisando o mantenedor do software sobre a existência do problema caso ele ocorra, como o *Ghost Inspector* https://ghostinspector.com/, por exemplo.

Figura 4 – Compação entre softwares que utilizam GUI para testes

Table 1 Comparison of testing frameworks

Framework name	Model generation	Model verification	Test case generation	Test oracles	Supported platforms
GUITAR	Rev. Eng (A)	Manual	Model based (A)	Custom	Multiple†
Monkey	None	None	None	Supported events	Android
NModel	Scripted (M)	Tool support (M)	Model based (A) & Scripted (M)	Custom	C#
Quick Test Pro	None	None	Scripted (M) & Captured (M)	Custom	Multiple‡
Selenium	None	None	Scripted (M) & Captured (M)	Custom	Web

A = automated, M = manual

† JFC, SWT, Web, Android, other platforms in alpha

‡ Java, Web, .NET

Fonte: (NGUYEN et al., 2014)

4 Machine Learning

4.1 Inteligência Artificial

4.1.0.1 Introdução

A busca por máquinas capazes de calcular remonta a Idade Antiga e já naquela época provavelmente os gregos conseguiram fazer uma calculadora astronômica, atualmente chamada Máquina de Antikythera (FREETH et al., 2006).

Pulando alguns séculos e indo para 1672 temos a máquina de calcular de Leibniz , que ele chamou de *stepped reckone* (NILSSON, 2009) que já era capaz de fazer cálculos simples.

No século XX tivemos grandes avanços na área da computação e da Inteligência Artificial:

- **Década de 20-40** Alan Turing e Godel que desenvolvem as bases da computação moderna (BRANDãO, 2018).
- **Décadas de 40 e 50** Pesquisas centradas no desenvolvimento de neurônios artificiais, que possibilitaria o desenvolvimento de máquinas de aprender. Durante o verão de 1956 um grupo de cientista entre os quais: Marvin Minsky, John McCarthy, Arthur Samuel e vários outros organizaram um *workshop* para discutir sobre mecanismos autômatos entre outras coisas, nessa reunião John McCarthy é provável que o termo "Inteligência Artificial"tenha sido utilizado pela primeira vez(NILSSON, 2009).
- **Décadas de 50 e 70** Nessas décadas houve grandes esforços para emular o raciocínio humano, planejar tarefas, reconhecimento de linguagem natural, aprendizado por analogia, aprendizado de jogos.
- **Décadas de 70 e 80** Nesta época os cientistas começaram a ter problemas relacionados a armazenamento de dados e tempo de processamento. Com a Teoria da Complexidade Computacional (COOK, 1971) provou que a solução de problemas não dependia somente de grande poder computacional. Nessa época as pesquisas acerca da Inteligência Artificial tiveram um grande declínio.
- **Décadas de 80 e 10** Nessa época houve uma grande concentração em pesquisas relacionadas para aplicações específicas como robótica.

- **Décadas de 00 e 20** Com o advento da Internet e a grande disponibilização de dados para treinamento, a Inteligência Artificial voltou se para a abordagem estatística.

4.1.0.2 Ramos de Pesquisa

Temos atualmente 4 principais ramos de pesquisa:

- Conexionista

  Baseia se no modelo do cérebro humano, a abordagem é feita através da tentativa de se reproduzir o cérebro humano modelando o em um computador. Redes Neurais são um exemplo deste ramo de pesquisa.

- Simbólico

  Tem como base que dado um conjunto de estruturas simbólicas e um conjunto de regras de utilização dessas estruturas e símbolos, já tem-se os meios necessários para se criar "inteligência". A principal aplicação deste ramo são os sistemas especialistas, como sistemas de manufatura.

- Evolucionista

  Baseia - se na Teoria Evolucionista de Darwin, onde podemos fazer com que uma população de agentes solucionadores evoluam conforme se adaptem cada vez mais ao cenário do problema, isto é uma população gera uma nova com algumas mutações, os mais aptos sobrevivem e são expostos a novas mutações. São aplicados em algoritmos genéticos ou mutantes.

- Estatístico / Probabilístico / Incerto

  É baseado em modelos estatísticos e probabilísticos, a partir de um treinamento utilizando uma amostra de dados é possível "ensinar"um programa fazer inferências a partir de um determinado conjunto de entradas. Utilizado em Machine Learning, Redes Bayesianas.

4.2 *Machine Learning*

4.2.0.1 Introdução

A introdução da Internet comercial na década de 90 propiciou ao mundo uma nova plataforma para a interação entre os seres humanos e desde então aliada ao barateamento do poder computacional e da capacidade de armazenamento de dados pode se armazenar e processar uma quantidade talvez jamais vista na história da humanidade.

A interação dos seres humanos com dispositivos que estão na Internet tem se tornado notavelmente intensa na última década, informações como impressões digitais, rostos, fotografias, vídeos, mapas, mapas geográficos e uma infinidade de outras informações são coletadas e armazenadas a cada segundo nos servidores da Internet.

Os hábitos das pessoas e seus comportamentos frente a situações também tem sidos inseridos em diversos banco de dados no mundo com o intuito de analisar e prever o comportamento dos mesmos em uma próxima visita a um sítio de comércio eletrônico por exemplo.

Todas essas informações se tornaram valiosas pois elas podem prever comportamentos futuros de consumidores como qual é o melhor remédio para se prescrever para um determinado paciente com pressão alta dadas as condições dele como idade e comorbidades.

4.2.1 Dados e Aprendizagem

Segundo (MONARD; BARANAUSKAS, 2003):

> "*Machine Learning* é uma área de IA cujo objetivo é o desenvolvimento de técnicas computacionais sobre o aprendizado bem como a construção de sistemas capazes de adquirir conhecimento de forma automática. Um sistema de aprendizado é um programa de computador que toma decisões baseado em experiências acumuladas através da solução bem sucedida de problemas anteriores."

A indução é uma forma de inferência lógica que permite fazer conclusões genéricas a partir de um conjunto particular de exemplos, isto é a partir do aprendizado de algo em particular se generaliza ele para o todo.

Segundo (TALEB, 2015), em seu livro "A lógica do Cisne Negro"ele fornece um exemplo de indução ao afirmar que os europeus deduziram que todos os cisnes eram brancos por nunca antes terem observador um cisne de outra cor.

Outro exemplo de indução é o cálculo do preço de um seguro de carro, nas décadas passadas o cálculo era realizado baseado na probabilidade de ocorrer um sinistro com o carro a ser segurado, com base em algumas variáveis como:

- Local onde o proprietário mora
- Local onde trabalha
- Kilometragem que ele roda diariamente
- Usa carro para trabalhar

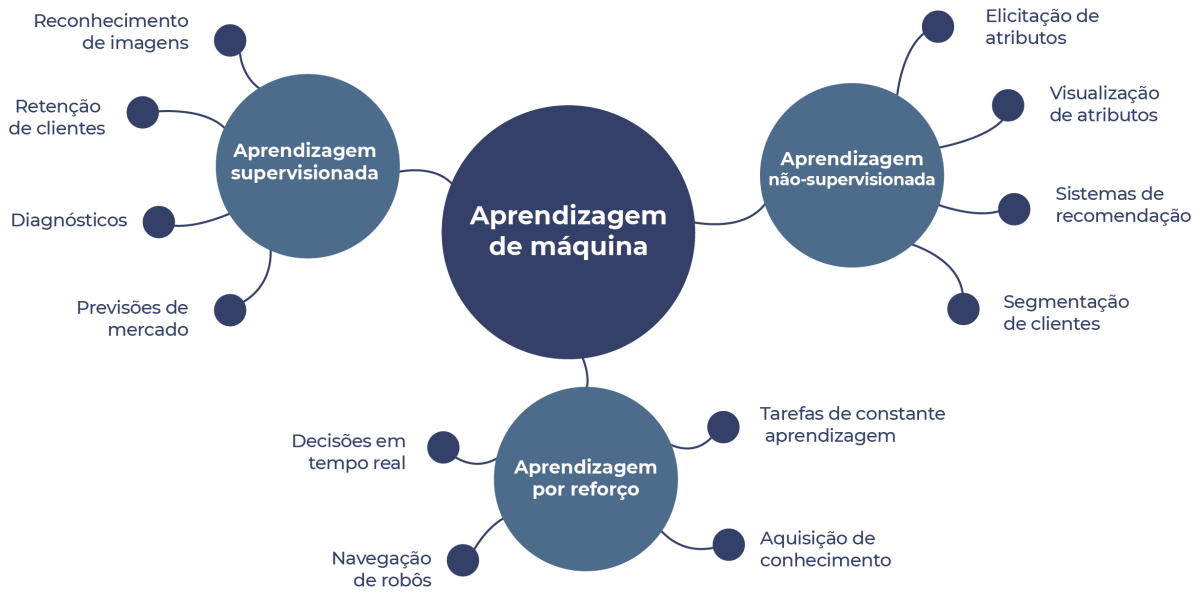


Figura 5 – Classificação dos tipos de Aprendizado em ML

- Idade do carro
- Condições do carro

Com bases nesses parâmetros era feito um cálculo estatístico que calculava a probabilidade de ocorrer um sinistro, e assim o valor do seguro era calculado.

Mesmo Newton deduziu sua famosa equação da gravidade a partir da observação da interação entre os diversos astros que compõem o sistema solar, isso pode ser visto como um exemplo de indução lógica, a partir da observação de alguns astros ele deduziu a Teoria Geral da Gravitação Universal.

Atualmente com a grande quantidade de dados que temos disponíveis, poder computacional e a teoria de *Machine Learning* fazer tais inferências se tornou um processo mais acessível.

O aprendizado de máquina é efetuado a partir de um raciocínio em cima de exemplos fornecidos externamente para o sistema de aprendizado, que pode ser dividido em supervisionado, não-supervisionado. Temos também um terceiro caso em que o aprendizado é feito através da tentativa e erro, o aprendizado por reforço, nessa abordagem, é possível por exemplo, ensinar um sistema a priorizar hábitos em detrimento de outros, com recompensas proporcionais ao acerto.

- **Aprendizado Supervisionado**

No aprendizado supervisionado são fornecidos dados de treinamento previamente rotulados. Geralmente os dados são fornecidos em forma de um *array* com diversos atributos nele e um rótulo associado a ele. O algoritmo de indução então constrói um classificador que tentará determinar a qual rótulo pertencerá uma nova entrada que não esteja rotulada. Para rótulos de classe discreta este problema é conhecido como regressão e para valores contínuos como regressão. Esses tipos de algoritmos são muito utilizados no reconhecimento de imagens, em estratégias para retenção de clientes, diagnósticos ou até mesmo em previsões de mercado.

- **Aprendizado Não Supervisionado**

  No aprendizado não-supervisionado o indutor tenta classificar e agrupar os grupos de dados em *clusters*, caso consiga e necessário uma análise externa para que se possa fornecer insumos do que aquelas classificações possam significar. Esse tipos de algoritmos são muito utilizados para visualização de dados, na recomendação de itens, segmentação de clientes ou até mesmo na elicitação de atributos em problemas complexos.

- **Aprendizado Por Reforço**

  No aprendizado por reforço não existe um treinamento prévio, o treinamento ocorre durante a execução do programa, os dados são geralmente criados em tempo real e reforçam o aprendizado da máquina constantemente. Exemplos típicos de aplicação são robôs que aprendem a andar e se locomover em terrenos não planos, cada queda ou batida que o robô sofre serve como uma entrada de reforço para ele aprender a agir diferentemente da próxima vez, por outro lado toda vez que ele supera um obstáculo isso serve como entrada de como agir certo da próxima vez.

4.2.1.1 Abordagens em *Machine Learning*

Apesar de ser relativamente nova comparada a outras ciências como física e biologia, o campo de estudo de Machine Learning e Inteligência Artificial já possui um grande campo de pesquisa e abordagens, iremos abordar algumas das principais abaixo:

4.3 Classificação por Indução

A classificação por indução de hipóteses consiste em inferir a partir de uma base de dados conhecida uma classificação para um novo conjunto de dados.

Um exemplo é determinar a partir de um conjunto de características apresentadas por um paciente se ele tem determinada doença ou não, esse conjunto de características podem ser idade, peso, diabético ou não, hipertenso e/ou qualquer outra característica que possa influenciar e que seja relevante. Esse conjunto é chamado de vetor de atribuição.

E as características são comumente chamadas de rótulos e o estado do paciente de saída, neste caso saudável ou não.

Figura 6 – Dado rotulado - Fonte Wikipedia

Caracteristica01	Caracteristica02	Caracteristica03	Saída
1254.15	125.58	548.98	Doente
05.15	12.58	5.98	Saudável

Para um bom funcionamento do algoritmo de indução é necessária uma quantidade significativa de dados de treinamento e também que o algoritmo possa trabalhar com dados imperfeitos, mal formados e ausentes. O treinamento deve ser bom o suficiente para que a hipótese tenha acurácia suficiente para produzir resultados satisfatórios para quem deseja com dados genéricos fora do domínio de treinamento.

Quando uma hipótese apresenta uma baixa capacidade de generalização a razão pode ser que ela esta ajustada a um domínio de entrada muito restrito e dizemos que ela está especializada. No caso contrário pode ocorrer que a hipótese está genérica demais e apresenta uma baixa taxa de acerto mesmo no domínio de dados de treinamento (ALPAYDIN, 2017).

4.4 Regressão Linear e Polinomial

A regressão polinomial é um método de se determinar através de uma função polinomial um valor de saída y para uma data entrada x através de uma função polinomial *f(x)*.

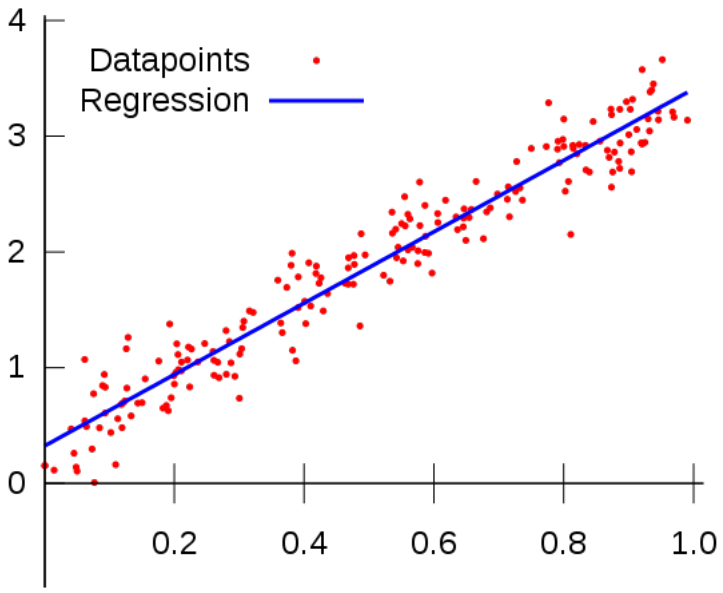
$$y(x) = f(x); \tag{4.1}$$

Onde f(x) é uma equação polinomial do tipo:

$$k + ax + bx^2 + cx\hat{}3 + dx^4 + ... = 0 \tag{4.2}$$

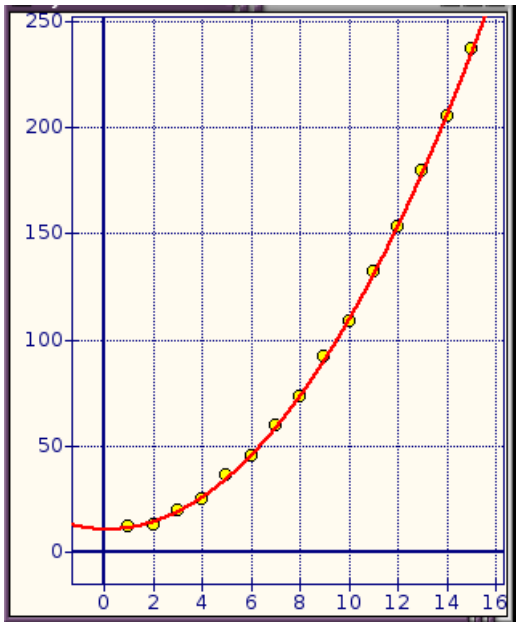
A partir de dados discretos de um determinado conjunto de dados de entrada podemos tentar estabelecer uma equivalência com uma função polinomial que melhor se ajusta aos valores discretos de entrada e com a maior equivalência possível, tal inferência é chamada regressão linear, quando temos uma regressão de utilizando uma função polinomial do primeiro grau, quadrática quando é do segundo grau e assim sucessivamente.

Figura 7 – Regressão Linear



Fonte: Wikipedia

Figura 8 – Regressão Quadrática



Fonte: Wikipedia

Acima temos dois exemplos gráficos de polinômios representando a regressão do

conjunto de dados de entrada, representados pelos pontos; e a regressão representada pela linha contínua.

Para se determinar qual é o melhor polinômio para se ajustar ao conjunto de dados de entrada e necessário determinar o grau e as constantes do polinômio e geralmente é realizado através do Método dos Mínimos Quadrados, mas também podem ser utilizados a máxima verosimilhança, Regularização de Tikhonov e o desvio absoluto (KUBAT, 2015).

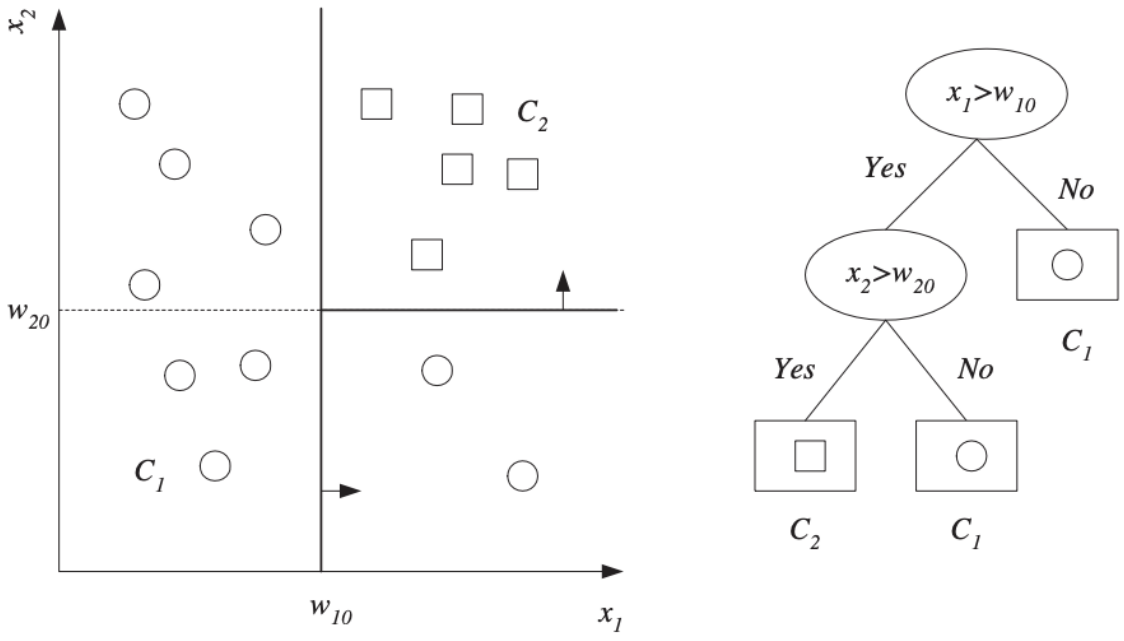
Uma vez realizada a regressão, obtemos o polinômio que servirá de inferência para as estimativas de novos valores de saída para a *Machine Learning*.

4.5 Árvores de Decisão

Outra abordagem muito utilizada para implementação de *Machine Learning* é a árvore de decisão. A árvore de decisão é um estrutura de dados hierárquica que implementa a estratégia do dividir e conquistar. Ela é um método não paramétrico muito eficiente que pode ser usado tanto para classificação quanto para regressão.

A árvore de decisão é composta de nós de decisão onde e folhas. Cada nó de decisão m implementa uma função de teste fm(x) com um valor de saída discreto que se torna a entrada para o próximo nó de decisão e assim sucessivamente (ALPAYDIN, 2017).

Figura 9 – Exemplo de conjunto de dados e a respectiva árvore de decisão



Fonte: (ALPAYDIN, 2017)

Cada função fm(x) implementa um domínio das variáveis de entrada, o nó raiz geralmente divide o domínio de entrada em duas partes similares do ponto de vista do

domínio de entrada, os próximos nós dividem novamente o domínio de entrada em duas partes similares e assim sucessivamente.

Com esta estratégia é possível mapear os valores de entrada e descobrir o(s) valor(es) de saída com n iterações.

Os problemas que são solucionados com o uso de *Machine Learning* raramente possuem uma única variável de entrada, por exemplo vamos analisar o caso em que é necessário administrar uma determinada droga para pacientes cujo valores de entrada são:

- idade (número natural),
- sexo (M,F),
- pressão arterial(alta,normal,baixa),
- colesterol (alto,normal,baixo)

e as possíveis saídas (decisão) são:

- droga A,
- droga B,
- droga C,

Para montar a árvore de decisão é necessário descobrir primeiramente qual será o nó que tem melhor representatividade como nó raiz, fazemos isso através do uso do cálculo de entropia é calculado a entropia de cada variável. A Entropia é uma medida de aleatoridade e incerteza, quanto maior a entropia maior a desorganização e quanto menor , menor a desorganização (CORTIZ, 2020).

$$Entropia = \sum_{i} -Pilog_2Pi$$

Fórmula de entropia

(4.3)

A estratégia para iniciar a montagem da árvore é feita através do cálculo do ganho de informação, que utiliza a equação abaixo para cada variável de entrada no domínio de entrada:

$$ganho = Entropia(pai) - \sum Pesos(filhos) * Entropia(filhos) \tag{4.4}$$

$$Peso(filho) = noamostrafilho/noamostrapai \tag{4.5}$$

Para cada atributo se realiza o cálculo do ganho de informação, o que tiver maior ganho de informação será escolhido como nó raiz, depois e feita uma nova iteração com os outros atributos de entrada e novamente é escolhido o que tem maior ganho de informação, no entanto as iterações não necessariamente param quando os atributos terminam, uma árvore alta pode correr o risco de *overfitting* ou especialização como citado anteriormente, o critério de número de iterações é feito através de ajustes.

4.6 Redes Neurais Artificiais

Na década de 40 e 50 os cientistas da computação iniciaram as tentativas de criar um neurônio artificial. Em 1943 Mcculloch e Pitts apresentaram o artigo onde citavam o Neuron e em 1958 Rosenblatt apresentou o Perceptron (CORTIZ, 2020), utilizado até hoje nas redes neurais.

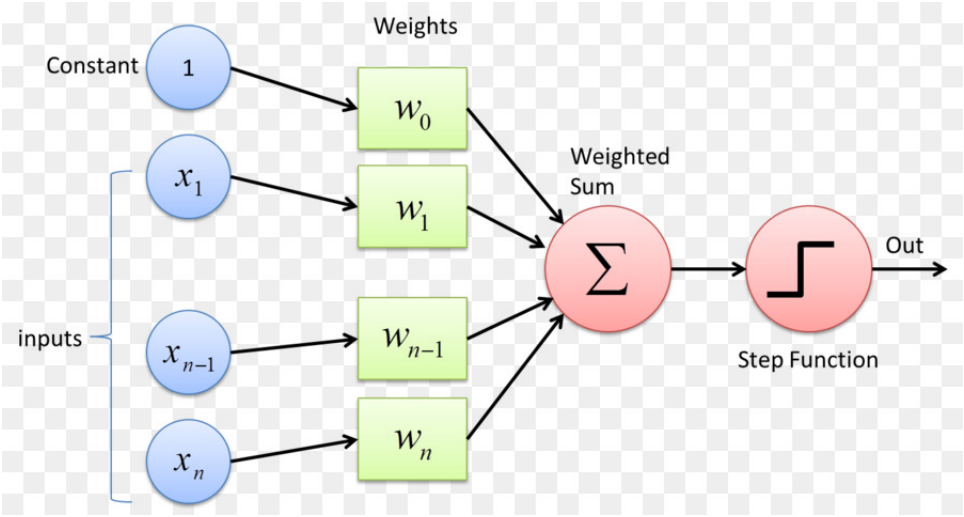


Figura 10 – Representação de Perceptron

Fonte: Wikipedia

O perceptron recebe N entradas x aos quais são atribuídos pesos W, cada entrada x e multiplicada pelo peso w, obtendo o valor y. É feito a somatória de todos os y e ele é passado para a função de ativação *F(x)* que geralmente retorna valores discretos de -1,0 ou +1 (CORTIZ, 2020).

O processo de se determinar os valores corretos para os pesos w é chamado de regra de aprendizado e o processo envolve iniciar a matriz de pesos com valores aleatórios entre -1 e +1. Depois que a rede aprende, esses valores são alterados até que se tenha decidido que a rede tenha sido resolvida. Neste caso a rede é resolvida quando os valores de saídas de um conjunto de dados de treinamento sejam correspondentes ao valores dos atributos de entrada desse conjunto. Durante o aprendizado o algoritmo da rede neural tenta minificar o erro através de técnicas de minimização baseadas na técnica de Gradiente descendente (NORIEGA, 2005).

Uma das limitações da rede perceptron de uma camada é que ela funcionam somente com conjunto de dados de entrada que sejam linearmente separável. Minsky e Papert analisaram mateticamente o Perceptron e demostraram que redes de uma camada não são capazes de solucionar este tipo de problema. Na época eles acreditavam que não seria possível construir um método de treinamento para redes com mais de uma camada. No

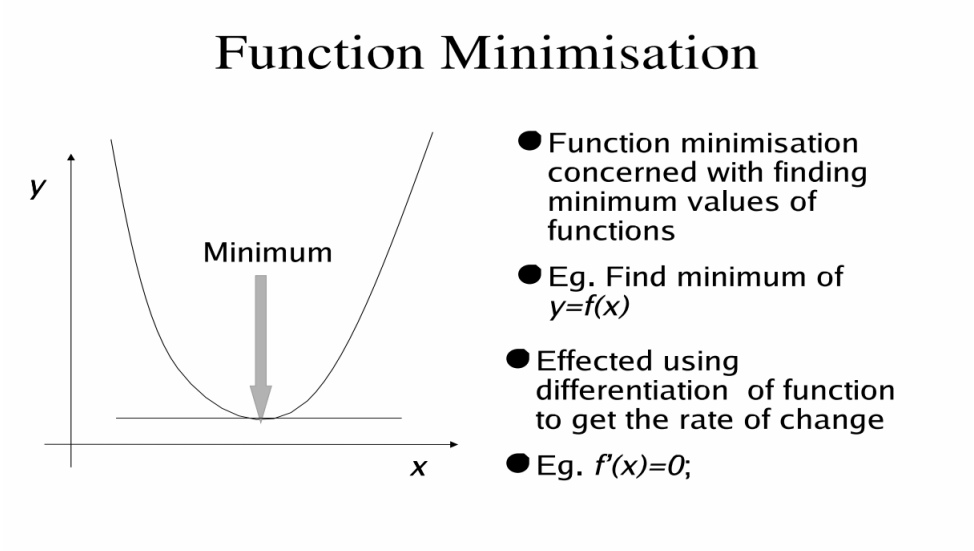


Figure 2: Function Minimisation using Differentiation

Figura 11 – Função de Minimização

Fonte: (NORIEGA, 2005)

entanto em 1986 foi desenvolvido um algoritmo de treinamento de Backpropagation por Rumelhart, Williams e Hinton, nascia assim as redes Perceptron Multi Camadas.

Na rede multi camadas, cada camada realiza uma função específica. A camada intermediária recebe os estímulos da camada anterior e envia para a próxima camada e assim sucessivamente. Este tipo de rede é amplamente utilizado hoje em aplicações que vão desde reconhecimento facial, agentes de atendimento online e detecção de doenças e erros médicos.

O *Deep Learning* é implementado com o uso de várias camadas de rede para reconhecimento supervisionado.

Embora todas as arquiteturas apresentadas seja redes neurais artificias, nem todas são de *Deep Learning*. A característica de modelos de *Deep Learning* são redes neurais artificias com muitas camadas intermediárias.

Como podemos ver existem diversas arquiteturas para implantação de *Deep Learning*, temos diversas arquiteturas, usadas para resolver diferentes tipos de problemas, como por exemplo as arquiteturas de redes neurais convolucionais usadas em problemas de Visão Computacional e as redes neurais recorrentes usadas em problemas de Processamento de Linguagem Natural.

Uma das críticas que as redes neurais multicamadas recebem é que elas são utilizadas em aplicações sensíveis como detecção de doenças e carros autônomos onde vidas são colocadas em risco e elas são ditas caixas pretas, pois o *modus operandi* da mesma não é

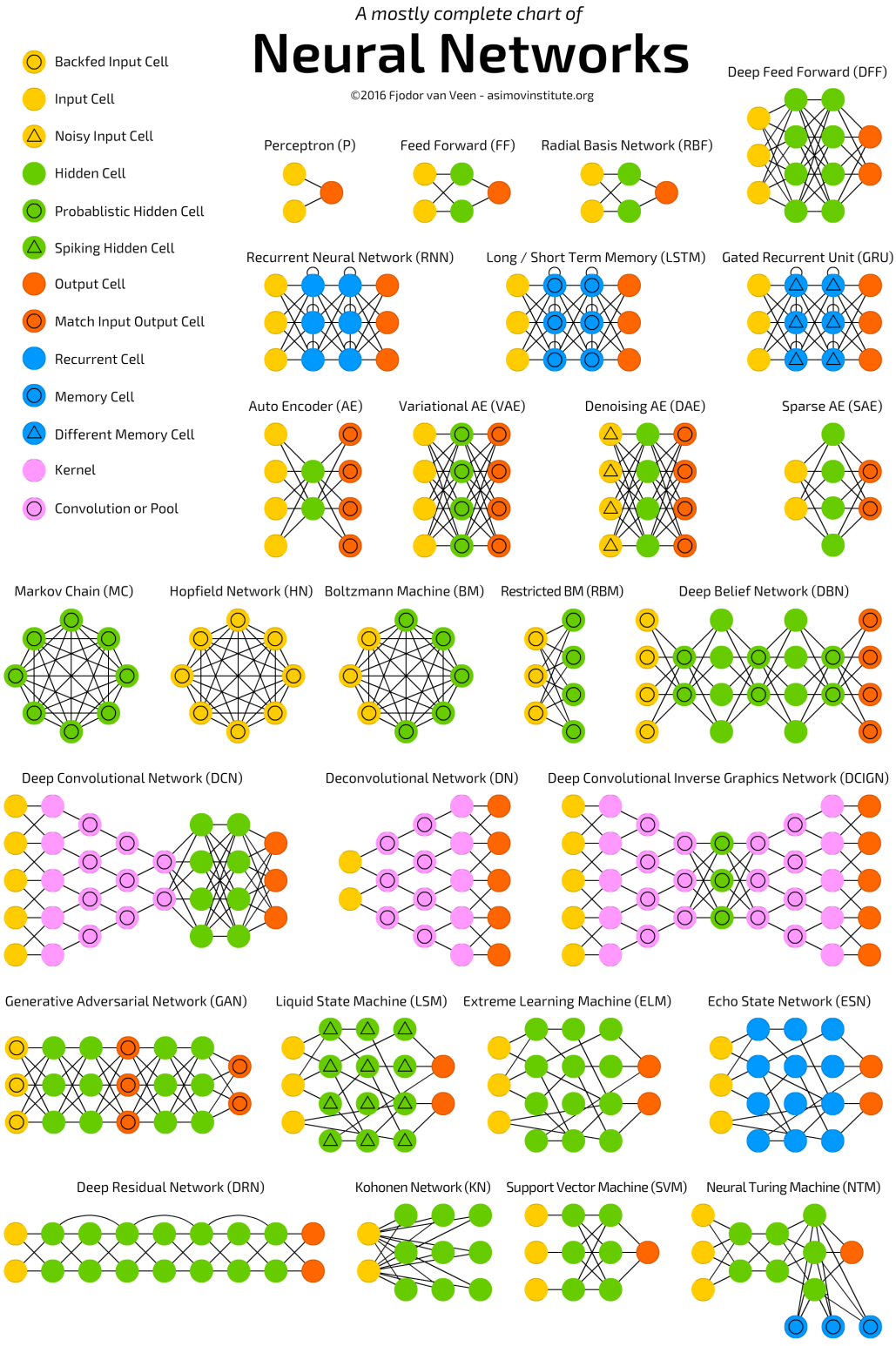


Figura 12 – Várias arquiteturas de implementação de Deep Learning

muito clara.

4.6.1 Autoencoders

O *Autoencoder* é um tipo especial de rede neural que não requer a existência de uma váriavel alvo ou uso de dados rotulados, caracterizando a como uma rede neural de aprendizado não supervisionado.

A principal ideia do Autoencoder é ela copiar a entrada para a saída com uma compactação da informação na sua camada intermediária, obviamente existem algumas restrições colocadas nas suas camadas intermediárias que visam reduzir a presença de ruídos nas imagens, extraindo somente a informação essencial de uma imagem por exemplo.

Elas compactam a entrada em uma representação do espaço latente e em seguida reconstrói na saída um modelo da entrada baseada no conhecimento adquirido. O principal objetivo é conseguir reconstruir a entrada com a menor quantidade de informação necessária nos seus pesos (GOODFELLOW; BENGIO; COURVILLE, 2016).

Elas são compostas de duas partes:

- Codificador (Encoder): é a parte da rede que compacta a entrada em uma representação de espaço latente (codificando a entrada). Pode ser representado por uma função de codificação h = f (x).
- Decodificador (Decoder): Esta parte tem como objetivo reconstruir a entrada da representação do espaço latente. Pode ser representado por uma função de decodificação r = g (h).

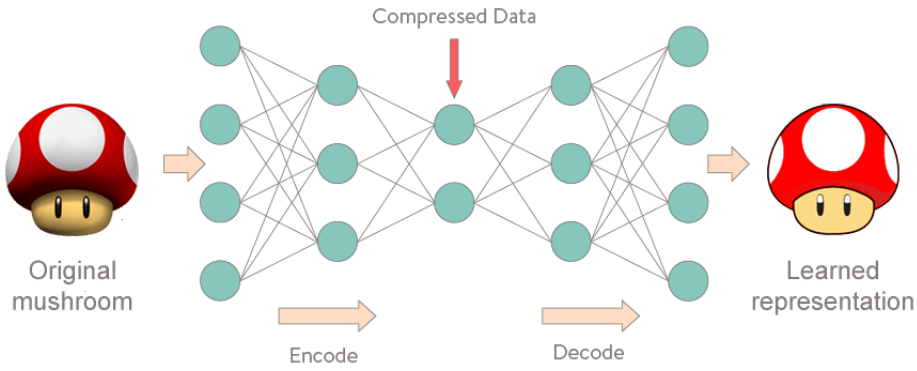


Figura 13 – Autoencoder

4.6.2 Convolutional Neural Network (CNN)

Também chamada de Convolutional Autoencoder é um Autoencoder amplamente utilizada para reconhecimento de imagens devido a suas características.

Ela é divida em três camadas:

- A Camada de Convolução

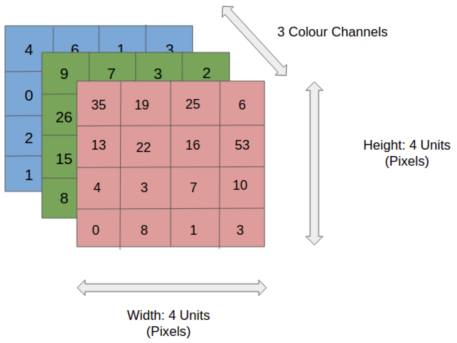


Figura 14 – Imagem de entrada

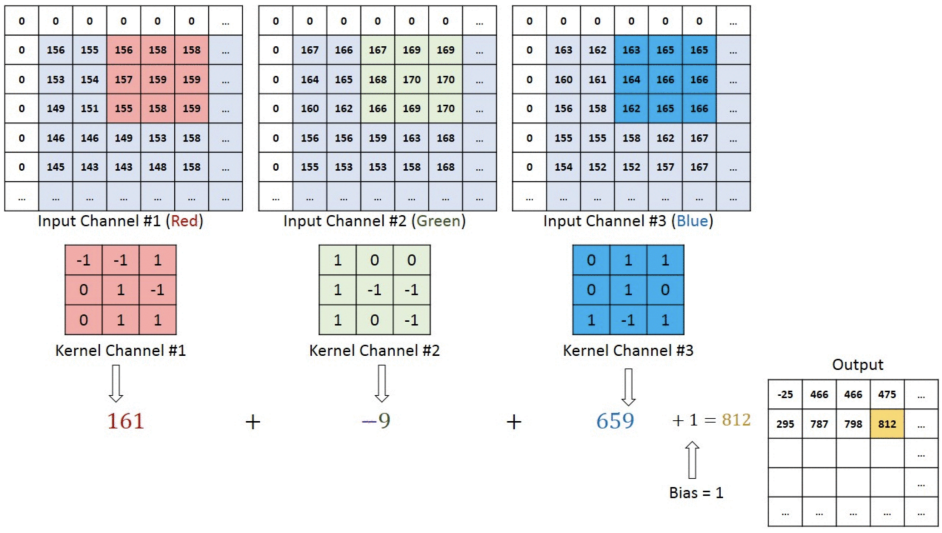


Figura 15 – CNN e Filtros

Nesta camada é realizado um mapeamento da imagem de entrada, usualmente temos um neurônio para cada grupo de *pixels* da imagem e usualmente utilizamos as cores vermelho, verde e azul do sistema RGB como entradas do neurônio. Cada grupo mapeado é transformado em uma nova matriz que é computada através de uma função de ativação e produz uma pontuação para esse grupo mapeado e que é colocado em uma nova matriz de saída com as pontuações obtidas. Esta fase é chamada de filtro.

- O passo de ReLUs

  A Rectified Linear Unit (ReLUs) é a camada da rede que realiza uma retificação dos valores negativos para 0, ela atua como uma função retificadora em Engenharia Elétrica.

- A Camada de Max Pooling

  A camada de *pooling* reduz o nível de informação novamente para evita o excesso de especialização da rede, ou seja ela torna o reconhecimento mais genérico do que estava na camada anterior. Dentro de um grupo é novamente escolhido o com a maior pontuação que é colocado na saída.

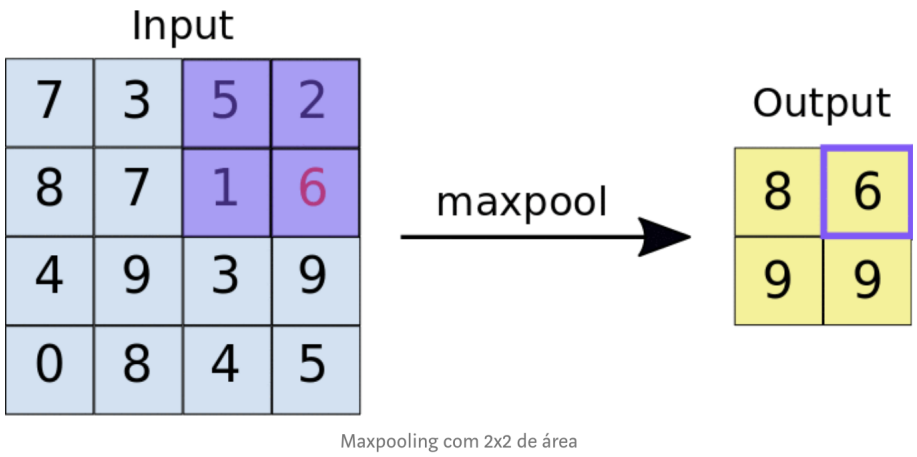


Figura 16 – Pooling

5 Modelo de Mecanismo de Detecção de anomalias

5.1 Anomalias

A Detecção de Anomalias é a identificação de eventos ou produtos que tenham características que são raras de serem observadas e geralmente são diferentes da maioria dos dados/produtos observados. Na produção de um parafuso seria por exemplo a ausência da rosca, em operações bancárias são as fraudes bancárias, em software geralmente as anomalias estão relacionadas a um estado raramente observado durante os testes e operação do software e que possivelmente indique uma falha.

Existem três abordagens na detecção de anomalias:

- **Supervisionadas** onde é necessário que os dados estejam previamente rotulados como normal e anormal para que se possa criar um conjunto de dados de treinamento para a Machine Learning
- **Não supervisionada**, aqui supostamente temos um conjunto de dados no qual a maioria é apresentada como tendo um comportamento normal, nesta técnica procura-se dados que sejam discrepantes do resto dos dados.
- **Semi supervisionado**, nesta técnica são construídos modelos representando o comportamento normal de um determinado conjunto de dados de treinamento normal e em seguida é testado a probabilidade de uma instância de teste ser gerada pelo modelo aprendido.

5.2 O Problema e dados de atribuição

Vamos tratar particularmente de 2 tipos de problemas, o primeiro é verificar se uma determinada função de software *f(x1,x2,x3,x4...xn)* está computando os valores de entrada como esperado, gerando saídas s1,s2,s3...sn esperadas. O segundo problema é se uma página de um website está sendo renderizada conforme o esperado.

Como exemplo de dados de atribuição do modelo proposto vamos usar um grupo de vetores de entrada e saída de uma determinado software como o exemplo apresentado na Tabela 2, a priori os critérios e valores de saída podem ser quaisquer entre números de ponto flutuante ou *strings*.

Tabela 2 – Exemplo de Dados de atribuição

Critério	Pontos
Pagamentos em dia	10
Salário	8
Dívidas em aberto 30dias	7
Dívidas nos últimos 120 dias	4
Valor dívidas	4
Crédito últimos 60 dias	3
Score	34

Podemos a partir da execução em tempo real do software coletar os dados de entrada e saída em um arquivo de log do tipo texto e tabulá-los.

Outro tipo de dado de atribuição que podemos coletar são imagens de um *snapshot* da tela de um *website* várias vezes ao dia, digamos a cada segundo, serão 86400 entradas durante o dia. Sabe-se que por exemplo que o *website* de um site de jornalismo como o G1 (https://g1.com.br) muda algumas vezes durante o dia, no entanto a distribuição dos textos imagens, títulos e demais componentes mantém um certo padrão durante o dia.

Para solucionar o primeiro problema podemos utilizar uma Machine Learning com aprendizado supervisionado, gerando a partir do log rótulos de normal e anormal e tratá-los com um técnicas como regressão polinomial ou árvore de decisão ou clusterização de dados, no entanto é deveras custoso rotular a saída de uma grande quantidade de atributos, o que se pode fazer neste caso é primeiramente utilizar uma ferramenta de teste e assumir que os dados desta primeira iteração são considerados normais e a seguir gera-se se diversos tipos de saídas erradas propositais (abordado no capítulo 2) e a rotula-se este conjunto de dados como anômalo e o utilizamos para o aprendizado da Machine Learning.

Para o segundo problema podemos utilizar uma a *Convolutional Neural Network (CNN)* para fazer o reconhecimento das páginas.

Através da automação de um capturador de tela podemos alimentar a CNN com uma grande quantidade de entradas para que ela comece a identificar páginas que estão

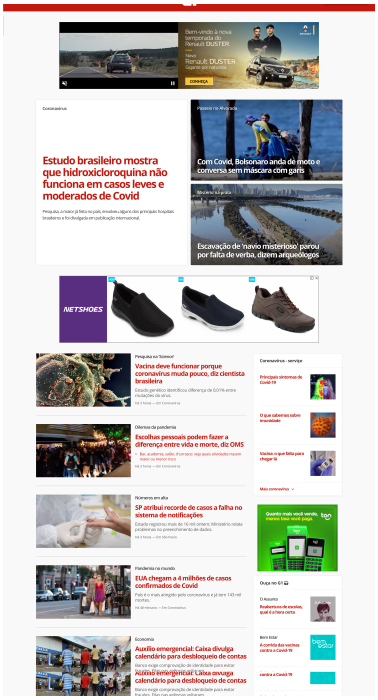


Figura 17 – Sítio do G1

no padrão normal e aquelas que fogem ao padrão.

5.3 Ferramentas e Tecnologias sugeridas

Atualmente dispomos de uma grande quantidade de ferramentas para implementar ambas as técnicas acima como:

- Keras - https://keras.io/
- PyTorch - https://pytorch.org/
- TensorFlow - https://www.tensorflow.org/
- GoFullPage - https://gofullpage.com/

Fica como sugestão a implantação da arquitetura descrita acima com algumas das ferramentas sugeridas.

6 Conclusão

A Inteligência Artificial tem trazido diversos avanços e benefícios na mais diversas áreas, como medicina diagnóstica, carros autônomos, classificação de imagens, reconhecimento facial, segurança e reconhecimento de linguagem natural. No entanto o percentual de acertos nessas áreas raramente chega a 100 por cento mas chega perto em alguns casos. A eficácia pode ser comprovadamente não de 100 por cento, mas isto não impede que ela seja uma grande ferramenta que contribua significativamente no avanço dessas diversas áreas.

Na área de qualidade de software também não é diferente, podemos concluir que podemos utilizar *Machine Learning* como um grande auxiliar na parte de melhoria de qualidade de software, podendo inclusive substituir algumas das atuais ferramentas de automação de testes de software.

Há ainda grandes aplicações da Inteligência Artificial no campo da Engenharia de Software que podem ser estudadas, como reconhecimento e classificação de código, testes de mutantes com IA, este é um assunto que pode ser explorado no futuro para os leitores desta dissertação.

Referências

ABRAHAM, K.; HORST, B. *Artificial intelligence methods in software testing*. Singapore: World Scientific, 2004. v. 56. 19

ALPAYDIN, E. *Introduction to machine learning*. [S.l.]: MIT press, 2017. 21, 44, 46

ATHOW, D. *Pentium FDIV: The processor bug that shook the world*. 2014. Disponível em: <https://www.techradar.com/news/computing-components/processors/pentium-fdiv-the-processor-bug-that-shook-the-world-1270773>. 22

BASS, L.; WEBER, I.; ZHU, L. *DevOps: A software architect's perspective*. [S.l.]: Addison-Wesley Professional, 2015. 36

BOURQUE, P.; FAIRLEY, R. E. et al. *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0*. New York: IEEE Computer Society Press, 2014. 19, 24

BRANDãO, P. Alan turing: da necessidade do cálculo, a máquina de turing até à computação. 03 2018. 39

BROOKS, F. P. No silver bullet. *IEEE Computer*, New York, v. 20, n. 4, p. 10–19, 1987. 22, 28

COHANE, R. *Financial Cost of Software Bugs*. 2017. Disponível em: <https://medium.com/@ryancohane/financial-cost-of-software-bugs-51b4d193f107>. Acesso em: 23.05.2018. 22

COOK, S. A. The complexity of theorem-proving procedures. In: *Proceedings of the third annual ACM symposium on Theory of computing*. [S.l.: s.n.], 1971. p. 151–158. 39

CORTIZ, D. *Curso de Inteligência Artificial*. 2020. Disponível em: <https://www.youtube.com/channel/UC5MXrSUoLW0JRd2j7q1ef7Q>. 47, 49

DELAMARO, M.; JINO, M.; MALDONADO, J. *Introdução ao teste de software*. Rio de Janeiro: Elsevier Brasil, 2017. 18, 19, 21, 25, 26, 27, 28, 29, 32, 33, 34

DIJKSTRA, E. W. The humble programmer. *Communications of the ACM*, ACM, v. 15, n. 10, p. 859–866, 1972. 28

FERREIRA, F. A. *Inteligência artificial na verificação e teste de software para desenvolvimento ágil*. Tese (Doutorado) — Instituto Superior de Engenharia de Lisboa, 2017. 19

FREETH, T. et al. Decoding the ancient greek astronomical calculator known as the antikythera mechanism. *Nature*, Nature Publishing Group, v. 444, n. 7119, p. 587–591, 2006. 39

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. *Deep learning*. [S.l.]: MIT press, 2016. 52

HOWDEN, W. E. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, IEEE, n. 3, p. 208–215, 1976. 34

JUNIOR, H. d. S. C.; PRADO, A. F. do; ARAÚJO, M. A. P. Complexity tool: Uma ferramenta para medir complexidade ciclomática de métodos java-complexity tool: a tool for measuring cyclomatic complexity in java methods. *Multiverso: Revista Eletrônica do Campus Juiz de Fora-IF Sudeste MG*, v. 1, n. 1, p. 66–76, 2016. 30

KOSCIANSKI, A.; SOARES, M. dos S. *Qualidade de Software-2ª Edição: Aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software*. São Paulo: Novatec Editora, 2007. 19, 21, 24, 25

KUBAT, M. *Introduction to machine learning*. [S.l.]: Springer, 2015. 21, 46

KUHN, D. R.; WALLACE, D. R.; GALLO, A. M. Software fault interactions and implications for software testing. *IEEE transactions on software engineering*, IEEE, v. 30, n. 6, p. 418–421, 2004. 32

KURZWEIL, R. *The age of spiritual machines: When computers exceed human intelligence*. New York: Penguin, 2000. 18

LEVESON, N. G.; TURNER, C. S. An investigation of the therac-25 accidents. *Computer*, IEEE, v. 26, n. 7, p. 18–41, 1993. 18, 22

LIONS, J.-L. et al. *Ariane 5 flight 501 failure*. [S.l.]: Technical report, European Space Agency, July 1996. Available as http://www. esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep. html, 1996. 18, 22

LIU, C.-H. et al. Structural testing of web applications. In: IEEE. *Proceedings 11th International Symposium on Software Reliability Engineering. ISSRE 2000*. [S.l.], 2000. p. 84–96. 36

LLP, A. R. *Global Artificial Intelligence Market Analysis & Trends - Industry Forecast to 2025*. 2013. Disponível em: <https://www.researchandmarkets.com/research/c8k4lk/global_artificial>. Acesso em: 23.05.2018. 18

MCCABE, T. J. A complexity measure. *IEEE Transactions on software Engineering*, IEEE, n. 4, p. 308–320, 1976. 29

MONARD, M. C.; BARANAUSKAS, J. A. Conceitos sobre aprendizado de máquina. *Sistemas inteligentes-Fundamentos e aplicações*, Manole Ltda, v. 1, n. 1, p. 32, 2003. 41

MURPHY, C.; KAISER, G. E.; ARIAS, M. An approach to software testing of machine learning applications. In: *SEKE*. New York: [s.n.], 2007. v. 167. 19

MYERS, G. J.; SANDLER, C.; BADGETT, T. *The art of software testing*. New Jersey: John Wiley & Sons, 2011. 32

NEUMANN, J. V. First draft of a report on the edvac. University of Pennsylvania Moore School of Electrical Engineering, Pennsylvania, 1945. 17, 27

NGUYEN, B. N. et al. Guitar: an innovative tool for automated testing of gui-driven software. *Automated software engineering*, Springer, v. 21, n. 1, p. 65–105, 2014. 37, 38

NILSSON, N. J. *The quest for artificial intelligence*. [S.l.]: Cambridge University Press, 2009. 39

NORIEGA, L. Multilayer perceptron tutorial. *School of Computing. Staffordshire University*, Citeseer, 2005. 49, 50

PAULK, M. C. et al. The capability maturity model for software. *Software engineering project management*, v. 10, p. 1–26, 1993. 24

PRESSMAN, R.; MAXIM, B. *Engenharia de Software-8ª Edição*. [S.l.]: McGraw Hill Brasil, 2016. 29

PRESSMAN, R. S. *Software engineering: a practitioner's approach*. [S.l.]: Palgrave Macmillan, 2005. 17

RICCA, F.; TONELLA, P. Analysis and testing of web applications. In: IEEE. *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*. [S.l.], 2001. p. 25–34. 36, 37

RUSSELL, S. J.; NORVIG, P. *Artificial intelligence: a modern approach*. New York: Prentice Hall, 1995. 19

STANDARD, I. Standard 610-1990. *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries*, 1990. 22

SUGIMORI, Y. et al. Toyota production system and kanban system materialization of just-in-time and respect-for-human system. *The International Journal of Production Research*, Taylor & Francis, v. 15, n. 6, p. 553–564, 1977. 24, 25

TALEB, N. N. *A lógica do cisne negro: o impacto do altamente improvável*. [S.l.]: Editora Best Seller, 2015. 41

TAPSCOTT, D.; WILLIAMS, A. D. *Wikinomics: How mass collaboration changes everything*. New York: Penguin, 2008. 18

THIBODEAU, P. *Study: Buggy software costs users, vendors nearly $60B annually*. 2012. Disponível em: <https://www.computerworld.com/article/2575560/it-management/study--buggy-software-costs-users--vendors-nearly--60b-annually.html>. Acesso em: 23.05.2018. 18

TURING, A. M. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, Wiley Online Library, v. 2, n. 1, p. 230–265, 1937. 17