João Pedro Martins Pereira

Licenciado em Engenharia Eletrotécnica e Computadores

Métodos de Machine Learning para Eficiência Energética

Dissertação para obtenção do Grau de Mestre em Engenharia Eletrotécnica e Computadores

Orientador: José Barata de Oliveira, Professor Doutor, FCT-UNL

Júri:

Presidente: Prof. Doutora Maria Helena Silva Fino

Arguente(s): Prof. Doutor João Francisco Alves Martins
Vogal(ais): Prof. Doutor José Barata de Oliveira

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Março, 2018

Métodos de Machine Learning para Eficiência Energética

Para a minha mãe, pai, irmão e todos aqueles que levo no coração.

Agradecimentos

Neste capítulo deixo os mais sinceros agradecimentos a todos aqueles que tornaram a realização desta dissertação possível, tanto a nível académico como pessoal. Desde já, obrigado a todos.

Quero agradecer em especial ao Pedro Monteiro, que embora não tenha sido possível colocar o seu nome, foi para mim um incansável Co-orientador que me ajudou não só a estruturar ideias, como também mostrar o caminho para o sucesso da realização desta tese. Um grande obrigado.

Ao meu Orientador, o Professor Doutor José Barata, um grande obrigado por me ter dado a oportunidade de desenvolver a minha dissertação num tema proposto por ele. Foi sem dúvida um tema bastante desafiante e que futuramente poderá causar um impacto enorme na sociedade.

Gostava também de agradecer à ITCONIC, empresa motivadora deste tema, e também à TRANE, por me ter facilitado o *software* de recolha de dados.

Como a tese não deixa de ser a reta final de 5 anos intensos em Engenharia Electrotécnica e Computadores, quero agradecer a todos os meus colegas de faculdade, amigos e professores por me terem proporcionado momentos únicos e inesquecíveis e por terem participado no meu desenvolvimento técnico e pessoal. Uma parte desta tese, é vossa.

E porque em último costuma vir sempre o agradecimento mais especial, um obrigado de tamanho infinito aos meus pais, irmão e avôs. Desde a mudanda de cidade para começar este percurso até à entrega da dissertação, o apoio deles foi incansável, acreditando sempre nas minhas capacidades e dando-me tudo o que tinham e o que não tinham. Um obrigado nunca chegará.

Resumo

Os centros de processamento de dados (*Data Centers*) são locais onde existe uma quantidade significativa de armazenamento de dados e vários equipamentos para os processar. Devido à grande evolução do volume global de dados (*Big* Data) e à tendência de processar quantidades de dados cada vez maiores, a gestão dos *Data Centers* torna-se mais complexa. São o núcleo dos negócios modernos das grandes empresas nos tempos de hoje, onde a utilização da *cloud* aumenta exponencialmente tornando o acesso e armazenamento de dados mais rápido e eficaz, e que combina com um conceito que vai ganhando cada vez mais credibilidade- Internet das coisas (*Internet of Things*).

Com estes fatores que provocam o rápido crescimento do *Big Data*, os equipamentos num *Data Center* consequentemente aumentam a carga da sua utilização. O aumento da sua utilização provoca um maior aquecimento nesses equipamentos, que gera aumentos de temperatura nos *Data Centers*.

Um dos maiores custos para a organização que controla um *Data Center* é a energia, principalmente a energia utilizada na refrigeração. Devido às evoluções apresentadas, a energia utilizada para a refrigeração aumenta drasticamente. No entanto, este aumento não é controlado, tornando o uso da energia bastante ineficiente, constituindo um custo excessivo para as organizações possuidoras de *Data Centers*.

Neste projeto são apresentadas soluções para combater o uso ineficiente da energia nos *chillers*, equipamentos que realizam a refrigeração dos *Data Centers*. Estas soluções incluem implementações de algoritmos *machine learning* que oferecem inteligência artificial a estes equipamentos de refrigeração, com o intuito de tornar o seu funcionamento mais adequado para

a situação apresentada. Este tipo de algoritmos tem a possibilidade de aprender continuamente, sendo possível prever situações futuras e descobrir padrões interessantes de modo a que sejam tratados da forma mais adequada.

Foram realizados vários testes para validar os algoritmos implementados, sendo o objetivo principal, na prática, o aumento da eficiência energética por parte dos *chillers* e consequentemente a diminuição dos custos para os proprietários dos *Data Centers*.

Palavras-Chave: *Data Center, Big Data, Eficiência Energética, Chillers, Machine Learning*

Abstract

Data Centers (DC) are places where there are a significant amount of data storage and several equipment to process those data. Due to the great evolution of Big Data and the tendency to process larger amounts of data, the management of Data Centers has become increasingly complex. DCs are the core of the modern business for today's enterprises, where the use of cloud increases exponentially, making data access and storage faster and more effective, which combines with the increasingly credible concept of Internet of Things.

With this rapid growth of Big Data, the use of the equipment in a Data Center consequently increase. This increase of usage causes a greater heating in these equipment, which generates increases of temperature in the DCs.

One of the biggest costs for the organization that controls a Data Center is the energy, especially the energy used in refrigeration. Due to those evolutions previously explained, the energy used for cooling the Data Center increases drastically, However, this increase is not controlled, turning the use of energy inefficient, being an excessive cost for the organizations that have Data Centers.

In this project, it is presented solutions to combat the inefficient use of energy in the chillers, equipment that perform the cooling of Data Centers. These solutions are about implementations of machine learning algorithms to provide artificial intelligence to this refrigeration equipment, in order to make its operation more suitable for the situation explained. This type of algorithms has the possibility to learn continuously, being possible to predict future situations and to discover interesting patterns so that they can be treated in the most appropriated way.

Several tests were carried out to validate the algorithms implemented, with the main objective to increase the energy efficiency of the chillers and consequently reduce costs for Data Center owners.

Keywords: *Data Center, Big Data, Energy Efficiency, Chillers, Machine Learning*

Índice

Índice de Figuras

Índice de Tabelas

Acrónimos

API	Application Programming Interface
CC	Cloud Computing
DC	Data Center
DAG	Directed Acyclic Graph
EWT	Entering Water Temperature
IT	Information Technology
IoT	Internet of Things
LWT	Leaving Water Temperature
ML	Machine Learning
NoSQL	Not-only SQL
PUE	Power Usage Effectiveness
SaaS	Software as a Service
SQL	Structured Query Language
KNN	K-Nearest Neighbours

1

Introdução

Os seres humanos sempre trabalharam com o sentido de se tornarem mais eficientes nas suas atividades, criando métodos para um maior conforto e eficácia. Numa altura em que a sociedade se torna cada vez mais desenvolvida, com a tecnologia em constante evolução, as necessidades de conforto aumentam, o que leva a um aumento do consumo de energia. No entanto, é possível fazer uma utilização responsável e cuidada no sentido de diminuir o consumo – isto é chamado de eficiência energética. Por outras palavras, eficiência energética é a otimização do consumo de energia, ou seja, a diminuição do desperdício energético.

Os *Data Centers* (DC) são conhecidos como locais onde existe uma quantidade significativa de armazenamento de dados e equipamento para os processar. Antes das últimas grandes evoluções da tecnologia, a gestão de um DC era considerada um processo simples. No entanto, devido à tendência inexorável de processar cada vez mais dados, a gestão dessas instalações cresceu em complexidade. Complicando ainda mais a situação, as decisões operacionais no DC incluem fatores como potência térmica, refrigeração e espaço físico para as máquinas. Tornaram-se o núcleo dos ambientes de negócios modernos, desde que a utilização da *cloud* começou a aumentar exponencialmente, tornando o acesso e armazenamento de dados mais rápido e eficaz. A mudança

de computação "*consumer-side*" para sistemas *cloud*, com grandes empresas armazenando dados em *cloud*, combinando com a rápida adoção da *Internet*, aumentaram a utilização de DCs. Este aumento gera o que é considerado o problema central para os DCs – o consumo de energia. São os usuários com o maior crescimento no uso de energia, consumindo biliões de kWh de energia, que resultam em custos financeiros na ordem dos biliões, valores que tendem a aumentar nos próximos anos.

A energia é então o maior custo para a organização que controla um DC, grande parte devido ao arrefecimento. Este arrefecimento torna-se importante pois aumenta a eficiência das máquinas (servidores neste caso) no DC, máquinas estas que aquecem consoante a sua utilização. Tendo em conta o aumento da utilização dos DCs, é necessária uma monitorização mais complexa e moderna para controlar estes aumentos de temperatura. Técnicas como contenção de ar quente, economização de água e monitorização extensiva são algumas boas maneiras de criar um arrefecimento mais eficiente. Num exemplo prático, uma pequena variação no *set point* da temperatura no corredor das máquinas deve gerar variações na carga do equipamento de arrefecimento (*chillers*, permutadores de calor, bombas de água, etc.), variações que devem funcionar de forma eficiente. O problema geral é que a maioria dos DCs no mundo não estão preparados para estas variações de temperatura, sendo que o equipamento de arrefecimento em muitos dos casos funcionará igualmente tanto para variações pequenas como variações significativas, resultando numa gestão ineficiente ou cara da energia. Alguns DCs modernos já combatem este problema, tentando mudar essa situação utilizando inteligência artificial para as máquinas de arrefecimento.

Posto isto, a ITCONIC, empresa que lidera a gestão e operação de DCs na Península Ibérica propôs que se procurasse uma solução eficaz no sentido de aumentar a eficiência energética num dos seus DC, situado em Lisboa. Isto é, controlar aumentos de temperatura drásticos de uma maneira eficiente, sendo monetariamente menos dispendioso para o usuário e melhor para o ambiente. Para isto, a solução que será desenvolvida tem como base no desempenho dos *chillers*. *Chiller* é uma máquina de arrefecimento usado em DCs que consiste em remover o calor de um elemento, colocando-o noutro elemento. O objetivo é tornar este sistema o mais eficiente e inteligente possível, respondendo a variações de temperatura de um modo adequado a cada situação diferente. Como trabalho futuro, será desenvolvido um *software* de otimização que "aprenda" como os *chillers* trabalham durante um certo período numa variação de temperatura na sala de um DC. Todos os dados retirados sobre os *chillers* serão usados para melhorar a eficiência geral da energia utilizada no DC, e irão servir para determinar os parâmetros mais eficientes a serem executados pelo aparelho.

Para este objetivo, o capítulo 2 apresentará a pesquisa sobre o estado da arte, relacionado com o tema da dissertação. É possível encontrar alguns conceitos importantes relacionados com o porquê e a resolução do problema proposto bem como soluções de outras entidades para problemas semelhantes. Isto torna-se importante para que no futuro se faça uma comparação entre as soluções

existentes e a solução a ser proposta no trabalho futuro. É também dado a conhecer algumas tecnologias interessantes para o tratamento de dados e criação de algoritmos de inteligência artificial, tecnologias estas onde a solução proposta no capítulo 4 se foca. Contém também uma conclusão geral sobre a investigação feita para o estado da arte, dificuldades encontradas e que as que irão aparecer no desenvolvimento do trabalho futuro.

Para que fosse implementada a solução proposta para o problema apresentado foi necessário pensar numa arquitetura eficaz, apresentada no capítulo 3. O problema principal incide na relação entre o *chiller* e o DC, que é explicada neste capitulo com o intuito de se perceber melhor onde a solução apresentada irá atuar. Explicada esta relação, é apresentada a arquitetura, onde é possível verificar as suas complexas relações entre os dados obtidos e os algoritmos implementados, bem como diagramas que ajudam na compreensão.

No capítulo 4 é descrita a implementação dos algoritmos de *machine learning*. Estes algoritmos terão como principal objetivo oferecer inteligência artificial aos *chillers* para que estes atuem da maneira mais eficiente consoante as anomalias apresentadas nas variações de temperatura nas salas dos DCs. São explicadas as tecnologias utilizadas para a obtenção dos algoritmos, os seus funcionamentos e como estes vão ao encontro da arquitetura apresentada.

Já no capítulo 5 encontra-se a comparação do desempenho dos algoritmos de *machine learning.* Foram realizados vários testes de *performance* e verificações das soluções apresentadas pelos diferentes algoritmos, de modo a perceber qual se adequa melhor a este caso.

Por fim, no capítulo 6 são apresentadas as conclusões gerais do estudo realizado e das propostas implementadas para combater o problema. São indicadas as dificuldades encontradas bem como as competências adquiridas, tanto a nível técnico como a nível cultural. É apresentado também o trabalho futuro que tem em vista o seguimento deste, para que seja devidamente testado nos *chillers* e perceber se são viáveis ou não as soluções apresentadas.

2

Estado da Arte

Para propor uma solução para o problema, existem vários pontos essenciais que é preciso focar, identificando os processos chave e as tecnologias que mais se adequam.

No capítulo 2.1 é apresentado o cerne do problema, nomeadamente o crescimento exponencial da quantidade de dados, a evolução da tecnologia e o impacto que tem nos DCs. Nos capítulos 2.2 e 2.3 são discutidas algumas tecnologias existentes, comparadas de modo a compreender melhor quais se adequam mais para o problema apresentado. Estas tecnologias referidas servirão para o tratamento de dados - processar, armazenar e analisar. Será um tópico importante pois a solução proposta será baseada na qualidade dos dados recolhidos e na sua análise. No capítulo 2.3 é apresentada a tecnologia usada para a criação da inteligência artificial dos *chillers* e algumas propostas existentes relacionadas com a resolução do caso de estudo proposto. Por fim, no capítulo 2.4, são apresentadas algumas conclusões gerais sobre a pesquisa realizada e algumas dificuldades relacionadas com a solução do problema.

2.1. *Big Data*

Big Data (BD) é um termo bastante usado nos dias de hoje, que se associa a grandes e complexos conjuntos de dados. Devido ao desenvolvimento da tecnologia, o BD tornou-se um termo que grandes empresas ainda não conseguem controlar de forma perfeita. A computação de BD é um paradigma para a descoberta científica e análise de dados em infraestruturas de grande escala. Os dados recolhidos, ou produzidos, a partir de várias explorações científicas e transações comerciais muitas vezes requerem ferramentas para facilitar a sua gestão, análise, validação, visualização e divulgação de dados eficazes, preservando o valor intrínseco desses mesmos dados.

Foi previsto que poderia haver um crescente aumento no armazenamento de dados digitais em 40% entre 2012 e 2020 (Gantz & Reinsel, 2012), algo que se verificou. O gráfico da Figura 1 mostra este aumento até 2015, com a linha a indicar que irá aumentar nos anos seguintes.

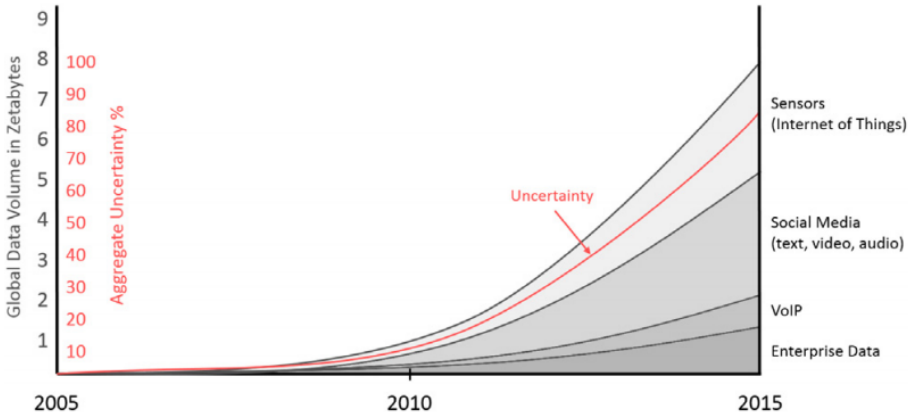


Figura 1 - Crescimento global de dados
(Snášel, Nowaková, Xhafa, & Barolli, 2017)

Os novos avanços nas tecnologias permitem: 1) processamento de dados mais rápido; 2) armazenamento em grande escala; 3) redes mais rápidas e poderossas a preços mais acessíveis permitindo que grandes volumes de dados sejam preservados e utilizados a uma taxa mais rápida. Também os recentes avanços nas tecnologias de *cloud* permitiram preservar cada bit de dados recolhidos, oferecendo alta disponibilidade de armazenamento e processamento (Kune, Konugurthi, Agarwal, Chillarige, & Buyya, 2011).

BD representa dados de volumes grandes e com tendência a crescerem, em formas variadas de dados em frequente mudança de fontes múltiplas e autónomas no tempo mínimo possível. BD é atualmente caracterizada por 5 Vs : volume, velocidade, variedade, veracidade e valor (Özköse, Arı, & Gencer, 2015).

2.1.1. *Internet of Things*

Descrita por alguns como "uma rede de grande escala de dispositivos com uma identidade única, interligados, tendo por base protocolos *standard* de comunicação", a *Internet of Things* (IoT) é uma tecnologia que rapidamente vingou no cenário moderno das telecomunicações sem fios (Xiao & Wang, 2011). Esta tecnologia tem como conceito base a presença de uma variedade de objetos como sensores, atuadores, *smartphones*, com a capacidade de interagir e cooperar uns com os outros de modo a alcançar objetivos comuns. A IoT é um termo que se utiliza para caracterizar uma evolução na tecnologia. Esta evolução tem vindo a crescer gradualmente e inconscientemente nos dias de hoje devido à necessidade da conectividade entre os seres humanos e os dispositivos (Farooq & Waseem, 2015).

Em relação ao caso de estudo proposto, a IoT tem um potencial efeito de transformação na economia dos DCs, nos seus clientes, fornecedores de energia, tecnologias e nos modelos de venda e marketing. Esta evolução na tecnologia gera grandes quantidades de dados que precisam de ser processados e analisados em tempo real. Como consequência, os DCs terão uma maior carga de trabalho, gerando novos desafios de segurança, capacidade e análise (Atzori, Iera, & Morabito, 2016). Como mencionado, IoT conecta dispositivos remotos e fornece um fluxo de dados em tempo real entre estes dispositivos e o sistema de gestão. Com o crescimento do uso de dispositivos e necessidade de conecta-los, serão geradas enormes quantidades de dados. Esta situação faz com que as operações nos DCs sofram transformações de modo a garantir a segurança de toda a informação. Será preciso implementar plataformas de gerenciamento de dades com capacidades viradas para o futuro, problema que vai ao encontro do problema apresentado.

2.1.2. *Cloud Computing*

Cloud Computing (CC) refere-se tanto aos aplicativos fornecidos pelos serviços da internet como ao *hardware* e sistemas de *software* fornecidos pelos DCs. Este serviços são chamados de *Software as a Service* (SaaS) (Josyula, Orr, Page, & Press, 2011) e são uma maneira eficaz de distribuir e comercializar *software*. CC é uma mudança de paradigma na indústria de *Information Technology* (IT). Muitas das principais empresas de IT, como a Amazon, Google, Microsoft, IBM, HP ou Cisco, afirmam que CC é o próximo passo lógico para controlar os recursos de IT, bem como um meio para reduzir o custo total de empresas fornecedoras destes serviços (Josyula et al., 2011). Mais que uma palavra-chave na indústria da tecnologia, CC promete revolucionar a maneira como os recursos de IT são implementados, configurados e manipulados nos próximos anos. Os fornecedores destes serviços percebem assim que existem

grandes vantagens ao se moverem para este modelo de entrega, "*everything as a service*" (Hung, Giang, Keong, & Zhu, 2012).

A tecnologia IoT tem a necessidade de interligar todos os dispositivos numa rede geral através de um ponto de ligação. A *Cloud Computing* pode ser o ponto de ligação necessário para a tecnologia IoT, e assim, um fator que irá afetar o desempenho dos DCs (Rong, Zhang, Xiao, Li, & Hu, 2016). Como foi referido, CC oferece serviços de IT orientados a serviços públicos para usuários em todo o mundo. Permite "alojar" aplicações do consumidor, científicas e de empresas. No entanto, os DCs que armazenam estas aplicações *Cloud* consomem enormes quantidades de energia, contribuindo para os altos custos operacionais e o crescimento nas emissões de carbono para o ambiente (Uddin, Darabidarabkhani, Shah, & Memon, 2015).

Posto isto, as empresas acharam necessário encontrar soluções não só para economizar energia para o ambiente como também reduzir os custos operacionais, que vai ao encontro do problema apresentado.

2.1.3. A relação entre *Big Data* e *Data Centers*

Data Center (DC) é conhecido como um local onde existe uma grande quantidade de equipamentos de armazenamento e processamento de dados. Tornaram-se o núcleo dos ambientes dos negócios modernos, como já foi mencionado, desde que o processo de computação começou a ser realizado na *Cloud*. O recente desenvolvimento de poderosos *frameworks* de distribuição de dados, como o MapReduce ou Hadoop (apresentados no capítulo 2.2), bem como serviços *web*, motores de pesquisa, comércio *online* e redes sociais levaram ao crescimento sem precedentes no tamanho e complexidade de conjuntos de dados, armazenados em dezenas de milhares de máquinas (Al-Fares, Radhakrishnan, & Raghavan, 2010).

Os DCs modernos são uma complexa relação entre diversos sistemas informáticos, computacionais, mecânicos, elétricos e de controlo. A maioria dos equipamentos que podem ser vistos num DC são servidores, montados em *racks* de aço, geralmente colocados em salas individuais. DCs são projetados de modo a respeitar padrões internacionais rigorosos – os edifícios requerem uma alta segurança física e lógica. Tipicamente projetado para ter uma segurança extrema, os DCs podem armazenar milhares de servidores e bancos de dados e processar uma grande quantidade de informações. É necessário ter um ambiente operacional permanentemente monitorizado de modo a controlar a temperatura, fumo, acesso restrito a funcionários e clientes bem como outros aspetos físicos e lógicos que para o caso de estudo pretendido não são relevantes (Song, Zhang, & Eriksson, 2015).

Com o crescimento exponencial de informação combinado com a evolução de CC e a tecnologia IoT cada vez mais sólida, os DCs tornam-se menos capazes de lidar com o processamento de dados. É necessária uma mudança na arquitetura de monitorização e segurança para garantir o bom funcionamento e qualidade no armazenamento e processamento da informação.

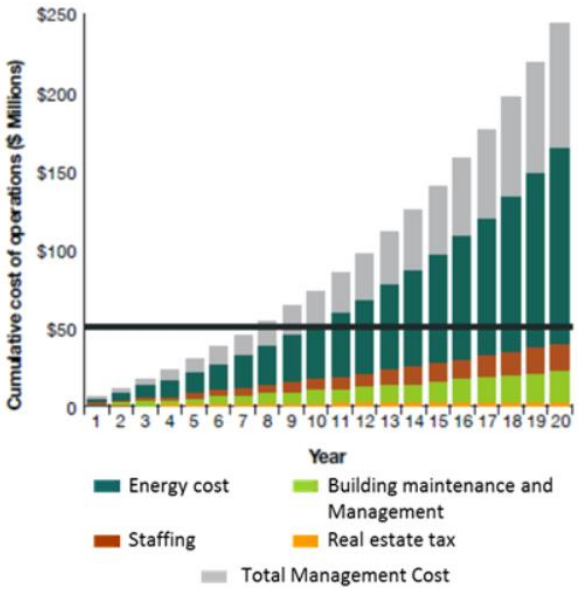


Figura 2 - Gráfico que mostra o crescimento exponencial em vários sectores num DC (Panduit Corporation, 2013)

É possível verificar esse crescimento exponencial no gráfico 2, não só o crescimento da quantidade de dados, mas também o crescimento de outros parâmetros como o número de funcionários, custos de energia, impostos e na manutenção e gerenciamento dos edifícios.

2.2. Processamento de Dados

Nos últimos anos um grande número de programadores, especialmente da Google (Cattell, 2011), implementaram vários programas com o propósito de processar grandes quantidades de dados, como documentos indexados, registos de solicitações na *web*, entre outros servidores geradores de *Big Data*. Estes programas trabalham de uma forma direta, ou seja, são utilizados para conjuntos pequenos de dados que são processados localmente. No entanto, os dados de entrada geralmente são grandes e os programas têm de ser distribuídos entre centenas ou milhares de máquinas para terminar o processamento num período de tempo razoável (Inoubli, Aridhi, Mezni, & Jung, 2016).

Neste capítulo serão apresentadas algumas ferramentas de armazenamento e processamento de dados, comparadas de modo a perceber a melhor solução para resolver o problema proposto.

2.2.1. Armazenamento

Quando se fala em armazenamento de dados é preciso referir duas ferramentas – SQL e NoSQL. Estas ferramentas permitem armazenar os dados de forma otimizada, permitindo ainda a sua recuperação. No entanto, estas ferramentas têm comportamentos diferentes, específicos para cada caso. A distribuição de bases de dados SQL para uma escala maior torna-se difícil. Devido ao crescimento e aumento da complexidade do mundo digital, ou seja, ao crescimento de *Big Data*, a utilização das ferramentas de tratamento de dados convencionais deixou de ser eficiente ou até mesmo impossível para processar tanta informação. Foram então criadas outras ferramentas específicas para esta evolução (Moniruzzaman & Hossain, 2013). Bases de dados NoSQL tentam resolver alguns dos problemas que as bases de dados SQL tradicionais apresentam. Ao não tornar importante certos aspetos, ou até mesmo apresentar uma disponibilidade diferente, NoSQL pode funcionar melhor em circunstâncias onde essas restrições não são necessárias. Isto torna possível adicionar mais servidores em paralelo com o aumento dos clientes (van der Veen, van der Waaij, & Meijer, 2012). No entanto, muitas das grandes organizações afirmam que utilizando o melhor dos "dois mundos", um sistema hibrido, seja a melhor solução (Vilaça, Cruz, Pereira, & Oliveira, 2013).

2.2.1.1. SQL

SQL (*Structured Query Language*) é usado em programação e desenhado para tratar os dados numa "*relational database management system"* (RDBMS). RDBMS é um sistema de tratamento de uma base de dados baseado num modelo relacional e permite aos usuários criar, atualizar e administrar essa base de dados (Frank, Omiecinski, & Navathe, 1992). Tem sido um mecanismo *standart* de armazenamento de dados por mais de quatro décadas. O SQL é usado para consultar, atualizar, inserir e modificar a informação da base de dados, sendo uma grande vantagem para os administradores de DCs, pois são frequentemente obrigados a trabalhar com diferentes plataformas de bases de dados (Li & Manoharan, 2013).

SQL é geralmente considerado o padrão da programação de bases de dados. É de uso simples e há diversos recursos para a aprendizagem de SQL. As *queries* (pode-se compreender como perguntas) são escritas especificando declaradamente a forma dos resultados, e é tarefa do *software* entender a melhor maneira de aceder aos dados. É uma ferramenta flexível, embora exista muitos SQL *standards*, muitas empresas desenvolveram extensões de *open source* para o

SQL. Por exemplo, a linguagem de base de dados ATLAS é considerada uma extensão de SQL, que concede aos usuários implementar aplicações intensivas para bases de dados, escrevendo novas funções de agregação e tabelamento (Wang, Zaniolo, & Luo, 2003).

Resumindo, SQL é um sistema aprovado por todas as companhias que necessitem de tratamento de dados. É uma ferramenta fácil, familiar, compatível e poderosa de escrever *queries*. No entanto, existem algumas desvantagens em relação ao SQL – é difícil de criar uma interface gráfica para os usuários, considerado um processo complexo que necessita de várias linhas de código, requer conhecimento detalhado sobre a estrutura da base de dados, é necessário código repetido para tabelas diferentes e os mecanismos de encapsulamento podem tornar-se ineficientes (Kumar, Gupta, Charu, Bansal, & Yadav, 2014).

2.2.1.2. NoSQL

NoSQL (*Not-only* SQL) fornece um mecanismo diferente comparado com o SQL, diferente das relações em tabelas como na RDBMS. Esta ferramenta apareceu principalmente devido a grandes companhias como o Facebook, Google ou Amazon, que apresentavam desafios em lidar com enormes quantidades de dados que as soluções RDBMS convencionais não conseguiam lidar (Moniruzzaman & Hossain, 2013). NoSQL é cada vez mais utilizado em grandes dados e aplicações *web* em tempo real. Estes tipos de sistemas caracterizam-se por serem não-relacionais, distribuídos, *open source* e por ser expansível horizontalmente (Cattell, 2011). Neste caso, programadores trabalham com aplicativos que geram enormes volumes de novos tipos de dados e rapidamente transformáveis: estruturados, semiestruturados ou não estruturados. Aplicativos que foram usados para um número finito, com NoSQL são agora entregues como serviços que devem estar sempre ativos e acessíveis a partir de diferentes dispositivos e dimensionados globalmente para milhões de usuários. As grandes organizações viraram-se para uma arquitetura de escalabilidade usando servidores *open source*, e CC, em vez de grandes servidores e infraestruturas de armazenamento.

Conclui-se então que bases de dados RDBMS não foram implementadas para ser expandidas e ter a agilidade pretendida para os desafios lançados pelas aplicações modernas. Os requisitos computacionais e de armazenamento de aplicativos como o *Big Data*, inteligência de mercado e redes sociais empurraram o SQL como bases de dados centralizados para os seus limites. Isso levou ao desenvolvimento de bases de dados NoSQL, horizontalmente expansível e não-relacionalmente distribuídos. Dos principais usos do NoSQL, destaca-se o processamento de dados em grande escala, busca e recuperação de informações básicas de máquina em máquina, análise exploratória de dados semiestruturados e armazenamento de dados de enorme volume.

Isto torna NoSQL numa grande ferramenta para BD, uma ferramenta que continua em expansão e melhoramento (Li & Manoharan, 2013).

2.2.1.3. Comparação entre SQL e NoSQL

SQL e NoSQL foram grandes inovações ao longo das últimas décadas na área de tratamento de dados e têm sido utilizados para manter o armazenamento e recuperação de dados otimizada da melhor maneira. É talvez impossível de ignorar uma das ferramentas e trabalhar completamente com a outra. Ambas as tecnologias são as melhores no que fazem, e é decisão do *developer* escolher a que melhor se adequa à solução de um determinado problema. Embora os bancos de dados NoSQL estejam a criar grande impacto e a tornarem-se num importante sistema, grandes empresas criam sistemas híbridos, utilizando funcionalidades das duas tecnologias. Muitos dos populares sistemas NoSQL adicionaram as ferramentas *query* do sistema SQL, criando uma base de dados hibrida. Este sistema hibrido combina a elevada disponibilidade, escalabilidade do NoSQL e a funcionalidade das bases de dados SQL tradicionais (Vilaça et al., 2013). Um exemplo disto é o sistema criado pela Google – "F1" - *A Distributed SQL Database That Scales*. "F1" é um sistema de base de dados híbrido que combina a alta disponibilidade, escalabilidade de um sistema NoSQL e consistência e usabilidade das bases de dados SQL tradicionais. Este sistema funciona de modo a fornecer uma replicação síncrona entre DCs cruzados e uma forte consistência (Shute et al., 2013).

Normalmente afirma-se que NoSQL é um termo criado para inferiorizar a tecnologia apresentada pelo SQL. Na realidade, o termo significa "Not-Only SQL". A ideia é que ambas as tecnologias possam coexistir e cada uma será usada para objetivos específicos (Moniruzzaman & Hossain, 2013). A seguir é apresentada uma tabela a comparar alguns parâmetros destas duas tecnologias, com o objetivo de justificar as diferenças entre elas.

Tabela 1 - SQL vs NoSQL

(Gentz, Sunny, & Lucas, 2016)

	NoSQL	**SQL**
Modelo	• Não Relacional • Guarda dados em: doc. JSON, chaves/pares, gráficos	• Relacional • Guarda dados em tabelas
Dados	• Oferece flexibilidade sem que todos os dados possuam as mesmas propriedades • Novas propriedades podem ser adicionadas • Semiestruturados, complexos, dados aninhados	• Ótimo quando os dados têm as mesmas propriedades • Adicionar propriedades requer alterar o esquema da base de dados • Dados estruturados
Esquema	• Dinâmico e flexível	• Restrito
Transações	• Transações ACID variam por solução	Suporta transações ACID
Consistência e Disponibilidade	• Forte consistência depende da solução • Consistência, disponibilidade e *performance* podem mudar consoante as necessidades	• Consistência forte aplicada • Consistência é prioridade sobre a disponibilidade e *performance*
Performance	• *Performance* pode ser maximizada diminuindo a consistência • Toda a informação sobre entidades é singular, alterações podem acontecer numa só operação	• Depende da velocidade de escrita e da consistência. Pode ser maximizada utilizado recursos de escalabilidade e estruturas de *memory-in*. • Informações sobre uma entidade podem estar espalhadas por várias tabelas ou linhas, requerem muitas junções para completar uma *query*
Escalabilidade	• Horizontal	• Vertical

2.2.2. Análise

As grandes organizações, ou até mesmo programadores particulares, que precisam de trabalhar com uma base de dados têm uma gama de plataformas *open source* por onde escolher com o propósito de analisar dados. No tratamento de BD, diferentes plataformas foram construídas com diferentes finalidades (Pääkkönen & Pakkala, 2015). Aqui neste capítulo serão apresentadas algumas tecnologias interessantes para o problema proposto, tecnologias estas que possuem características importantes como processamento em *real-time*, processamento em *batch*, *open source frameworks* ou abstração para línguas de programação. Apacha Flink, Apache Spark, Apache Storm e Apache Hadoop são os *frameworks* em estudo e onde é possível encontrar as características necessárias para a resolução do problema.

Cada uma destas plataformas tem um conjunto diferente de características e vantagens, cabendo ao utilizador escolher o melhor para o seu problema. Para processar grandes quantidades de dados é necessário recorrer a estas plataformas de processamento. Serão comparadas com o objetivo de perceber em que situações cada uma é mais eficaz, e para concluir qual das ferramentas se adequará mais ao caso de estudo.

2.2.2.1. *Apache Flink*

Apache Flink é um sistema *open-source* para o processamento de dados via *streaming* (processamento de fluxo de dados) e *batch* (processamento de dados estáticos). É uma plataforma de computação de *clusters* com poderosas abstrações de programação em Java e Scala (Bez, 2015), um tempo de execução de alto desempenho e otimizações automáticas de programas, oferecendo um suporte nativo para as iterações (Carbone et al., 2015a). É uma verdadeira plataforma de fluxo de dados que usa uma arquitetura de alto rendimento. As operações como o *Map, Reduce, Union, FlatMap, MapPartition, Filter,* entre outras, podem ser encadeados num *Directed Acyclic Graph* (DAG) de modo que os algoritmos mais complexos possam ser expressos num único fluxo de dados (Junghanns, Petermann, Teichmann, Gómez, & Rahm, 2016). *Flink* processa fluxos de dados em tempo real (*real-time streaming*), realizando a análise destes no momento em que eles chegam. É também uma poderosa ferramenta para o processamento *batch*, isto é, executa uma série de funções num programa que esteja a correr no computador. Lida com o processamento *batch* como se se tratasse de um caso especial de *streaming*. Outra grande vantagem do Flink é permitir aos programadores usarem processos criados noutras plataformas (*Hadoop, Storm,* etc.) no motor do *Flink*, devido à sua forte compatibilidade.

Conclui-se então que o *Flink* se baseia numa filosofia de muitas classes de aplicativos de processamento de dados, incluindo a análise em tempo real, *pipeline* contínuo de dados, processamento de dados estáticos (*batch*) e algoritmos iterativos (*machine learning*, análise de gráficos). Oferece ainda um processamento rápido e capacidade de executar processos existentes criados noutras plataformas (Carbone et al., 2015b).

2.2.2.2. *Apache Hadoop*

Apache Hadoop é um sistema *open-source* de computação distribuída que permite aos usuários distribuir o processamento de grandes conjuntos de dados (*batch*) entre *clusters* de computadores para executar modelos de programação simples, não suportando processamento em *streaming* (Yazici, Alayyoub, & Karakaya, 2016). Esta técnica possui uma *framework* com base de programação em Java, suportando assim o processamento de *data sets* grandes num ambiente de distribuição computacional. É escalável de um único para milhares de servidores, e cada servidor conectado pode fornecer os seus próprios serviços de computação e armazenamento. O seu sistema *Hadoop Distributed File System* (HDFS) facilita as taxas de transferência rápida de dados entre os nós e permite que o sistema continue a operar sem interrupção no caso de falhas (Kumar, Gupta, Charu, & Jangir, 2014). Entre muitas vantagens, destacam-se: funcionalmente distribuído, tornando as redes mais robustas; caso aconteça uma falha num *cluster*, continuará a

ser executado; não é necessário exigir aplicativos para enviar grandes quantidades de dados através da rede; escala linearmente, armazenando quantidades significativas de dados.

Muito utilizado pelas grandes organizações que trabalham com *Big Data*, este sistema processa e armazena grandes conjuntos de dados de uma forma rápida e eficiente. A *framework* do *Hadoop* é usado, por exemplo, pela Google, Yahoo e IBM, maioritariamente para aplicações que envolvam motores de busca e publicidades (Kumar, Gupta, Charu, & Jangir, 2014). O Facebook também escolheu este *framework* para o seu sistema de mensagens. Para isto, foi necessário juntar a tecnologia do *Hadoop* com o *HBase* (*framework* altamente escalável e rápido a entregar *random writes* e *random streaming reads*, para suportar todas as mensagens por dia), criando um sistema em tempo real rápido, eficaz e com o mínimo *delay* possível (Borthakur et al., 2011).

2.2.2.3. *Apache Storm*

Processar dados em tempo real torna-se cada vez mais necessário devido à evolução do *Big Data*. O *Apache Storm* é considerado como o *framework* mais fiável no setor de processamento de dados em tempo real e tolerante a falhas (Iqbal & Soomro, 2015), é considerado também como o "*Hadoop* do processamento em tempo real" . A topologia deste sistema é concebida como um *Directed Acyclic Graph* (DAG), assim como o *Flink*. Funciona como um recetor de dados provenientes de fontes externas, criando fluxos para suportar o processamento em tempo real. Grandes organizações como a Yahoo integraram o Storm com a tecnologia YARN (um gerenciamento de dependências) obtendo um sistema muito poderoso de análise em tempo real, de *machine learning* e monitorização continua (Kumar, Gupta, Charu, & Jangir, 2014).

Cinco atributos chave fazem do *Apache Storm* umas das primeiras escolhas para processamento em tempo real de dados sem limites: fácil de usar; rápido a processar milhões de bytes de dados por segundo; tolerante a falhas, isto é, caso um nó que interliga o sistema com a fonte de dados morrer, o processo reinicia noutro nó; confiabilidade, garantia do processamento de dados pelo menos uma vez; escalabilidade e processamento de dados em paralelo através de um *cluster* de máquinas (Iqbal & Soomro, 2015).

Todas estas características fornecidas pelo *Storm* garantem todas as condições para o processamento em tempo real. Baixa latência caracteriza o processamento do sistema, assim como a compatibilidade com praticamente todas as linguagens de programação e a alta escalabilidade que oferece. É sem dúvida das ferramentas que melhor controla a evolução de *Big Data* (Iqbal & Soomro, 2015).

2.2.2.4. *Apache Spark*

A arquitetura *Spark* é considerada como a próxima geração *open source* de processamento de dados que combina *batch*, *streaming* e análise interativa em todos os dados de uma plataforma com capacidade de recurso a memória. É um sistema de computação *cluster* do Apache, especializada em tornar a análise de dados mais rápida. Alta velocidade não só na execução de programas, bem como na escrita de dados. Como mencionado, suporta a computação em memória, o que permite consultar dados de um modo rápido e eficaz em comparação com as outras plataformas mencionadas (Iqbal & Soomro, 2015). Fácil de usar, proporciona uma capacidade rápida de escrever aplicativos com operadores internos e APIs. Estudos realizados com o API do Twitter revelam que esta plataforma é interativa, flexível e muito competitiva em termos de processamento em *stream*, mostrando níveis de atraso muito baixos (Iqbal & Soomro, 2015). O *Spark* suporta também uma variedade de linguagens como Java, Python e Scala e pode ser executado em cima de processos criados no *Hadoop*. Foca também a possibilidade de implementação de algoritmos que, inerentemente, realizam inúmeras iterações sobre dados, como por exemplo algoritmos de *machine learning*, facto importante para o caso de estudo apresentado. Estudos concluem que o *framework Spark* é 20 vezes mais rápido que o *Hadoop* no que se refere ao uso dos algoritmos citados (Shoro & Soomro, 2015).

No entanto, esta ferramenta apresenta uma desvantagem importante. Caso os seus processos não sejam bem ajustados, poderá haver problemas de memória (erros *out-of-memory*) afectando o seu próprio processo de armazenamento (Rana & Deshmukh, 2015).

2.2.3. Comparação das plataformas

Diferentes plataformas foram criadas para o tratamento de *Big Data*, cada uma com diferentes finalidades. *Hadoop* e *Spark* são as estruturas mais conhecidas e desenvolvidas, comparadas com o *Flink* e *Storm*. São estruturas de processamento *batch*, ou seja, de conjuntos de dados estáticos, enquanto o *Flink* e o *Storm* são *frameworks* de processamento em tempo real, com ferramentas de *streaming* muito poderosas. No entanto, muitas destas plataformas foram construídas de modo a processar dos dois modos, como por exemplo o *Flink*.

Cada *framework* tem as suas vantagens e desvantagens, umas processam de tal modo, outras possuem uma velocidade de processamento mais rápido, e cabe ao usuário escolher a melhor plataforma para a finalidade pretendida. Na tabela **2** é apresentada uma comparação de alguns parâmetros entre as quatro plataformas referidas. *Hadoop* é atualmente uma das soluções mais utilizadas. O ecossistema do *Hadoop* é dotado de um conjunto rico de ferramentas que torna possível a gestão de *clusters* com grande porte. Juntamente com a tecnologia YARN, torna-se

numa opção bastante adequada para configurar soluções de *Big Data* em vários nós. Um exemplo prático, o *Hadoop* é usado pelo Yahoo para gerenciar 24 milhares de nós (Inoubli et al., 2016). Esta *framework* armazena os dados em HDFS como mencionado, e não permite computação iterativa, enquanto que o *Flink*, *Storm* e *Spark* permitem diferentes fontes de dados e computação iterativa. Conforme verificamos na tabela **2**, a importância do Spark reside nas suas características no uso de memória e capacidades de processamento de *batch*, especialmente no processamento iterativo e incremental (Bajaber et al., 2016). Além disso, *Spark* oferece ferramentas que permitem explorar *clusters* em tempo real. Embora este *framework* seja conhecida como a estrutura mais rápida, devido ao conceito de RDD (*Resilient Distributed Dataset*) não é uma escolha adequada num caso em que seja necessário processar dados complexos. Para isto, o *Flink* pode ser um bom candidato. O mecanismo do *framework* do *Flink* transforma cada tarefa num gráfico otimizado, o que permite superar o *Spark* no processamento de dados complexos (Inoubli et al., 2016). O *Flink* compartilha semelhanças e características com o *Spark*. Oferece um bom desempenho de processamento ao lidar com estruturas complexas de *Big Data*, como gráficos por exemplo. Embora existam outras soluções para processamento de gráficos em grande escala, *Flink* e *Spark* são enriquecidos com APIs e ferramentas específicas para *machine learning*, análise preditiva e análise de fluxo de gráficos.

Tabela 2 - Comparação de alguns parâmetros entre frameworks de Big Data (Inoubli et al., 2016)

	Flink	***Hadoop***	***Storm***	***Spark***
Formato dos dados	Key-Value	Key-Value	Key-Value	RDD
Modo de processamento	Batch/Stream	Batch	Stream	Batch/Stream
Fontes de dados	Múltiplos	HDFS	Spoots	HDFS, DBMS e KAFKA
Modelo de Programação	TrasnformAt	Map/Reduce	Bolts	Transformation/Action
Línguas de programação suportadas	Java/Scala	Java	Java	Java, Scala e Python
Cluster Manager	ZOOKEEPER	YARN	YARN ou ZOOKEPPER	YARN
Recursos Partilhados	CPU	Disco	Sem partilha	Memória
Comentários	Extensão de MapReduce com modo gráfico	Armazena grande quantidade de dados no HDFS	Suporta aplicações em tempo real	Fornece vários APIs para desenvolver aplicações interativas
Computação de Interações	Sim	Não	Sim	Sim

2.3. Metodologias de *Machine Learning*

Machine Learning (ML) é um tipo de inteligência artificial que fornece a determinadas máquinas a capacidade de aprender sem que sejam explicitamente programados. Representa a busca pela perfeição de um algoritmo através do seu treino, que produz hipóteses gerais e pode fazer previsões sobre instâncias futuras. Seres humanos muitas vezes tendem a cometer erros durante a análise de dados ou, possivelmente, ao tentar estabelecer relações entre vários recursos. Isto torna difícil a procura de soluções para um determinado problema. ML pode muitas vezes ser aplicado com sucesso a estes problemas, melhorando a eficiência dos sistemas e dos desempenhos das máquinas (Kotsiantis, 2007).

ML concentra-se no desenvolvimento de programas de computador que podem ser treinados para crescer e mudar quando expostos a novos dados. No entanto, em vez de se extrair dados para a compreensão humana, ML usa esses novos dados para detetar padrões e ajustar as suas ações. ML precisa de ser treinada com dados (*training data*). Existem dois tipos de ML – supervisionada e não supervisionada. ML supervisionada requere que a *training data* seja rotulada (*labelled*), ou seja, os tipos e parâmetros dos dados são conhecidos. Depois de analisar os dados de treino, uma função é produzida. Esta função é chamada de classificador (*classifier*) caso seja uma saída no modo discreto ou regressão (*regression*), caso a saída seja em modo contínuo. Esta função tem o objetivo de prever o valor de saída para uma determinada entrada válida (Kotsiantis, 2007). Não supervisionado utiliza conjuntos de dados não rotulados (*unlabelled*). O objetivo deste tipo de ML é descrever os aspetos de um determinado conjunto de dados, como é distribuído por exemplo, em vez de tentar prever valores de saída. Resumindo, transforma os dados numa representação que torna mais fácil entender e usá-lo para outros objetivos. No entanto, é possível usar *data sets* onde algum dos dados sejam rotulados e outros não rotulados – ML semi-supervisionado. Este conceito tem as suas vantagem pois é bastante mais fácil encontrar dados não rotulados que rotulados (Bengio, Courville, & Vincent, 2013).

ML está a tornar-se cada vez mais útil com a evolução e crescimento do *Big Data*. Outra grande razão para o uso desta tecnologia é o facto de se adaptar às mudanças de condições num certo caso. Não é viável ter programadores a adaptar constantemente o código de extração de informação. Nestes casos, torna-se crucial o fato de ML poder aprender constantemente com novos dados. Um exemplo interessante são os motores de busca, que usam algoritmos de ML para calcular probabilidades e previsões de modo a que as publicidades apresentadas aos usuários estejam relacionadas com os termos procurados (Com, 2010).

2.3.1. *Machine Learning e Casos de Estudo*

Muitas são as funcionalidades de ML usadas nos dias de hoje. A evolução na tecnologia leva grandes organizações a confiarem o controlo e monitorização da mudança de dados a algoritmos de ML. Como foi referido no exemplo atrás, o *machine learning* é usado para a escolha da publicidade do usuário através do conteúdo que este procura nos motores de busca. *Bing*, o motor de busca da Microsoft utiliza algoritmos ML para tratar deste caso. Chama-se *Bayesian Click-Through Rate* e é um algoritmo de previsão usado na pesquisa patrocinada (*sponsored search*) pelo *Bing* (Com, 2010).

Outro exemplo onde a aplicação de ML tem sucesso é nas energias renováveis. A previsão da potência de saída dos sistemas de painéis voltaicos é importante para o bom funcionamento da rede elétrica e gestão da energia. Para isto é necessário a previsão da irradiação solar, realizada através das imagens de nuvens combinadas com modelos físicos e tecnologia de ML (Voyant et al., 2017). Continuando no setor da energia, um problema bastante atual são os requisitos energéticos de edifícios, que constituem uma grande percentagem da energia consumida em todo o mundo. Uma estimativa precoce de energia pode ajudar muito os arquitetos e engenheiros a criar estruturas sustentáveis. Estudos propõe um método inovador para estimar o consumo de energia dos edifícios com base num método chamado ELM, *Extreme Learning Machine*. Este método é aplicado às espessuras do material de construção e à sua capacidade de isolamento térmico, sendo realizadas mais de 150 simulações para diferentes valores (Naji et al., 2016).

Na área da saúde, são também muitos os casos onde ferramentas de ML são aplicadas. Os avanços notáveis da biotecnologia e das ciências da saúde levaram à produção significativa de dados, como dados genéticos e informações clínicas, geradas a partir de grandes registos de saúde. Para isto, a aplicação de métodos de ML é, nos dias de hoje, vital e indispensável nos esforços para processar de um modo inteligente toda a informação disponível. Como exemplo particular, temos os diabetes, definidos como um grupo de distúrbios metabólicos que exercem uma pressão significativa sobre a saúde humana em todo o mundo. A pesquisa extensiva sobre todos os aspetos dos diabetes conduz a uma grande quantidade de dados. Usando técnicas de ML, é possível extrair um conhecimento valioso que conduz a novas hipóteses na compreensão dos sintomas dos diabetes nos seres humanos (Kavakiotis et al., 2017). Outro exemplo prático é no controlo da obesidade e excesso de peso no crescimento juvenil. ML pode ser usado para produzir classificações sobre eventos específicos de atividade física (Fergus et al., 2017).

Como é possível concluir, ML pode ser aplicado em diversos sectores, para controlar a evolução de *Big Data*, analisar e prever consumos de energia de modo a minimiza-los, ou até mesmo sendo muito importante no sector da saúde. Muitos são os exemplos desta tecnologia, que continua a desenvolver-se e a ser usada cada vez mais, com o principal objetivo de substituir algumas atividades humanas para diminuir erros e falhas.

2.3.2. Otimização dos *Data Centers*

Como já foi referido, devido à quantidade de servidores dentro dos DCs, a temperatura tende a subir, tornando esses servidores menos eficientes e podem até mesmo ser danificados. A evolução da tecnologia, do *Big Data*, do *Cloud Computing*, do aumento no uso de redes socias, todos estes aspetos contribuem para um aumento no consumo de energia por parte dos DCs.

Para controlar o aumento da temperatura é necessário o uso, como referido, de *chillers*. O desenvolvimento de poderosos *chillers* e ar condicionados permitiu aos DCs modernos instalar muitos servidores na mesma sala. No entanto, os *chillers* consomem enormes quantidades de energia, no que resulta numa grande despesa para os proprietários de DCs. Estes consumos devem-se ao uso inapropriado destes aparelhos – em muitos dos casos atuam de maneira igual tanto para uma pequena como para uma grande variação de temperatura. Soluções de inteligência artificial, como ML, serão o futuro para estes dispositivos de arrefecimento, com o objetivo de aumentar a sua eficiência e diminuir os custos ao proprietário, sendo também menos prejudicial para o ambiente. No próximo subcapítulo serão apresentadas soluções de ML relacionadas com este problema.

2.3.2.1. Propostas existentes

A Google, e outras grandes empresas a fornecerem *web service*, realizaram progressos significativos para melhorar a eficiência dos seus DCs, reduzindo assim o ritmo geral da diminuição do PUE (*Power Usage Effectiveness*) nos aparelhos e arrefecimentos instalados. PUE é uma relação que descreve o quão eficiente um servidor de um DC usa a energia, ou seja, quanto energia é usada pelos aparelhos informáticos (em contraste com o arrefecimento e outros custos). Calcula-se com a seguinte fórmula (1):

$$1) \quad PUE = \frac{Energia\ Total\ do\ DC}{Energia\ do\ equipamento\ informático\ do\ DC}$$

Este aumento do PUE significa um melhor e mais eficiente uso da energia. Para mostrar esta redução no ritmo da diminuição é apresentado o gráfico da figura 3, onde é possível verificar

a diminuição do PUE ao longo dos anos. No entanto, é percetível que o ritmo desta diminuição acalmou a partir do ano 2013.

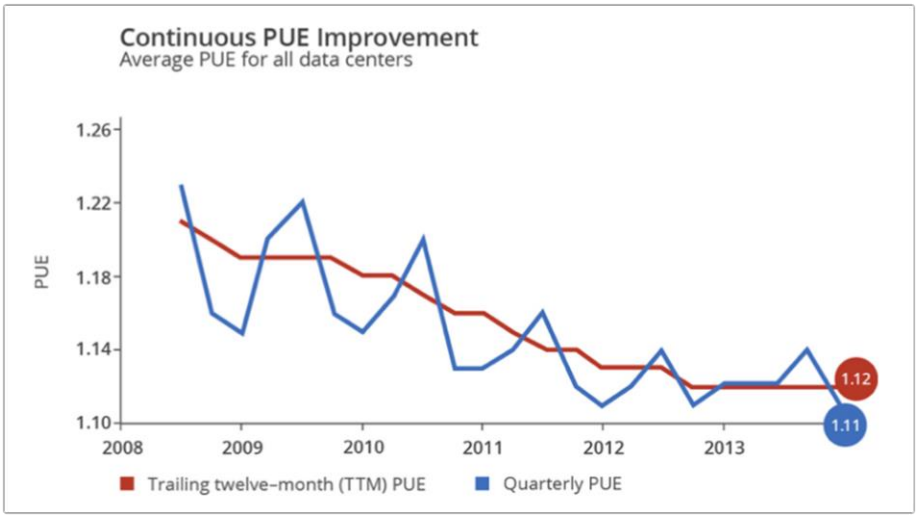


Figura 3 - Alteração do PUE ao longo dos anos
(Gao & Jamidar, 2014)

A aplicação de algoritmos de ML em dados existentes de monitorização fornece uma oportunidade para melhorar significativamente a eficiência operacional de um DC. Um típico DC de grande escala controla milhões de pontos de dados e milhares de sensores todos os dias, embora normalmente esses dados não sejam utilizados para outras aplicações além de monitorização. Os avanços na tecnologia criaram uma grande oportunidade para que ML forneça as melhores práticas e melhorar a eficiência de DCs. Um dos métodos utilizados são as redes neuronais (Gao & Jamidar, 2014). As redes neuronais são uma classe de algoritmos ML que "imitam" o comportamento cognitivo através de interações entre neurônios artificiais. Este modelo é vantajoso para modelar sistemas complicados pois não exigem que o utilizador predefine as interações entre os recursos. Em vez disso, a rede neuronal procura padrões e interações entre os recursos para gerar automaticamente um modelo com o melhor ajuste. Como acontece com quase todos os algoritmos ML, a precisão do modelo melhora ao longo do tempo à medida que novos dados de treino são adquiridos. Neste caso, os parâmetros de entrada serão os aparelhos IT (*Information Technology*) *load* (quantidade de servidores em funcionamento), condições climatéricas, número de *chillers*, *set-points* dos equipamentos, entre outros. Estes parâmetros entram no algoritmo, sendo processados com o objetivo de criar uma função de saída, e avalia o PUE calculado. Num exemplo prático, uma análise interna realizada pelo rede neuronal do PUE em relação à temperatura do corredor da sala no DC pode sugerir uma redução teórica de um determinado *ratio* de PUE através do aumento da temperatura de arrefecimento (Gao & Jamidar,

2014). Esta sugestão tornará mais eficiente o trabalho dos aparelhos de arrefecimento, diminuindo a energia consumida e os custos.

Uma proposta simples de ML para obter um DC eficiente, será usar uma estrutura que forneça uma metodologia inteligente de consolidação usando diferentes técnicas, como ligar/desligar aparelhos, algoritmos de *power-aware* e técnicas de ML para lidar com informações incertas e maximizar o desempenho. Na abordagem de ML, modelos de previsão dos consumos de energia, cargas de CPU e melhoramento nas decisões de agenda serão usados. Utilizando técnicas ML de *sheduling* num sistema real é possível controlar o consumo energético (Berral et al., 2010).

Um dos sistemas mais eficientes até agora é o DeepMind Al (sistema de redes neuronais), da Google. Este sistema de ML reduziu o consumo de energia nos DCs da Google numa média de 15%, onde 40% desse valor refere-se a redução de energia necessária nos processos de arrefecimento. Estes valores fizeram com que a Google poupasse centenas de milhões de dólares por ano (Vicent, 2016). Continuando nas soluções das grandes empresas *Big Data*, o Facebook apresenta o sistema AutoScale. Consiste na alteração da forma com o IT *load* é distribuída para os diferentes servidores *web* em *cluster*. A ideia base do AutoScale é que, em vez de uma abordagem puramente *round-robin* (algoritmo que atribui um intervalo de tempo igual a cada processo e de forma circular), o controlador do IT *load* irá concentrar a carga de trabalho num servidor até que possua um nível médio. Quando a carga de trabalho global for baixa (por exemplo, por volta da meia noite), o controlador do *load* usará apenas um conjunto de servidores. Os outros servidores podem ser deixados numa execução adormecida ou usados para processar amostras de cargas de trabalho, tornando o uso de energia mais eficiente. A tecnologia ML entra neste sistema com o objetivo de prever os momentos do dia em que os servidores terão mais carga de trabalho (Gandhi, Harchol-Balter, Raghunathan, & Kozuch, 2012).

Estes avanços pelas grandes organizações, como a Google, Facebook ou Amazon, resultaram na redução drástica dos custos operacionais, permitindo que empresas com menores recursos IT aumentassem de uma forma rápida e eficiente entre milhões de usuários. Estas tendências resultaram no aumento de DCs e no aumento dos desafios operacionais (Gao & Jamidar, 2014).

2.4. **Conclusões Gerais**

Pela descrição do problema proposto, é possível verificar que existe a necessidade de procurar soluções para a otimização no uso da energia por parte dos DCs, mais concretamente no uso da energia para os processos de refrigeração. Através de uma pesquisa de artigos

científicos relacionados com o caso de estudo foi possível escrever o estado da arte, onde é apresentado a origem do problema base, tecnologias de *Big Data* e de uma ferramenta ainda em evolução, mas que promete revolucionar o mundo da análise de dados – o *machine learning*. No estado da arte são também apresentadas algumas soluções e propostas para o problema do caso de estudo, algumas destas criadas por grandes empresas como a Google ou Facebook. Estas soluções demonstram resultados bastante agradáveis e uma alta *performance* no que diz respeito ao uso eficaz da energia, embora a tecnologia ML seja ainda difícil de se aplicar devido à sua complexidade.

Algumas dificuldades foram encontradas na descrição de certas tecnologias, ou por continuarem em evolução, ou até mesmo por ainda existir pouca informação sobre. No entanto foi possível realizar comparações entre estas de modo a escolher as que melhor solucionam o problema apresentado. Para trabalho futuro, será necessário a recolha de dados, o processamento destes e a escolha do algoritmo ML que melhor se encaixe neste caso de estudo.

Conclui-se, depois desta pesquisa, que o problema apresentado é um caso sério e que é necessário agir de modo a aumentar a eficiência energética. Algumas das grandes empresas, como já foi referido, começaram a agir criando tecnologias avançadas para monitorização de DCs tornando o uso de energia significativamente mais eficaz. No entanto, alguns DCs que operam para pequenas ou médias empresas utilizam a energia de um modo inapropriado, nomeadamente na refrigeração. Muitas destas entidades dependem de como a gestão de um DC é realizada, tornar os DCs mais eficientes não só aumenta a estabilidade económica dos proprietários e usuários, mas também diminui a degradação do meio ambiente.

3

Arquitetura Desenvolvida

O caso de estudo inicial é tornar os *chillers* de um DC o mais eficientes possível através de um bom processamento de dados e algoritmos ML. Devido aos avanços referidos no estado da arte de *Big Data*, os *chillers* tornaram-se ineficientes no seu processo, dado à elevada carga que os servidores estão sujeitos. Alguns DCs não utilizam todo o potencial destes aparelhos. Num exemplo prático do funcionamento dos *chillers* no DC em questão, quando ocorre uma subida da temperatura na sala dos servidores, é necessário que seja tratada para que os servidores trabalhem de um modo mais eficaz e não acabem danificados. É nestas situações que os *chillers* atuam de forma ineficiente. E é neste processo de refrigeração que existe um consumo excessivo por parte dos *chillers*, pois estes atuam de igual modo tanto para uma pequena como para uma grande variação de temperatura. Será neste ponto crítico que a arquitetura desenvolvida irá atuar, com o objetivo de atenuar as perdas energéticas causadas pelo uso pouco eficiente destas máquinas de refrigeração.

De seguida é apresentada a relação entre os *chillers* e os *Data Centers*, como se interligam entre si e a sua comunicação. Nos subcapítulos a seguir é explicado melhor como se obteve o

training data e a criação do algoritmo ML, bem como o esquemático da arquitetura e diagramas de sequência.

3.1. Relação entre *chiller* e *Data Center*

Antes de ser apresentada a arquitetura desenvolvida, é necessário mencionar o processo que realmente acontece num DC. Na Figura 4 encontra-se simplificado a relação entre o *chiller* e a sala do DC onde se encontram os servidores. O *chiller*, através de sensores de temperatura, mede a temperatura da água proveniente da sala (EWT) aplicando um *set point* de arrefecimento (LWT).

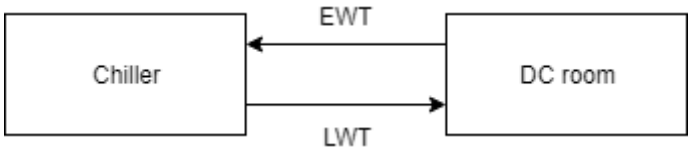


Figura 4 – Relação entre o *chiller e a sala do Data Center*

O que acontece é que, independentemente da subida de temperatura, os *chillers* funcionam da mesma maneira, ou seja, atuam sempre com o mesmo *set point* de temperatura. Ora pode-se perceber que para uma subida menos abruta da temperatura os *chillers* poderão atuar de uma maneira mais subtil, não consumindo energia em demasia. É neste ponto que o caso de estudo vai ao encontro do problema geral nos DCs, e é este caminho que a arquitetura apresentada seguirá.

3.2. Arquitetura

O objetivo fundamental desta arquitetura é desenvolver inteligência artificial, que atuará nos *chillers* de modo a que estes comportem-se de maneira mais eficiente, consoante a variação de temperatura medida. A arquitetura consiste em duas fases – treino e aplicação do algoritmo (Figura 5). Para iniciar a implementação é necessário recolher todos os dados precisos sobre os *chillers*. Estes dados referem-se ao valor do *load* (carga) e da eficiência (kW/kW) dos *chillers* consoante LWT (*Leaving Water Temperature* em °C), EWT (*Entering Water Temperature* em °C) e a temperatura ambiente em °C. O *load* é, em percentagem, o regime em que o *chiller* está a funcionar. Num exemplo prático, uma máquina de 100kW, caso necessite de entregar 50kW, estará com um *load* de 50%.

Para constituir a primeira fase do *training data* que irá treinar o algoritmo ML, os dados mencionados sobre os *chillers* serão adquiridos através do *software* TOPPS, fornecido pela

empresa americana Trane, produtora dos *chillers* instalados no DC. No seguimento da recolha destes dados, é necessário criar um *dataset* e assim procede-se ao treino do algoritmo ML. Note-se que, quanto melhor for o *dataset*, ou seja, maior qualidade nos dados, melhor desempenho terá o algoritmo ML.

3.2.1. *Inputs*

O algoritmo desenvolvido receberá dois *inputs* essenciais para produzir o *output* pretendido – a temperatura da sala (EWT) e a temperatura ambiente. A temperatura da água proveniente do DC é lida através dos sensores do *chiller*, em °C, enquanto a temperatura ambiente será obtida através de uma base de dados em tempo real que indique o valor, em °C, da temperatura ambiente da localidade onde se situa o DC. Estes dois parâmetros vão ser a base do algoritmo, que irá produzir um *setup* de configuração dos *chillers* para que estes trabalhem da forma mais adequada.

3.2.2. Esquema da Arquitetura

Apresenta-se aqui o esquema da arquitetura e as relações entre os diversos módulos presentes na arquitetura. A azul representa os *inputs* mencionados e a verde o *output*, ou seja, o *setup* de configuração dos *chillers*.

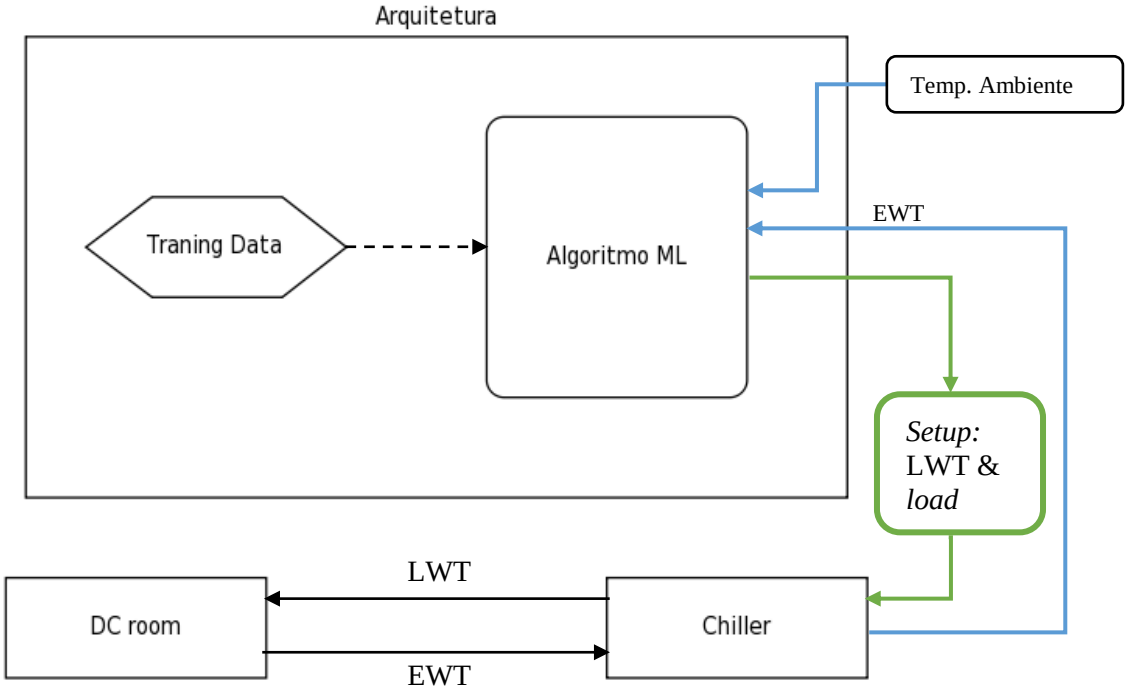


Figura 5 - Arquitetura implementada

Na Figura 5 podemos ver a arquitetura proposta, que por etapas vai ter a seguinte cronologia:

1. Criação do algoritmo ML;
2. Treinar o Algoritmo ML com o *training data* obtido*;*
3. O chiller lê a temperatura da água proveniente do DC (EWT) e envia para o algoritmo para ser processado. A temperatura ambiente também é obtida e enviada para o algoritmo;
4. O algoritmo ML irá escolher o melhor *setup* para os *chillers*, com o objetivo de manter a sala do DC no intervalo de temperatura ideal;
5. O *setup* terá os valores de *load* (carga) a que o *chiller* deve trabalhar e a temperatura que deve entrar no DC (LWT);
6. O *chiller* inicia o processo de refrigeração com os novos valores, enviando a temperatura LWT para a sala do DC.

As etapas e conexões poderão ser verificadas melhor nos diagramas de sequência e atividade presentes no subcapítulo 3.2.3. Esta arquitetura está implementada de modo a que se crie um ciclo de controlo, verificando sempre a temperatura EWT e a temperatura ambiente, processando-as e configurando o melhor *setup* possível.

3.2.3. Diagramas de Sequência e Atividade

Para mostrar melhor as relações existentes na arquitetura, criou-se diagramas de sequência e de atividade. Estes diagramas permitem dar uma ideia sobre a ordem das conexões criadas, as tarefas de cada módulo e as etapas mencionadas nos subcapítulos anteriores.

Como se verifica na Figura 6, depois da criação do algoritmo tudo começa pelo treino deste. Com o treino completo, o algoritmo estará apto para responder a todas as ocorrências de subida de temperatura. Os sensores do *chiller* estão constantemente a ler a temperatura da sala, ao mesmo tempo que o algoritmo obtém a temperatura ambiente do local do DC. Cabe ao algoritmo encontrar o melhor *setup* para o *chiller* funcionar da maneira mais eficiente possível e assim tratar eficazmente as subidas de temperatura apresentadas.

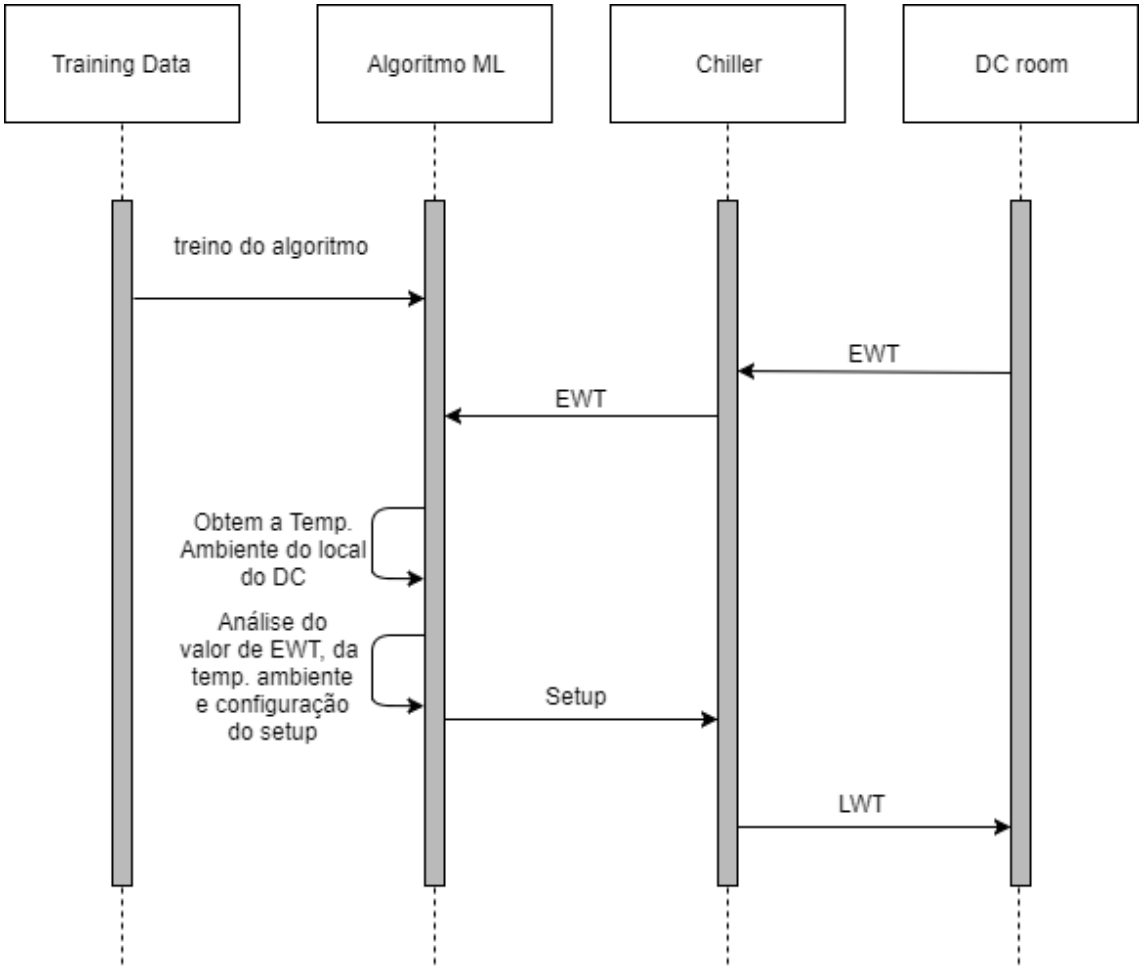


Figura 6-Diagrama de sequência da Arquitetura

Pela observação da Figura 6, torna-se mais fácil perceber o porquê de no caso atual o funcionamento do *chiller* é ineficiente. O valor EWT medido pelo *chiller* nunca é processado pelo algum algoritmo, então o valor de LWT será sempre igual como foi referido, independentemente do valor EWT. Quando se diz que LWT é sempre igual, significa que o *set point* ou a carga do *chiller* será sempreo mesmo. Isto torna o consumo de energia ineficiente, pois o *setup* de configuração do *chiller* será sempre o mesmo.

Verifique-se a atividade geral da arquitetura no diagrama de atividade apresentado na Figura 7. Como é apresentado, o valor EWT e a temperatura ambiente são tratados pelo algoritmo escolhido, para que seja escolhido o melhor *setup* para a situação. O seu funcionamento termina quando não existir mais *inputs* de EWT e temperatura ambiente.

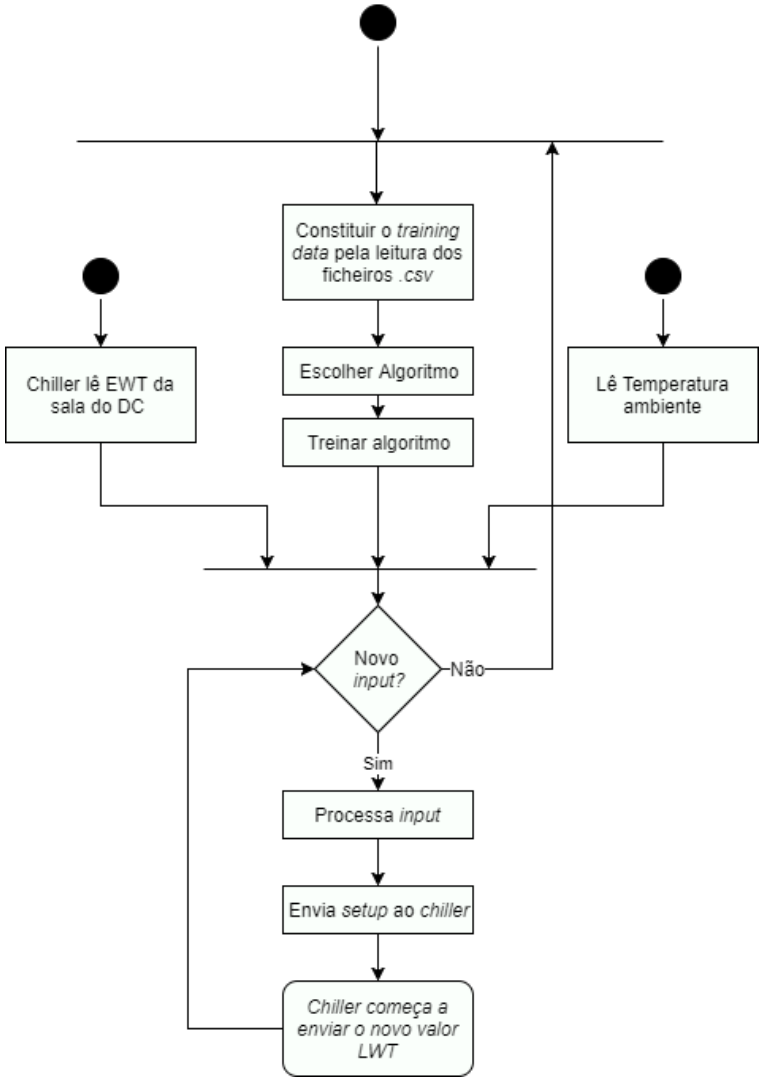


Figura 7 - Diagrama de atividade da arquitetura

3.2.4. *Training Data*

Na fase de treino realizou-se, em primeiro lugar, a recolha de todos os dados sobre os *chillers*. Estes dados foram recolhidos através do *software* mencionado, o TOPSS, com o objetivo de se criar o *training data* necessário, usado para "treinar" o algoritmo ML. Os dados recolhidos do *software* têm vários parâmetros que servirão para o bom funcionamento do algoritmo.

O *input* para a recolha dos dados é o mesmo que para o algoritmo e mais a LWT, para que se obtenha os parâmetros que são interessantes para realizar o *setup* dos *chillers*. Por exemplo, para um *input* específico, ou seja, para determinados valores de EWT, LWT e temperatura ambiente, o programa devolve um valor de *load* e um valor de eficiência, entre outros pouco relevantes para o caso de estudo. Este valor de eficiência vem em kW/kW e será o parâmetro de referência do algoritmo ML.

O TOPSS é fornecido pela empresa americana Trane, que produz sistemas de arrefecimento e aquecimento de grandes construções, como DCs, prédios, centros comerciais, entre outros. Neste *software* é possível escolher a máquina que se quer simular. Para o caso de estudo, o modelo do *chiller* estudado foi o *Air-Cooler Sintesis - Epinal* (ERTAF) com uma tonelagem nominal unitária de 175 a água, modelo este que é usado nas instalações do DC da ITCONIC, empresa que motivou o tema do caso de estudo. Na Figura 8 é possível ver o aspeto físico da máquina.

Figura 8 - Modelo do Chiller usado para simulação
("Air-Cooled Screw Chillers RTAF 300-1900 kW Trane Sintesis," 2014)

Na Figura 9 é possível observar a configuração dos parâmetros para a simulação do *chiller* no TOPSS. Depois de especificar o modelo utilizado, altera-se os valores de entrada, que na figura são denominados por "Evap leaving temp" (LWT), "Evap entering temp" (EWT) e a temperatura ambiente.

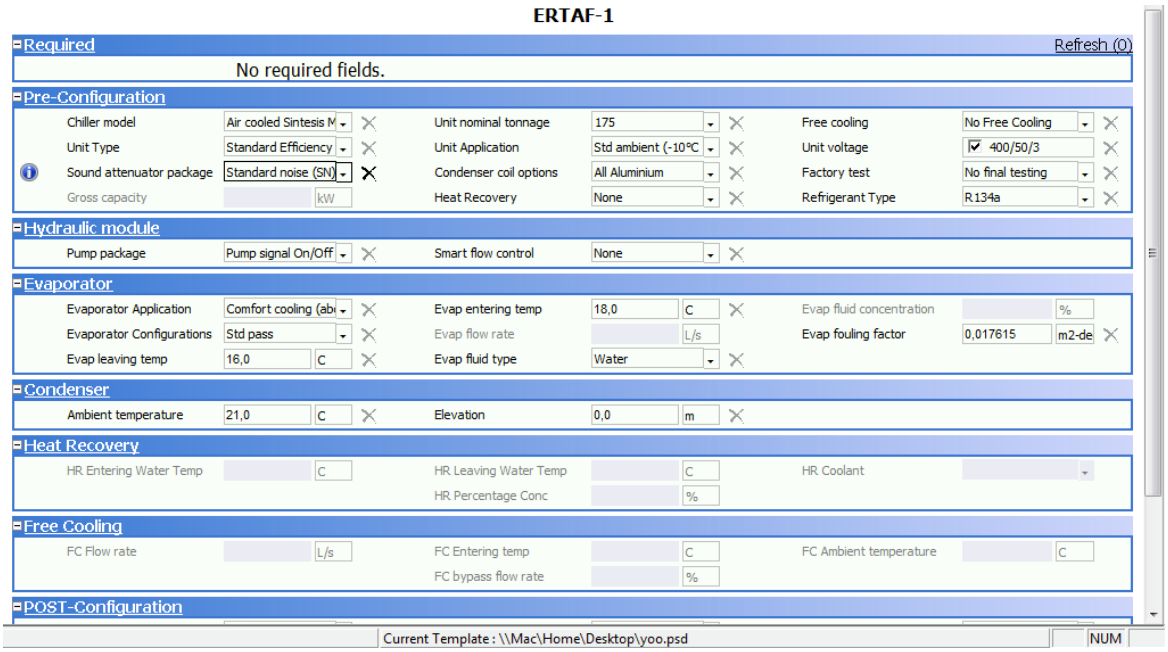


Figura 9 - Configuração da simulação no TOPSS

No caso do TOPSS, o valor de LWT será também um parâmetro de entrada, pois o objetivo é verificar a eficiência do *setup* existente no *chiller* quando existe determinados valores de EWT e temperatura ambiente conjugados com vários valores de LWT. Verifica-se melhor este aspeto na Figura 10.

Load	Gross Cap.	LWT Evap	EWT Evap	Flow Evap	WPD Evap	Ambient	Gross Power	Gross Eff.	Net Cap.	Net Power	Net Eff.
%	kW	C	C	L/s	kPa	C	kW	EER (kW/kW)	kW	kW	EER (kW/kW)
100	946.1	16.0	18.0	113.1	542.8	21.0	208.7	4.53	883.1	271.7	3.25
90	851.2	16.0	17.8	113.1	542.7	21.0	172.2	4.94	788.2	235.2	3.35
80	756.7	16.0	17.6	113.1	542.7	21.0	142.2	5.32	693.7	205.2	3.38
70	662.1	16.0	17.4	113.1	542.6	21.0	119.3	5.55	599.1	182.3	3.29
60	567.6	16.0	17.2	113.1	542.6	21.0	111.7	5.08	504.6	174.7	2.89
50	473.0	16.0	17.0	113.1	542.6	21.0	94.4	5.01	410.0	157.4	2.60
40	378.4	16.0	16.8	113.1	542.5	21.0	69.3	5.46	315.4	132.3	2.38
30	283.8	16.0	16.6	113.1	542.5	21.0	52.5	5.40			

Figura 10 - Resultados da simulação no TOPSS

Na Figura 10 é apresentado um exemplo de uma tabela, em formato *.csv*, que é gerada pela simulação do TOPSS. Neste exemplo apresentado foram escolhidos como parâmetros de entrada os seguintes valores:

- Intervalos de *load* de 10%;
- EWT (Temperatura da água que sai da sala e entra no *chiller*) = 18 °C. De notar que esta temperatura desce 0.2ºC à medida que o *load* desce 10%;
- LWT (Temperatura da água que sai do *chiller* e entra na sala) = 16 °C;
- *Ambient* (Temperatura ambiente) = 21 °C.

Depois de escolhidos os parâmetros, apenas foi preciso correr a simulação e obter a tabela apresentada na Figura 10. Pela observação da tabela, verifica-se que para valores de *loads* diferentes, obtém-se valores de eficiência (*Gross Eff.*) diferentes, tais como outros valores menos relevantes para a resolução do problema. E será este valor de eficiência o cerne do algoritmo ML, pois o principal objetivo deste caso de estudo é fazer com que o *chiller* trabalhe da maneira mais eficiente, ou seja, com o maior valor de eficiência.

Esta simulação foi realizada para vários valores de EWT, LWT e temperatura ambiente. Admitiu-se, depois de uma pesquisa sobre as temperaturas normais num DC e informações provenientes da ITCONIC, que a temperatura admissível da sala dos servidores é, normalmente entre os 20ºC e 25ºC, dependendo do acordo realizado entre os proprietários do DC e os clientes.

Portanto os valores escolhidos para as simulações foram:

- EWT entre 18ºC e 27ºC;
- LWT entre 7ºC e 16ºC;
- Temperatura ambiente entre os 15ºC e 39ºC.

Realizando os cálculos, foram gerados 1944 ficheiros *.csv*, cada ficheiro com 8 valores diferentes de *load*. Concluindo, obteve-se algo semelhante a 15500 hipóteses através das simulações realizadas no TOPSS.

Na Figura 11 é apresentado o diagrama de sequência que mostra a explicação anterior. Uma vez escolhido o modelo do *chiller*, são adicionados os parâmetros (EWT, LWT e temperatura ambiente) e efetua-se a simulação. A simulação gerará um ficheiro *.csv* que será processado e adicionado ao *training data*.

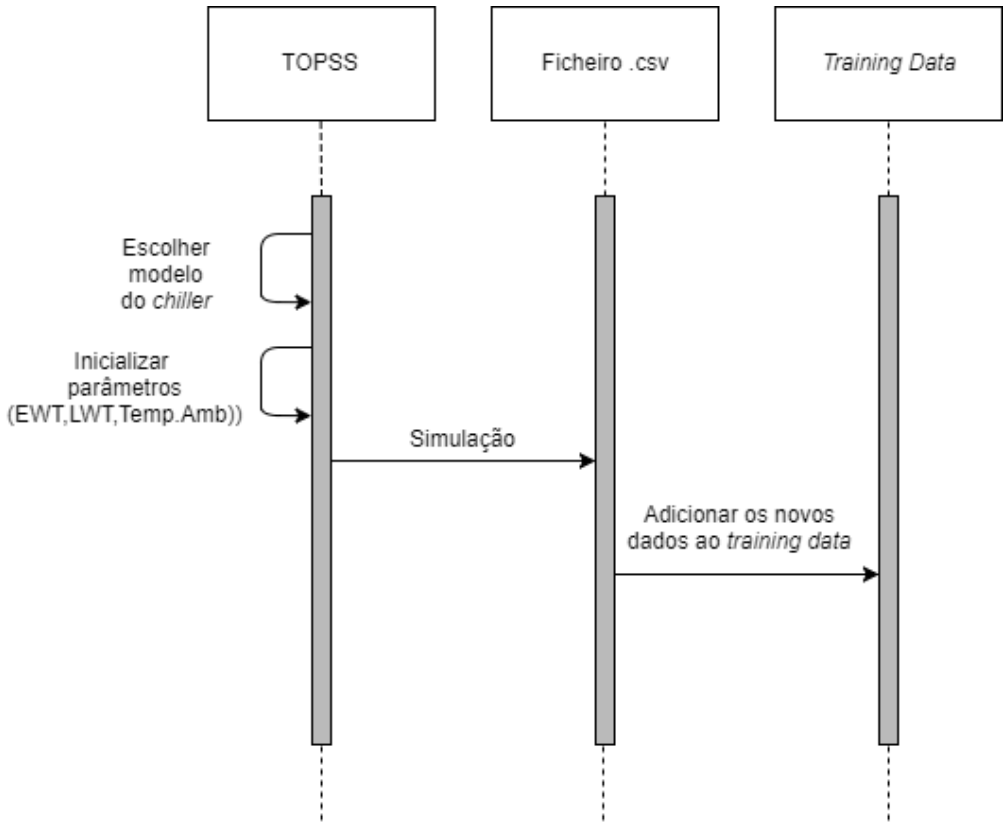


Figura 11 - Diagrama de sequência de como se extrai o *training data*

Para criar o *training data* a arquitetura implementada irá ler cada linha apresentada na tabela da Figura 10 e guardar estes valores em memória. O diagrama de atividade da Figura 12 mostra o procedimento efetuado.

Inicia-se por verificar se há ficheiros *.csv*. Sendo esta verificação positiva, é lida cada linha do ficheiro e os elementos são tratados e adicionados ao *training data*, elementos estes que são os observados na Figura 10.

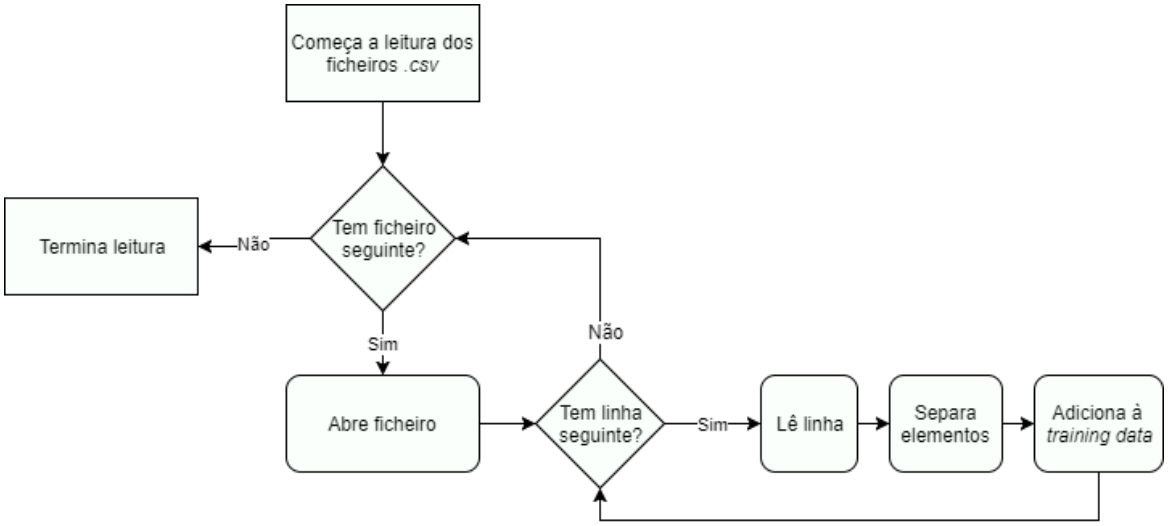


Figura 12 - Diagrama de atividade referente à criação do *training data*

Com o processo de recolha de dados completo, procedeu-se à criação de um *training data* robusto e sólido para o treino do algoritmo. Este *training data* é composto, como se viu anteriormente, por amostras, ou seja, por todas as 15500 hipóteses geradas pelo TOPSS e que serão usadas para que o algoritmo realize as melhores escolhas em relação ao *input* que entre. De seguida é explicada arquitetura do algoritmo e a sua relação com o *training data*.

3.2.5. Algoritmo ML

A implementação do algoritmo *machine learning* é o grande objetivo para a resolução deste caso de estudo. Um algoritmo ML é, como foi apresentado no capítulo 2.3 do estado da arte, um tipo de inteligência artificial que fornece, a determinadas máquinas, a capacidade de aprender sem que sejam explicitamente programadas. Ou seja, o algoritmo será inicialmente programado para que "aprenda" ao longo do seu funcionamento e com a ajuda do *training data*. Por isto, é muito importante obter um *training data* com qualidade, para que o algoritmo possua uma boa capacidade de iniciar uma aprendizagem contínua ao longo do seu funcionamento, prevendo as melhores opções a escolher no futuro.

Nesta arquitetura desenvolvida para o caso de estudo apresentado, o algoritmo ML será o cérebro do *chiller*, e irá tomar as melhores opções consoante a situação que seja apresentada. Como já foi explicado diversas vezes, o *chiller* é a parte *hardware* da arquitetura – é a máquina de refrigeração que irá medir a temperatura do DC e irá atuar constantemente para manter essa

temperatura nos intervalos aceitáveis. No entanto esta refrigeração, sem o algoritmo ML, é ineficiente. Independentemente da temperatura lida pelo *chiller* que se encontra na sala (EWT), este atuará da mesma maneira. Num exemplo muito prático, existe um intervalo ideal que a sala funciona em perfeição, assumiu-se como foi referido entre 20ºC a 25ºC. O objetivo será manter a sala dentro deste intervalo satisfatório. No entanto, manter a sala dentro deste intervalo requer energia que, sem o controlo do algoritmo ML, poderá ser consumida incontrolavelmente tornando o seu uso ineficiente e com elevados custos para o proprietário. Caso se verifique que a temperatura da sala seja de 40ºC ou de 50ºC (ou outro valor a cima do intervalo), ele atuará da mesma maneira (utilizando o mesmo *set point* de temperatura ou a mesma carga), não distinguindo as diferenças de temperatura, que resulta num uso problemático da energia. Torna-se fácil perceber que, no caso descrito, a energia usada para diminuir os 40ºC para o intervalo aceitável será menor que diminuir os 50ºC.

Para mostrar a relação do algoritmo ML com o *training data* e verificar o funcionamento do algoritmo, é apresentado um exemplo muito simples de como o algoritmo deverá proceder, com os seguintes *inputs*:

- O valor lido pelo *chiller* de EWT é 22ºC;
- A temperatura ambiente é de 20ºC;

A resolução deste problema passa por encontrar o maior valor de eficiência no *training data* correspondente ao *input* proposto. Para isto, o primeiro passo do algoritmo será aceder ao *training data* e obter os valores correspondentes ao *input* referido.

Load	Gross Cap.	LWT Evap	EWT Evap	Flow Evap	WPD Evap	Ambient	Gross Power	Gross Eff.	Net Cap.	Net Power	Net Eff.
%	kW	C	C	L/s	kPa	C	kW	EER (kW/kW)	kW	kW	EER (kW/kW)
90	850.1	15.0	22.0	360	630	20.0	180.0	4.50	901.2	200.7	4.30
70	824.4	15.0	22.0	342	613	20.0	174.2	4.90	872.5	186.3	4.80
100	962.7	16.0	22.0	384	643	20.0	208.0	4.63	959.2	211.7	4.53
90	902.5	16.0	22.0	357	629	20.0	176.0	5.23	900.4	196.3	5.13
80	835.7	14.0	22.0	332	624	20.0	173.0	5.63	882.4	194.7	5.53

Figura 13 - Exemplo de *training data* para EWT=22ºC e $T_{ambiente}$=20ºC. Verifica-se que a última hipótese é a que possui o maior valor de eficiência.

Pela observação da tabela da Figura 13 verifica-se que, referente ao *input* proposto, o melhor *setup* possível para resolver esta situação é colocar a máquina a funcionar com um *load* de 80% ou com um *set point* (LWT) a 14ºC. Neste caso, esta será a melhor solução para combater a situação proposta.

Este foi apenas um exemplo de como o algoritmo deverá atuar, não sendo um exemplo prático ou real. No capítulo de análise de dados serão apresentados casos reais da simulação.

Concluindo o funcionamento do algoritmo ML, para cada temperatura haverá um *load* satisfatório para que o *chiller* trabalhe, e uma LWT satisfatória que entre na sala do DC, com objetivo de ser o mais eficiente possível.

Ser o mais eficiente possível neste caso de estudo é consumir o mínimo de energia necessária para que algo seja completado dentro dos padrões admissíveis. E o objetivo do algoritmo ML será este mesmo, colocar a trabalhar o *chiller* em conformidade com a situação que a sala do DC apresente. Para cada EWT diferente, haverá um *setup* diferente que o algoritmo irá escolher e coloca assim o *chiller* a trabalhar consoante esse *setup*.

Uma vez explicado o procedimento da recolha do *training data*, é explicado nos subcapítulos a seguir os algoritmos implementados. Para este caso desenvolveu-se dois algoritmos ML, o KNN (*K-Nearest Neighbours*) e *o K-Means Clustering*. Estes algoritmos têm modos de funcionar diferentes que irão ser explicados, mas com o mesmo o objetivo, aumentar a eficiência energética dos *chillers*.

3.2.5.1. KNN - K Nearest Neighbours

O *K-Nearest Neighbours* (KNN) é um algoritmo *machine learning* bastante usado em regressão e classificação, sendo mais utilizado para classificação, como é o caso desta solução no caso de estudo apresentado. Considerado fácil de interpretar, possui um tempo de cálculo rápido e um poder de previsão forte, o KNN é muito utilizado em estimações de estatística e reconhecimento de padrões.

Em relação ao seu funcionamento, o KNN começa por preparar o *training data*. De seguida é pedido que seja enviado uma *query*, como parâmetro de entrada. Esta *query* é classificada pela maioria de votos dos pontos vizinhos, sendo então atribuída à classe mais comum entre os **k** vizinhos. Observe-se o exemplo da Figura 14, onde a *query* é representada pelo o ponto preto.

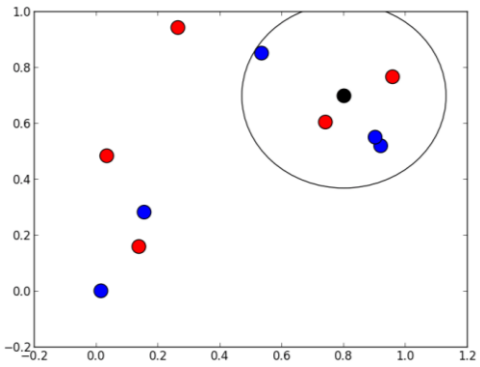


Figura 14 - Exemplo do funcionamento do KNN para um k=5.

No exemplo mostrado, o valor de **k** é 5, ou seja, a *query* será classificada pelas cinco amostras mais próximas. Note-se que o *training data* deste exemplo é constituída pelos pontos vermelhos e azuis. Como se verifica, existem mais pontos azuis que vermelhos nos cinco vizinhos mais próximos, então a *query* será classificada com a cor azul. Em caso de empate, o valor de **k** deverá ser decrementado até haver uma maioria na votação dos vizinhos mais próximos.

O valor **k** pode ser dimensionado pelo programador de modo a que o seu algoritmo seja o mais eficiente possível ou, como neste caso de estudo o *training data* é muito homogéneo, pode-se utilizar a técnica de *bootstrapping*. Esta técnica, bastante usada para encontrar o valor de **k** ótimo, diz que **k** é igual à raiz quadrada do número total de instâncias no *training data* (Hall, Park, & Samworth, 2008).

2) $k = \sqrt{Número\ de\ amostras}$

Devido a certos valores de simulação não aceites pelo TOPSS, algumas amostras não foram recolhidas por serem valores que não correspondem a situações reais no funcionamento dos *chillers*. Sendo assim, a recolha resultou num total de 15214 de amostras que completam o *training data*. Temos então que:

3) $k = \sqrt{15214} = 123.345 \approx \mathbf{124}$

Dimensionado o valor de **k**, o KNN começa a processar a *query*. Quando o algoritmo recebe uma *query*, é de seguida calculada a distância entre ela e todos os pontos presentes no *training data*. Para este cálculo é utlizada a distância euclidiana:

4) $\sqrt{(p_1 - q_1)^2 + (p_2 - p_2)^2 + \cdots + (p_k - p_k)^2} = \sqrt{\sum_{i=1}^{k}(p_i - p_i)^2},$

onde $\mathbf{p_k}$ e $\mathbf{q_k}$ são pontos num espaço n-dimensional e **k** o número de vizinhos dimensionado. Para o caso de estudo, o *training data* utilizado é bidimensional, então a nossa distância é medida pelo seguinte cálculo, onde a $\boldsymbol{Q} = (\boldsymbol{q_x}, \boldsymbol{q_y})$ é a *query*, e $\boldsymbol{T} = (\boldsymbol{t_x}, \boldsymbol{t_y})$ um ponto do *training data*:

$$5) \quad dist_{q \to t} = \sqrt{(q_x - t_x)^2 + (q_y - t_y)^2}$$

Uma vez calculadas todas as distâncias entre a *query* e os pontos do *training data*, é necessário ordenar de modo crescente. As **k** distâncias de menor valor, ou seja, os **k** pontos vizinhos mais próximos, classificarão a *query* enviada.

Verifique-se o funcionamento do KNN no diagrama de sequência da Figura 15.

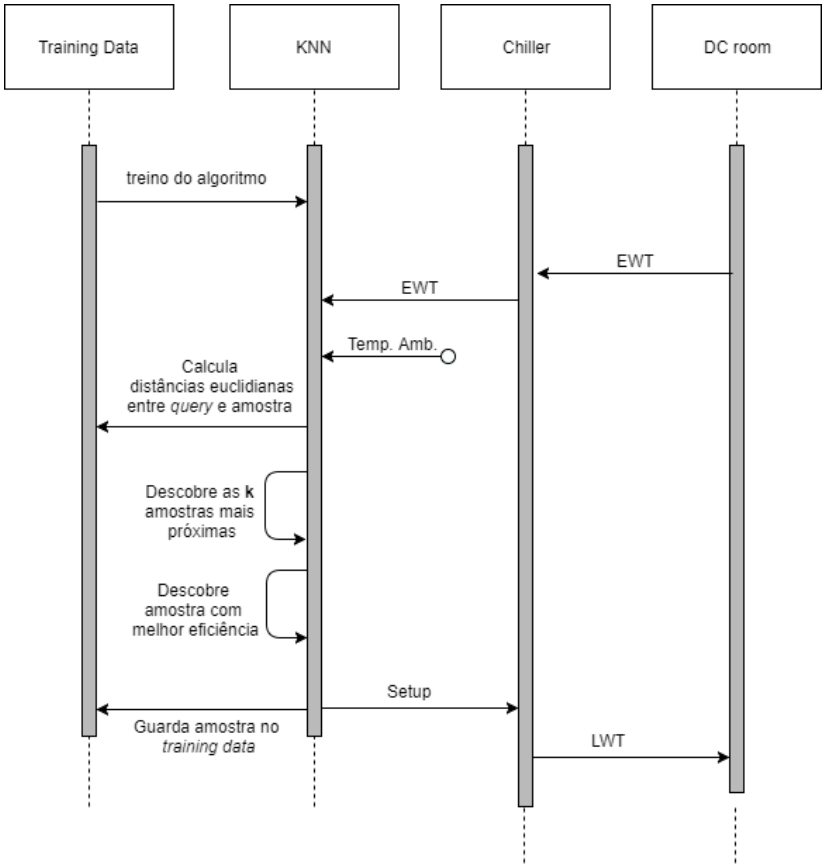


Figura 15 - Diagrama de sequência do funcionamento do KNN

Observando o diagrama de sequência, a primeira etapa será treinar o algoritmo. Uma vez treinado, espera até receber um *input* proveniente do *chiller* e a temperatura ambiente. Já com o *input*, começa o tratamento, calculando as distâncias euclidianas entre a *query* e as amostras e descobrindo as **k** amostras mais próximas. Descobertas as **k** amostras, o algoritmo procura qual dessas amostras classificará a *query* por ser a mais frequente dos vizinhos. Essa amostra irá

retornar o *setup* que enviará para o *chiller* e que inicia a refrigeração com os novos valores de LWT e Load.

Encontra-se também o diagrama de atividade (Figura 16) entre o KNN e o *training* data e a interação entre o algoritmo e o *chiller*, que resulta num novo *setup* mais eficiente para o *chiller* e consequentemente uma nova LWT a realizar para a sala do DC.

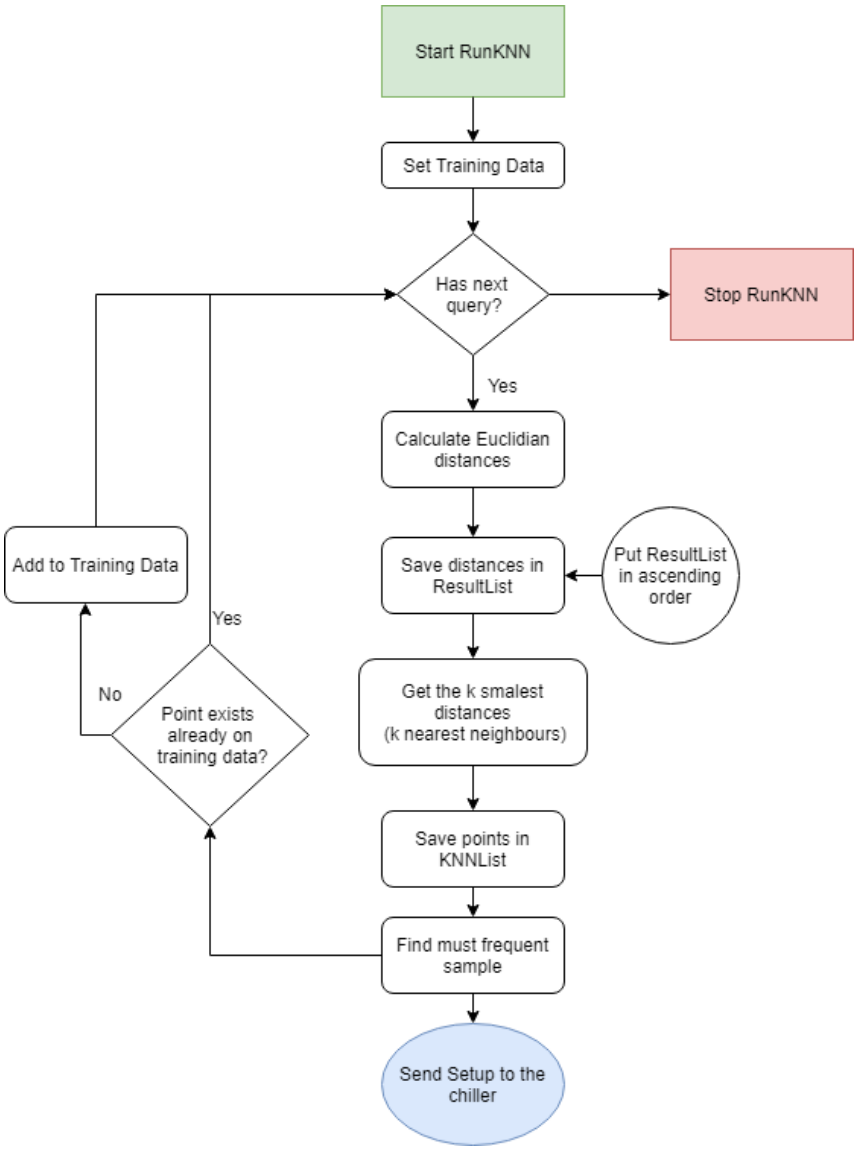


Figura 16 - Diagrama de atividade do KNN

3.2.5.2. *K-Means Clustering*

O *K-Means Clustering* tem como principal funcionalidade dividir o *training data* em grupos. Este algoritmo trabalha iterativamente sobre todas amostrar do *training data* para que sejam atribuídas a um grupo, grupos estes chamados de *clusters*. Neste caso, o valor **k** será o número de *clusters* que serão criados pelo algoritmo. As amostras são agrupadas com base na

similaridade dos recursos, ou seja, os pontos mais próximos são os mais parecidos uns com os outros, e estarão então no mesmo *cluster*. O centro de um *cluster* denomina-se *centroid* e é a partir deste ponto que se forma o *cluster*. Os resultados apresentados pelo funcionamento deste algoritmo são:

- O cálculo dos *centroids* de cada **k** *cluster;*
- Atribuição de cada amostra do *training data* a um e um só *cluster.*

Cada grupo, *cluster*, representa um valor, que será atribuído à *query* recebida. No caso de estudo, o grupo será representado pelo melhor valor de eficiência encontrado numa das amostras pertencentes a esse grupo e devolverá o *setup* dessa amostra ao *chiller*.

Para chegar à criação dos *clusters* e respetivos *centroids*, o algoritmo *K-Means Clustering* passa por processos iterativos até chegar à forma final. No início do seu funcionamento, começa por gerar *centroids* em posições aleatórias. Uma vez gerados os *centroids*, é possível criar os *clusters*. Para isto, é necessário percorrer todas as amostras existentes no *training data* e calcular a distância euclidiana entre as amostras e os *centroids*. Quando encontra o *centroid* mais próximo, essa amostra é atribuída ao *cluster* correspondente a esse *centroid* e assim sucessivamente até todas as amostras estarem todas atribuídas. Quando todas as amostras se encontrarem atribuídas, significa que os *clusters* foram todos formados.

Uma vez criados os *clusters*, o algoritmo inicia o seu processo iterativo de encontrar as posições perfeitas para os *centroids* de cada *cluster*. Durante uma iteração, o algoritmo executa várias etapas que se podem ver no esquema da Figura 17.

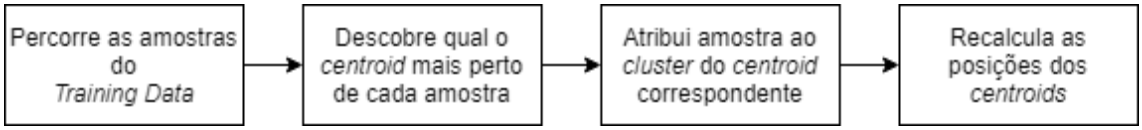


Figura 17 - Etapas de uma iteração

Antes de prosseguir para a próxima iteração, o algoritmo necessita de recalcular as posições dos *centroids*. Para isto, calcula a média das posições de todas as amostras presentes num *cluster* e atribui esse valor como sendo a nova posição do *centroid* desse mesmo *cluster*. Veja-se na Figura 18 um exemplo da reposição dos *centroids*.

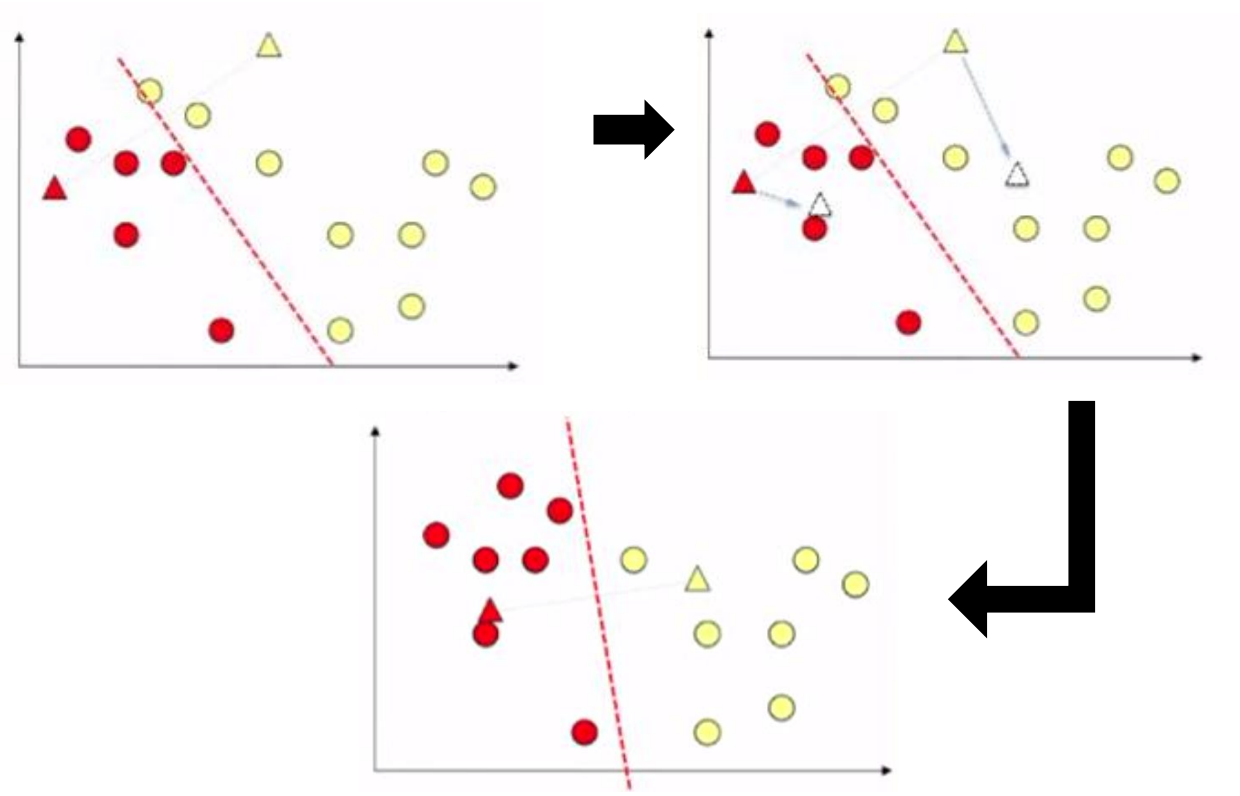


Figura 18 - Recálculo dos *centroids* na 2ª iteração

O exemplo apresentado na Figura 18 mostra um *training data* onde foram criados dois *centroids* (forma triangular, a vermelho e a amarela). Uma vez atribuídos os pontos a cada *centroid*, é recalculada a posição dos *centroids* como mostra a figura da direita. Depois de os *centroids* serem reposicionados é altura de voltar a atribuir as amostras aos *centroids* mais próximos (verifica-se na figura de baixo), e assim sucessivamente. O algoritmo deve iterar até as posições dos *centroids* convergirem, ou seja, a partir de uma certa iteração os *centroids* não irão mudar mais de posição. É responsabilidade do programador escolher o número de iterações mínimas para que as posições dos *centroids* convirjam.

Para determinar o número de *clusters*, ou seja, o valor de **k**, existem vários métodos comprovados. O método usado neste caso de estudo é bastante utilizado no cálculo do **k** devido às experiencias realizadas, no entanto carece de fundamento científico. É um método que pode ser aplicado para todo o tipo de *training data* (Kodinariya & Makwana, 2013). Devido à homogeneidade do *training data* usado neste caso, optou-se por utilizar este método:

$$6) \quad k \approx \sqrt{\frac{trainingData.size()}{2}}$$

Como se viu anteriormente, o *training data* possui 15214 amostras.

Então, tem-se que:

$$7) \quad k \approx \sqrt{\frac{15214}{2}} = 87.2 \approx \mathbf{87\ clusters}$$

Depois de todo este processo, cada *cluster* é caracterizado pela amostra dentro dele que possua o melhor valor de eficiência. Sendo assim, cada *cluster* será um *setup* diferente para o *chiller*. Concluindo, o algoritmo tem 87 *setups* diferentes para classifcarem uma *query*.

Assim, o algoritmo fica pronto a tratar de *queries*. Quando ocorre uma *query*, o algoritmo calcula as distâncias entre a *query* e todos os *centroids* até encontrar o mais próximo. O *cluster* referente a esse *centroid* irá caracterizar essa *query* e assim retornar o *setup* para o *chiller*. Esta *query*, depois de processada, é adicionada ao *training data* e inicia-se novamente o processo iterativo, para que o algoritmo atualize devido à nova amostra no *training data*.

Verifica-se a explicação no diagrama de sequência da Figura 19 e no diagrama de atividade da Figura 20.

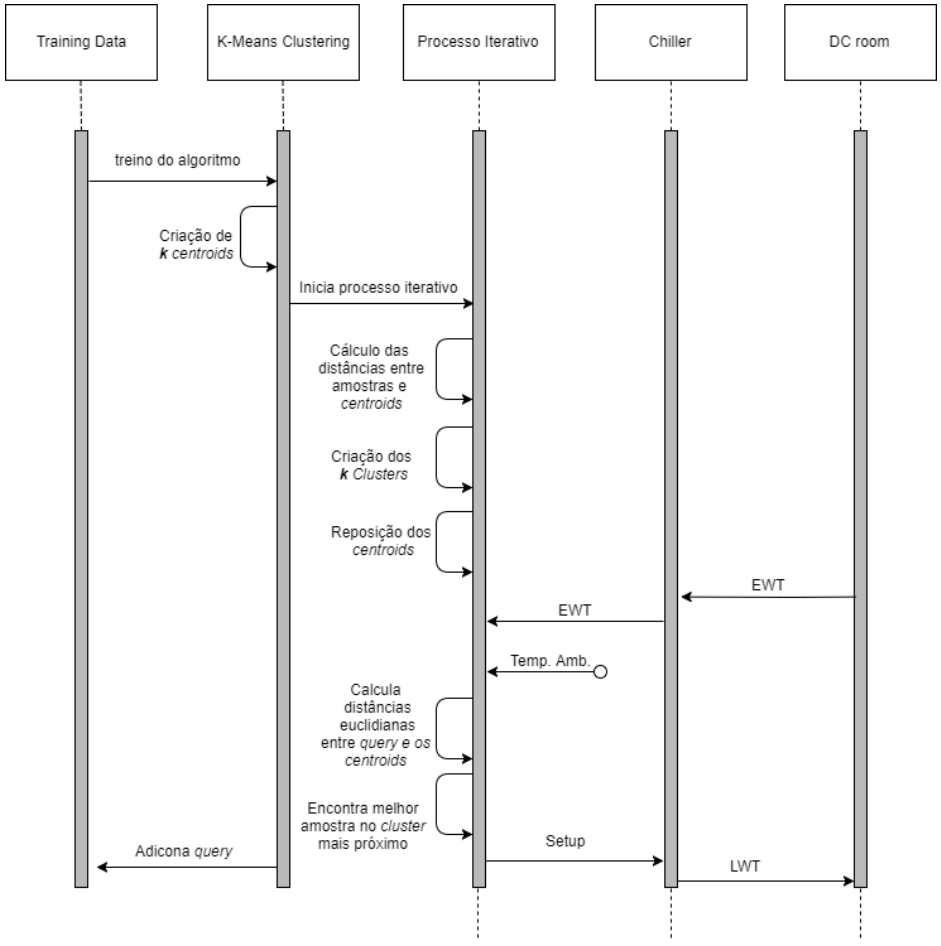


Figura 19 - Diagrama de sequência do algoritmo *K-Means Clustering*

Pela observação dos diagramas de sequência e de atividade é possível compreender o funcionamento do *K-Means Clustering* deste a sua criação até a resolução de uma *query*.

Como explicado anteriormente, o processo é iniciado pelo treino do algoritmo. Uma vez treinado, são criados *centroids* aleatoriamente no mapa do *training data*. Depois de criados os *centroids*, inicia-se o processo iterativo, que conta com várias etapas:

1. Cálculo das distâncias euclidianas entre as amostras e os *centroids* criados;
2. Formação dos *clusters*;
3. Reposição dos *centroids*
4. Repetir este processo o número de vezes escolhido para as iterações;
5. Receber *query*;
6. Calcular distâncias euclidianas entre a *query* e todos os *centroids*;
7. Encontrar o *centroid* mais próximo e classificar a *query* consoante a classificação do *cluster* correspondente a esse *centroid.*

Este processo pode ser verificado também no diagrama de atividade apresentado na Figura 20.

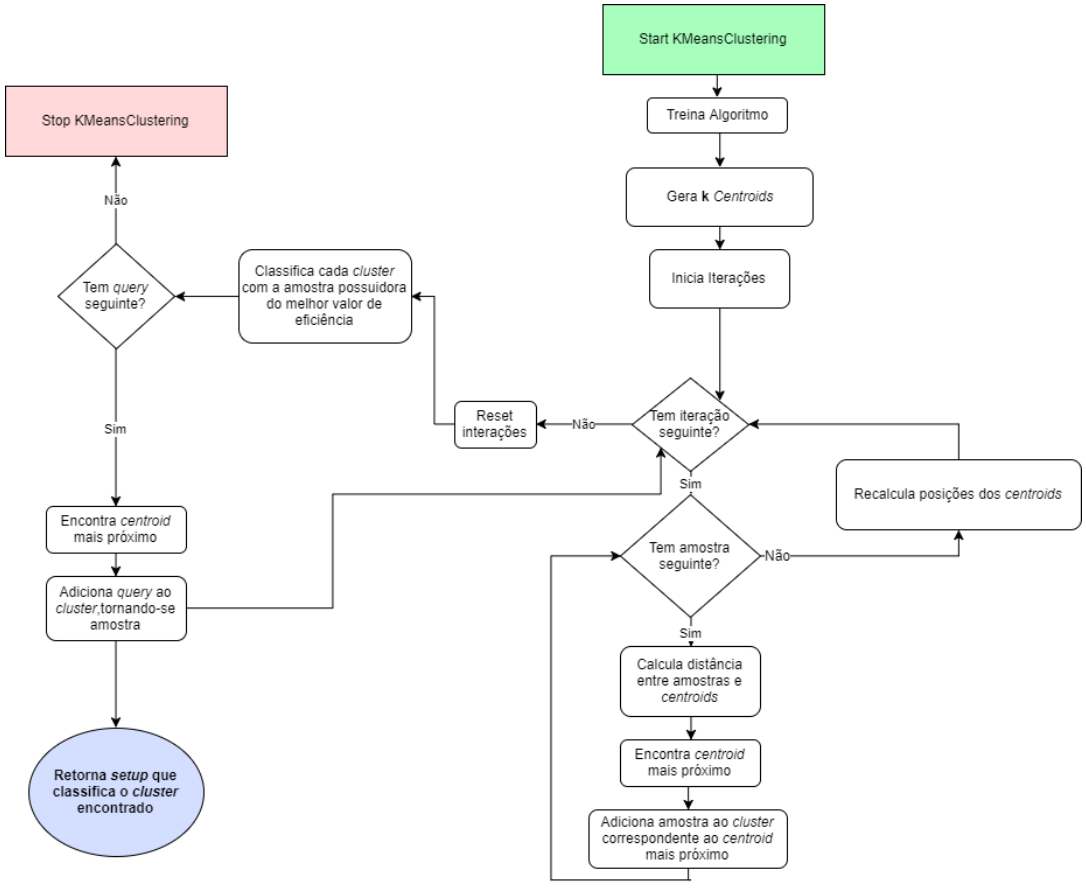


Figura 20 - Diagrama de atividade do algoritmo *K-Means Clustering*

No próximo capítulo é explicado como se procedeu à implementação dos diferentes algoritmos ML e como o processamento do *input* é realizado para que se obtenha o melhor valor de eficiência possível para cada caso. Exemplos matemáticos e gráficos do funcionamento de cada algoritmo serão mostrados para que se perceba o que foi explicado neste capítulo.

Serão também apresentadas as tecnologias utilizadas para a criação do simulador, bem como foram retirados os resultados.

4

Implementação

Nesta fase procedeu-se à implementação da simulação dos diferentes algoritmos de *machine learning*, com base no *training data* obtido. O objetivo desta simulação é que seja o mais realista possível, e que futuramente se possa converter para a realidade do problema apresentado no caso de estudo, de modo a que os DCs utilizem a energia da melhor maneira possível.

Para o desenvolvimento dos algoritmos foi utilizada como linguagem de programação o *java* num ambiente de desenvolvimento integrado gratuito, o Netbeans. Criou-se diferentes grupos que compõe a implementação – a simulação, *training data* e *machine learning*.

Neste capítulo é possível encontrar a descrição de cada grupo, as suas funcionalidades, classes, métodos e a explicação de cada algoritmo escolhido, bem como o porquê de ter sido escolhido para este caso de estudo. Nem todos os algoritmos ML existentes encaixariam neste problema, sendo assim há necessidade de se compreender o tipo de algoritmo que se procurou para este caso.

4.1. Estrutura da Implementação

O projeto contem três grupos de funcionamento – *training data*, *machine learning* e o simulador. Dentro destes grupos existem classes que tornam possível a comunicação entres eles. Na Figura 21 é possível verificar-se as diferentes classes e relações existentes.

Tudo se inicia no grupo "Simulador", que tem como primeira etapa a leitura para memória dos dados que irão constituir o *training data*. Esta etapa é realizada pelo *TOPSSdata()*, classe esta que lê ficheiros de extensão *.csv* de determinada maneira com o objetivo de criar uma lista de pontos do tipo *InputData*. Este tipo de dados foi criado para ser possível armazenar todos os dados com as informações necessárias no *training data*. É também possível verificar graficamente estes pontos através da classe *plotTrainingData()*.

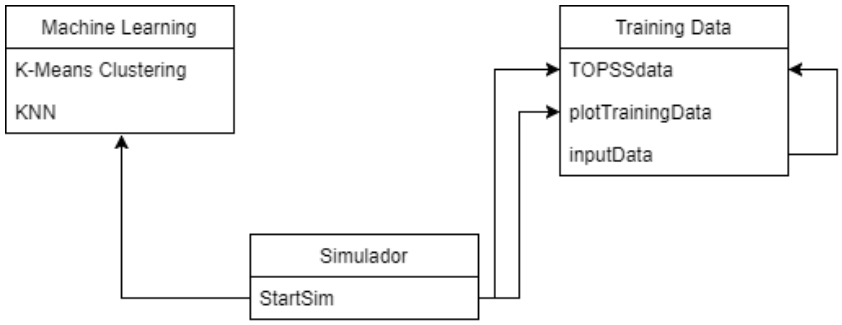


Figura 21 - Diferentes grupos da implementação e relações entre eles

Depois da leitura dos dados estar completa, segue-se a escolha do algoritmo ML. Como proposta para a resolução do problema deste caso de estudo, foram desenvolvidos dois algoritmos ML supervisionados – *K Means Clustering* e KNN (*K-Nearest Neighbors*). De notar que ambos os algoritmos implementados utilizam a mesmo *training data*, com o propósito de serem comparados de igual modo. Nos subcapítulos que se seguem será explicado o funcionamento de cada algoritmo e como a sua implementação foi realizada.

4.1.1. *Training Data*

Neste grupo pode-se encontrar três classes- *TOPSSdata*, *plotTrainingData* e *InputData*. A classe *TOPSSdata* tem como objetivo ler todos os ficheiros com extensão *.csv* criados no TOPSS e criar assim uma lista com todos os valores necessários. Para a criação desta lista, que no fundo será o transportador do *training data*, foi implementada a classe *InputData*, neste grupo, para criar um tipo de dados próprio do *training data*. A classe *InputData* foi criado essencialmente com o objetivo de simplificar o processamento de dados realizado na execução dos algoritmos ML.

Segue-se o diagrama de classes referente à classe do *training data*, Figura 22.

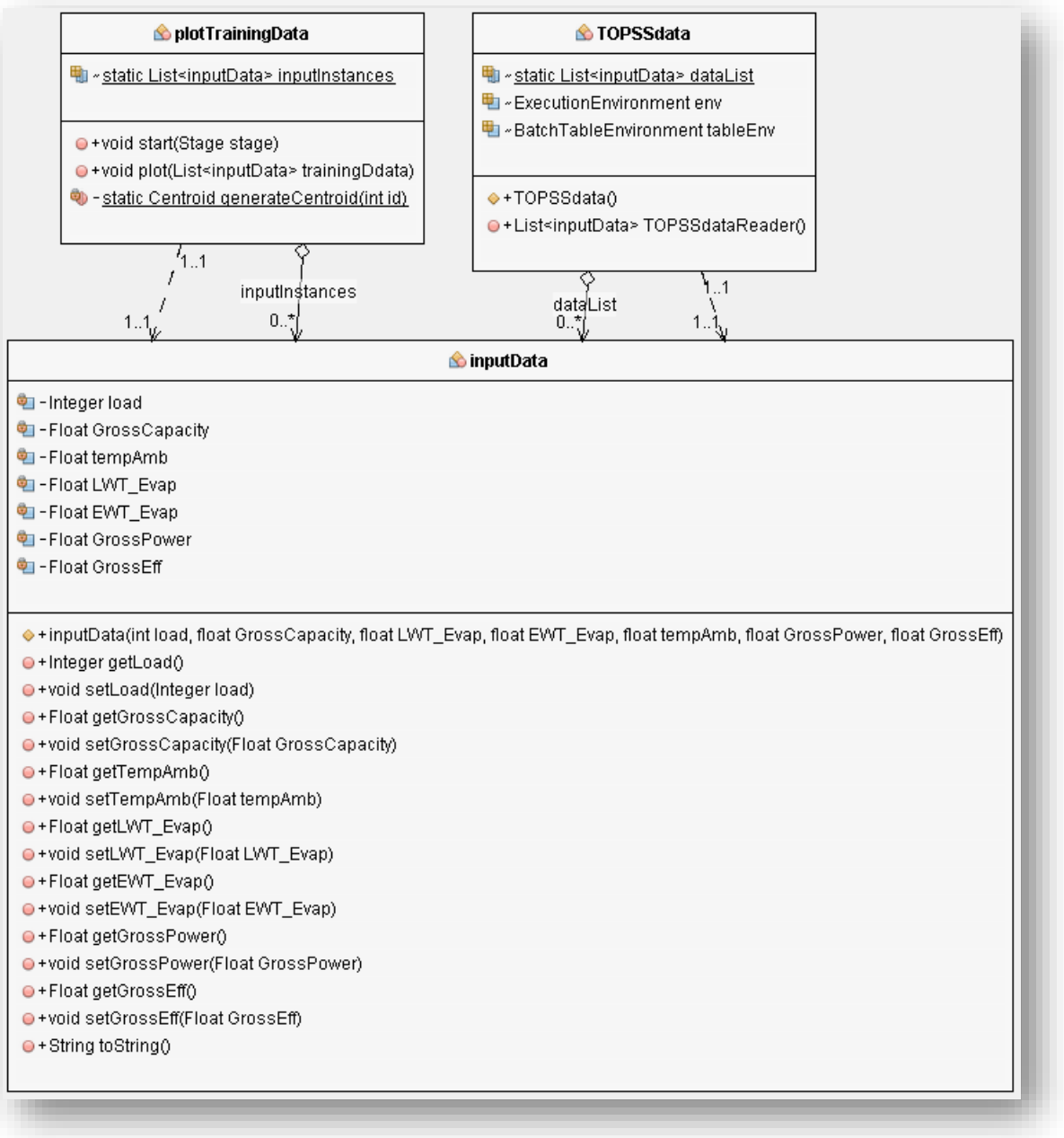


Figura 22 - Diagrama de classes do *training data*

Na Tabela 3 são explicados os métodos de cada classe, bem como a apresentação das variáveis utilizadas em cada uma.

Tabela 3 - Descrição das propriedades do Grupo *TrainingData*

Classe	Variáveis	Métodos
TOPSSdata	*List<InputData> dataList*	TOPSSdataReader () -Lê todos os ficheiros .csv produzidos pelo *software* TOPSS com o objetivo de produzir a lista *dataList* do tipo *InputData*. Retorna esta mesma lista para o Simulador.
PlotTrainingData (*Java Application*)	*List<InputData> dataList*	Plot(List<inputData> trainingData) – Recebe como parâmetro uma lista do tipo InputData, executa uma aplicação em java. Abre janela com gráfico do *training data*.
InputData	*int Load* *float GrossCapacity* *float tempAmb* *float LWT_Evap* *float EWT_Evap* *float GrossPower* *float GrossEff*	É criado o objeto *InputData*. O *training data* é uma lista com o tipo deste objeto. As variáveis existentes nesta classe são os parâmetros que o *software* TOPSS apresenta. Criados os métodos *get* e *set* para todas as variáveis do objeto

De seguida serão explicadas detalhadamente cada classe e os respetivos métodos criados, bem como as relações com as classes dos outros grupos.

4.1.1.1. Classe TOPSSdata

Esta classe foi desenvolvida para ler os ficheiros gerados pelo TOPSS, como já foi referido. Para isto, foi necessário criar uma pasta no local onde se encontra o projeto com todos os ficheiros *.csv* recolhidos do TOPSS, e sempre que o simulador inicia, o método *TOPSSdataReader* é chamado. Este método abre os ficheiros *.csv* e utiliza uma variável do tipo *Scanner*, ferramenta do *java*, para realizar a leitura do ficheiro. Esta ferramenta funciona como um *handler*, que lê linha a linha o ficheiro *.csv*.

Para se ter uma ideia de como um ficheiro *.csv* é gerado no TOPSS, pode-se verificar a Figura 10 no Capítulo 3. No entanto, um ficheiro *.csv* verdadeiramente separa cada elemento por uma vírgula. Ora veja-se na Figura 23:

```
Load, Gross Cap., LWT Evap, EWT Evap, Flow Evap, WPD Evap, Ambient, Gross Power, Gross
Eff., Net Cap., Net Power, NetEff,
%, kW, C, C, L/s, kPa, C, kW, EER (kW/kW), kW, kW, EER (kW/kW),
100,7161,160,210,159,245,240,1782,565,7157,1778,563,
90,6659,160,198,159,245,240,1722,480,6655,1718,478,
80,6407,160,186,159,245,240,1402,519,6403,1398,517,
70,5684,160,174,159,245,240,1585,433,5680,1581,431,
60,4386,160,162,159,245,240,1448,533,4382,1444,531,
50,4477,160,150,159,245,240,1314,529,4473,1310,527,
40,2898,160,138,159,245,240,1237,562,2894,1233,560,
30,2783,160,126,159,245,240,1282,600,2779,1278,598,
```

Figura 23 - Exemplo de um ficheiro ".csv"

Esta classe abre para leitura o ficheiro e de seguida utiliza a variável *Scanner* para ler linha a linha, onde cada elemento de cada linha é separado por vírgula. É então realizado a divisão de cada elemento e colocado na lista *DataList*, que será retornada para o Simulador com o fim de treinar os algoritmos ML. Note-se que o ficheiro gerado no TOPSS não traz os valores com vírgulas, ou seja, com parte decimal, foi necessário um tratamento especial para guardar estes dados em memíra. A lista *DataList* é do tipo *InputData*, então em cada posição pode-se encontrar uma amostra com os valores de *load, GrossCapacity, tempAmb, LWT_Evap, EWT_Evap, GrossPower*, valores estes que foram explicados nos capítulos anteriores (É possível encontrar um exemplo da lista na Figura 30).

Em cada posição da lista *DataList* é guardada uma amostra do tipo *InputData*, que será explicada no capítulo 4.1.1.3. Esta lista de amostras será a base de toda a implementação realizada, a *training data* final que irá treinar os algoritmos ML implementados.

De seguida é possível observar o diagrama de actividade (Figura 24) referente a esta classe, observando-se melhor como se processa a leitura dos ficheiros. Na Figura 25 observa-se o diagrama de sequência da leitura de uma linha num ficheiro *.csv*, realizada pela classe *TOPSSdataReader*.

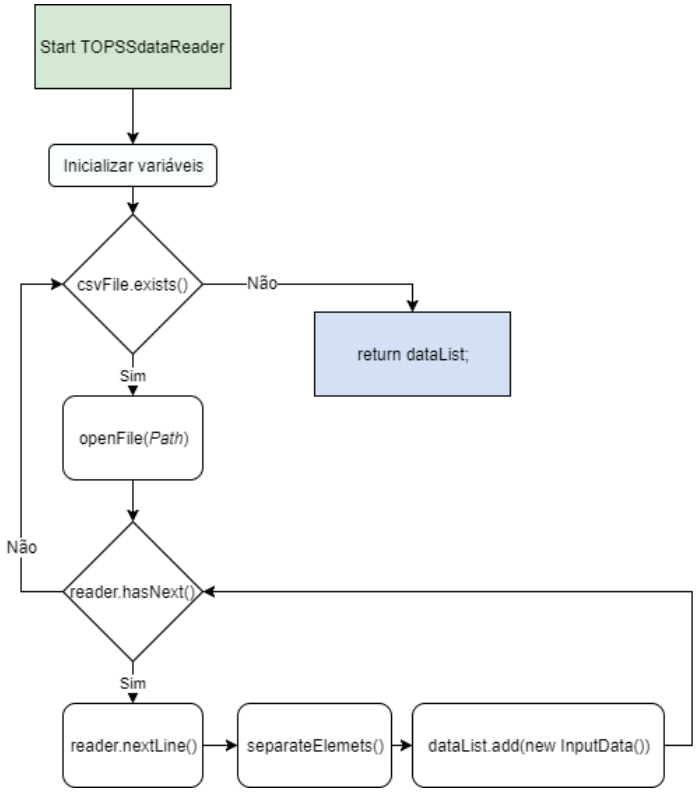


Figura 24 - Diagrama de atividade da recolha do *training data*

Quando o simulador chama o método *TOSSdataReader* da classe TOPSSdata, inicia-se o processo de leitura dos ficheiros *.csv* extraídos do *software* TOPSS. Depois de inicializar as variáveis necessárias, o método verifica se existe o ficheiro pretendido. Caso exista, irá abri-lo para leitura e começar a ler linha a linha. Enquanto o ficheiro tiver linhas, o método irá lendo e separando os elementos importantes, como o *load*, eficiência, LWT, EWT e temperatura ambiente. Separados os elementos, adiciona à *dataList*, que como referido anteriormente, será o *training data*.

Este processo terminará quando não existir mais ficheiros *.csv* para ler. Quando isto acontecer, a lista *dataList* é retornada para o simulador de modo a que possa ser usada pelos algoritmos ML no seu treino.

Verifica-se todo este processo descrito no diagrama de sequência apresentado na Figura 25, onde é possível verificar o modo de como uma linha é lida pelo método e adicionada à *dataList*.

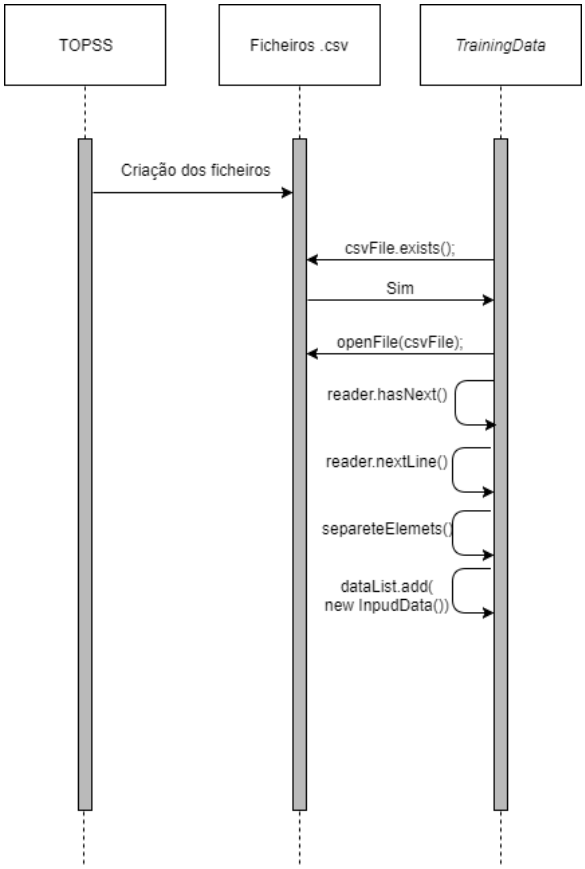


Figura 25 - Diagrama de sequência de como é tratada a leitura de uma linha

4.1.1.2. Classe *PlotTrainingData*

Esta classe tem como finalidade criar um gráfico de dispersão do *training data*, que é carregado através de uma janela *Java Application*. O método *Plot* recebe como parâmetro uma lista do tipo *InputData*. Esta lista é tratada para que seja apresentado um gráfico da *EWT* em função da temperatura ambiente, os principais *inputs* que o algoritmo ML irá utilizar.

Para a obtenção do gráfico foram utilizadas nomeadamente duas bibliotecas do *java*, *ScatterChart* (gráficos de dispersão) e *XYChart*. Uma vez com as propriedades da janela certas, é possível obter o seguinte gráfico da Figura 26.

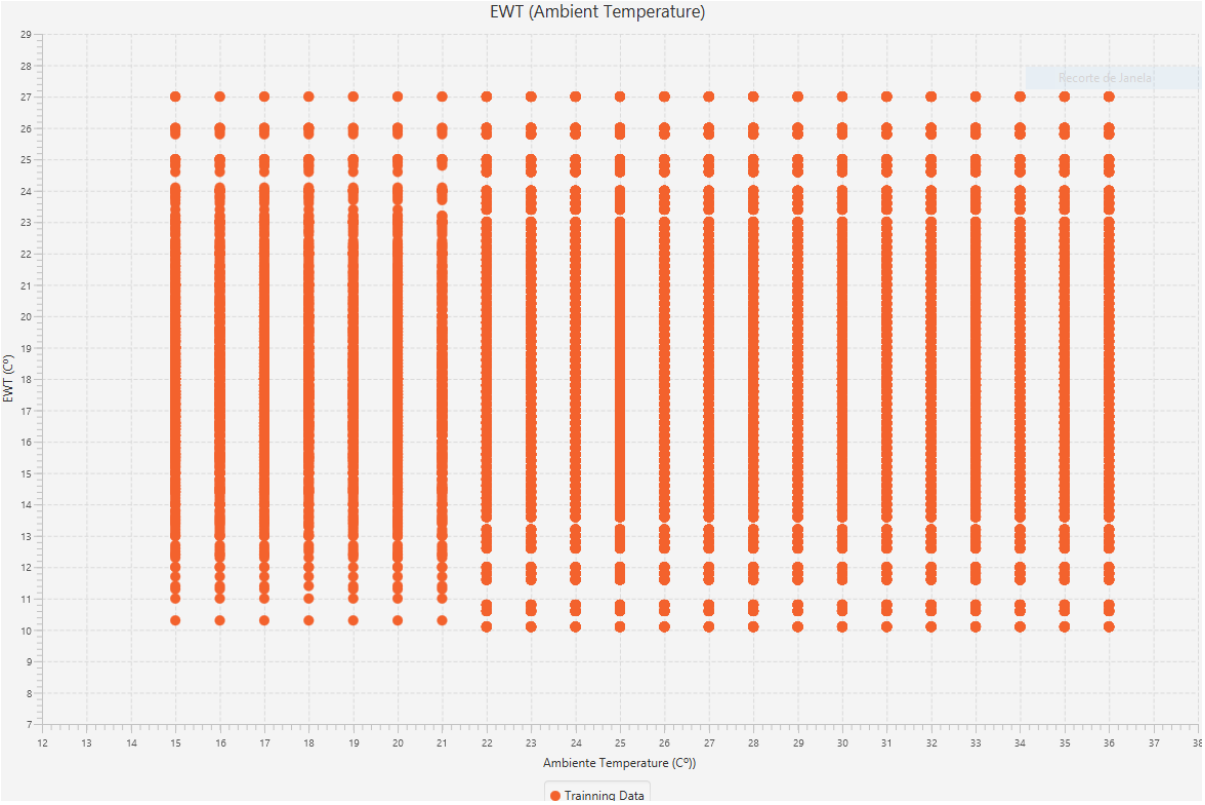


Figura 26 - Gráfico de dispersão do *training data*

Ao fazer-se uma ampliação, é possível verificar melhor os pontos:

Figura 27 - Gráfico de dispersão ampliado

Cada ponto destes representa uma variável do tipo *InputData*. Está aqui apresentado o *training data* usado para testar os diferentes algoritmos ML implementados.

Apresenta-se o diagrama de atividade (Figura 28) desta classe para compreender melhor como se obtém graficamente o *training data* utilizado. O diagrama de sequência também apresentado na Figura 29.

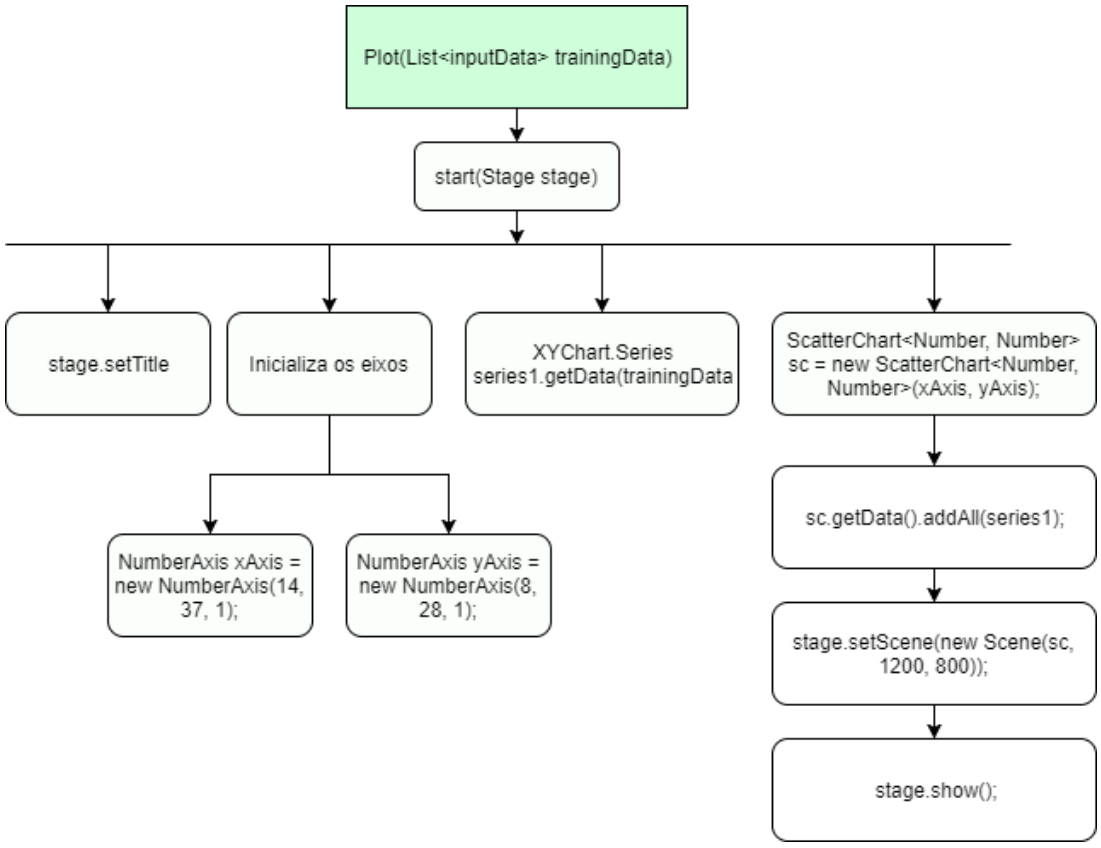


Figura 28 - Diagrama de atividade da classe *PlotTrainingData*

Para poder visualizar o *training data* graficamente, o simulador tem a possibilidade de lançar o método *Plot* da classe *PlotTrainingData*. Este método realiza vários processos que dão origem a uma janela *Java Application*. Esses processos passam por:

1. *setTitle* – atribuir um título à janela;
2. Inicializar os eixos – as variáveis *xAxis* e *yAxis* são criadas com os intervalos dos gráficos e a medida que apresenta. Por exemplo, no caso do eixo do *x*, definiu-se que a janela irá de 14 a 37, de 1 em 1 ponto;
3. *series1.getData(trainingData)* – a lista *trainingData* é inserida na variável series1 do tipo *XYChart.Series*, tipo este especifico para gráficos de dispersão em *java;*
4. Criação do gráfico pela variável *ScatterChart sc* – Define o tamanho da janela e realiza o gráfico pela função *stage.show().*

Verifica-se este processo no diagrama de sequência da Figura 29.

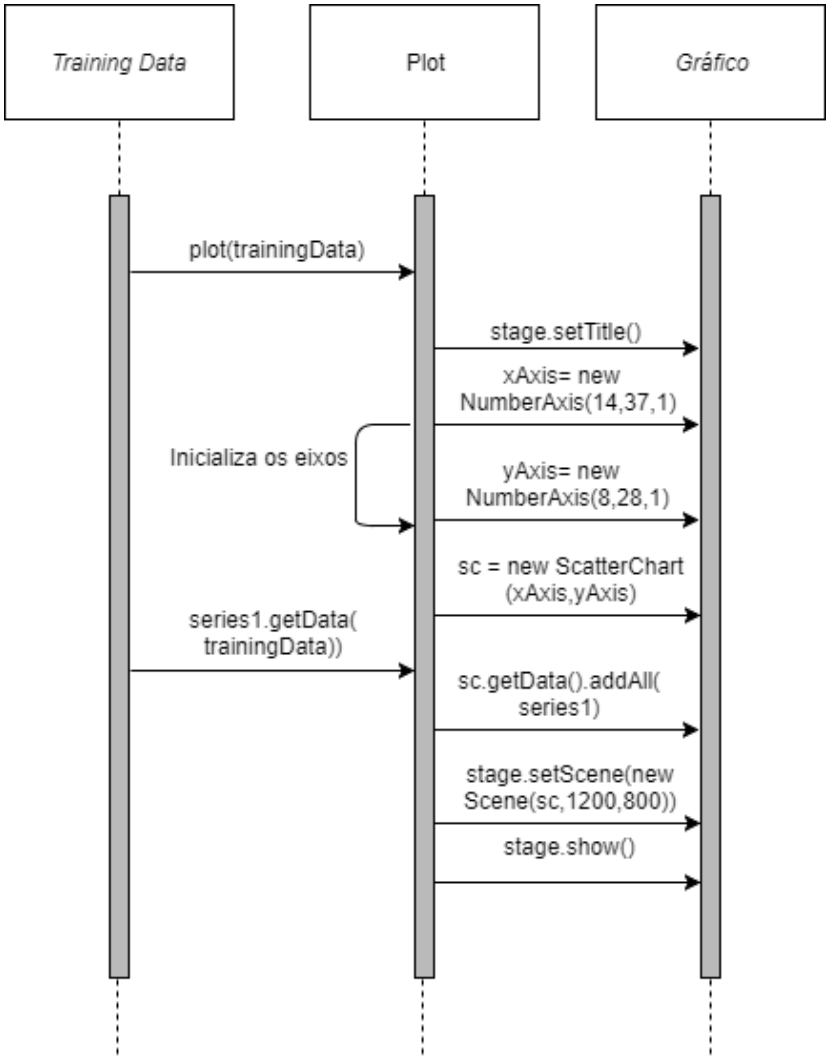


Figura 29 - Diagrama de sequência do PlotTrainingData

4.1.1.3. Classe *InputData*

Esta classe foi implementada para criar um tipo único de dados referentes aos recolhidos no TOPSS. Como foi mencionado, foi criada uma lista com o tipo *InputData* para guardar todo o *training data* a ser usado nos algoritmos. Isto significa que, para cada posição da lista, existe guardado um ponto que possui a informação do tipo *InputData*, ou seja: *load, GrossCapacity, tempAmb, LWT_Evap, EWT_Evap, GrossPower*. Verifica-se no seguinte esquema (Figura 30), um exemplo de como a informação é guardada na lista.

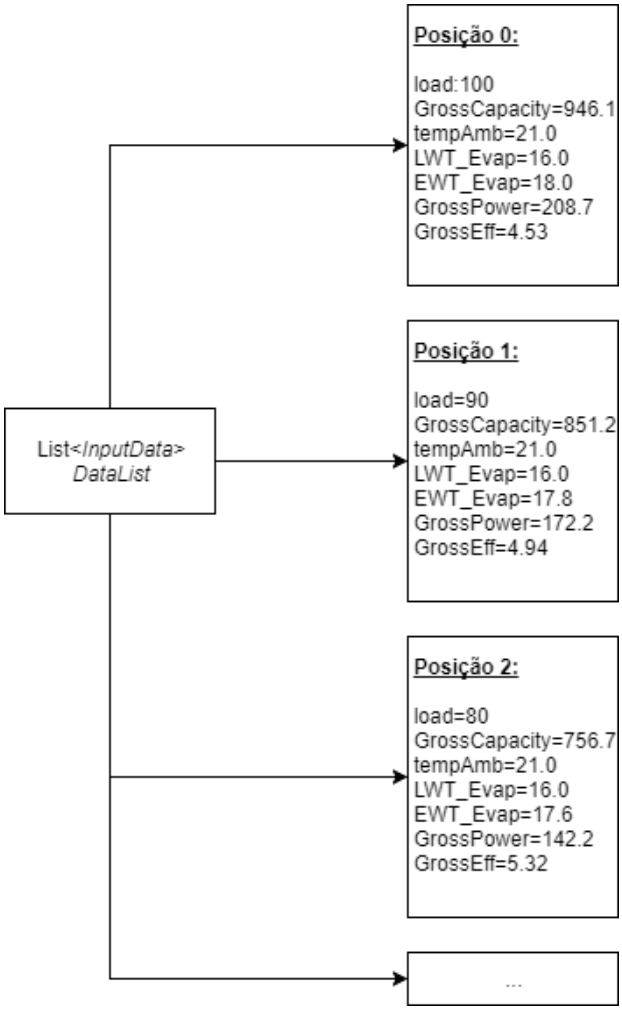


Figura 30 - Exemplo de como a informação é guardada na lista

Como se observa, este vetor do tipo *InputData* funciona como que de uma estrutura se tratasse. Todas estas posições são facilmente acedíveis, bem como a informação de cada elemento. Os pontos que se verificam no gráfico da Figura 26 são posições da lista com o tipo de dados *InputData.* Nesta classe são criados métodos que retornam o valor de cada elemento e métodos que possam alterar os valores. Todos estes métodos criados podem ser utilizados para todas as variáveis do tipo *InputData* existentes nas diferentes classes.

4.1.2. Algoritmos *Machine Learning*

Uma vez explicado o procedimento da recolha do *training data* e de como é possível visualiza-la através de um gráfico de dispersão, é explicado neste capítulo os algoritmos implementados. Para este caso desenvolveu-se dois algoritmos ML, o KNN (K-Nearest Neighbours) e o *K-Means Clustering*. Estes algoritmos têm modos de funcionar diferentes que irão ser explicados, mas com o mesmo o objetivo, aumentar a eficiência energética dos *chillers.*

É de seguida apresentado o diagrama de classes (Figura 31) bem como a explicação de cada classe e método na

Tabela 4 .

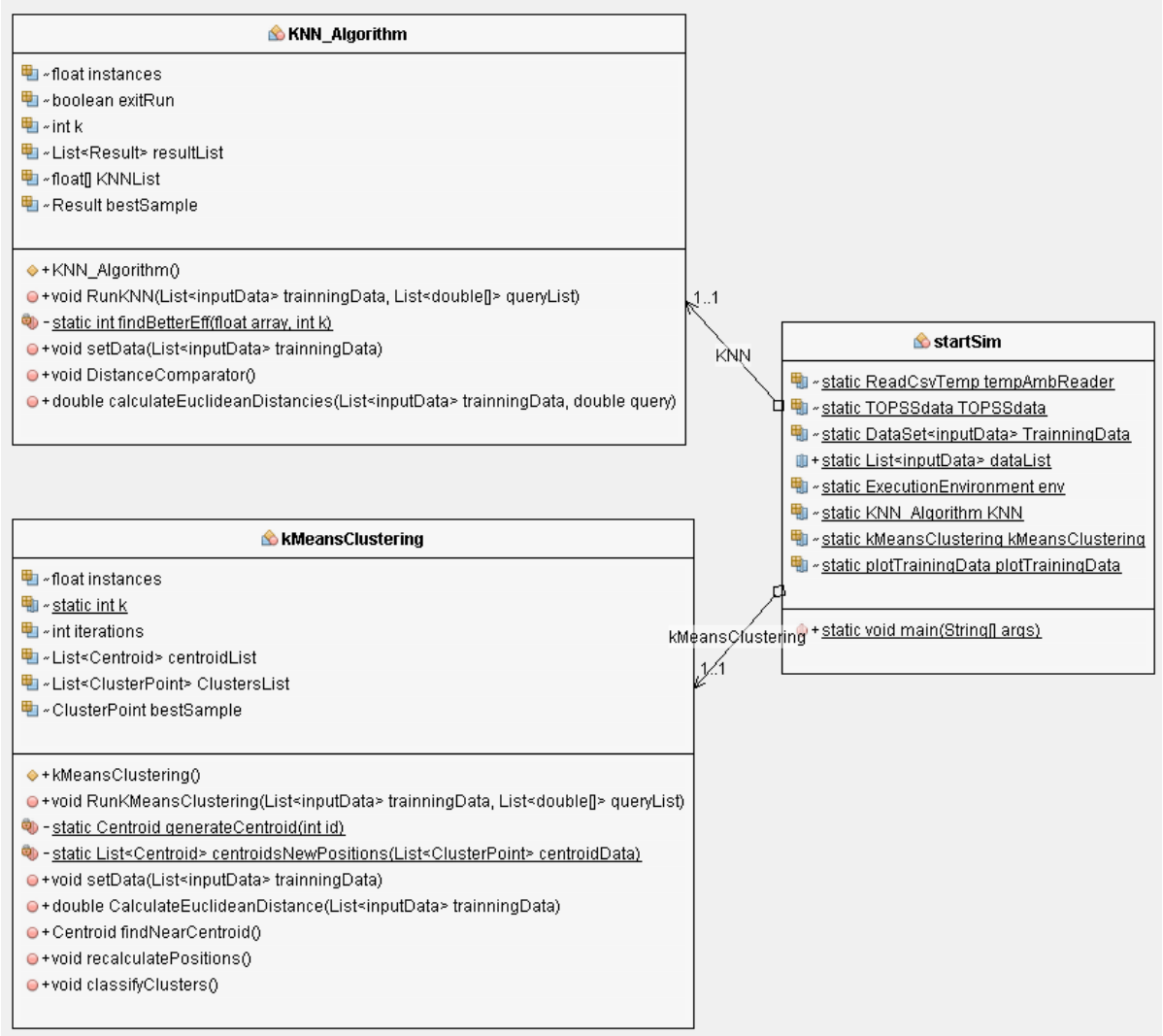


Figura 31 - Diagrama de classes do grupo Machine Learning

Tabela 4 - Descrição das propriedades do grupo Machine Learning

Classe	Variáveis	Métodos
KNN_Algorithm	• *float Instances [] []* • *List<Result> resultList = new ArrayList<Result> ();* • *Int k;*	• RunKNN(List<inputData> trainingData, List<double[]> queryList)– Recebe como parâmetros o *training data* e a lista de *queries* para processar. Inicia o funcionamento da

	• *float [] KNNList;* • *Result bestSample;*	*machine learning*, treinando o algoritmo com o *training data* e de seguida trata da lista de *queries*. • FindBetterEff (flat array [], int k) – Percorre todos os vizinhos de forma a encontrar a *sample* mais frequente. Retorna a posição de uma dessas *samples* que classificará a *query*; • DistanceComparator – Compara distâncias, com o objetivo de ordena-las por ordem crescente; • setData(List<inputData> trainningData) – coloca as variáveis *input* dentro da variável *Instances*. • calculateEuclideanDistancies – calcula as distâncias euclidianas entre a *query* e todos os pontos do *training data*. Guarda resultados na lista *resultList*.
kMeansClustering	• *float Instances [] []* • *int k;* • *int iterations*; • *List<Centroid> centroidList* • *List<ClusterPoint> ClustersList;* • *ClusterPoint* bestSample	• RunKMeansClustering(List<inputData> trainningData, List<double[]> queryList)– Recebe como parâmetros o *training data* e a lista de *queries* para processar. Inicia o algoritmo que consiste em duas partes: treino e processamento de *queries*; • generateCentroids(int id) – gera uma posição random X (entre 36 e 15) e uma posição random Y (28 a 10), cria o *centroid* com essas posições e retorna-o. • centroidsNewPositions (List<ClusterPoint> centroidData) – Recebe como parâmetro a lista de pontos organizado por *clusters*. Calcula a média das posições dos pontos dentro de cada *cluster*. Atribui essas médias às novas posições dos *centroids* de cada *cluster*.

4.1.2.1. K-Nearest Neighbours (KNN)

De modo a que seja mais explicito a explicação apresentada no capítulo da arquitetura, será mostrado um exemplo do algoritmo implementado a funcionar com o *training data* mostrado na Figura 26. Desde modo, será também uma maneira mais percetível de como a implementação deste tipo de algoritmos poderão ajudar no problema energético dos *chillers*.

Na solução implementada, o simulador aciona o funcionamento do algoritmo KNN, chamando o método *RunKNN* que recebe como parâmetro a *DataList* (lista com os pontos do *training data*). Este método começa por preparar o *training data*, como é mostrado na Figura 26.

De seguida, é esperado que seja enviado uma *query*. Esta *query* será um *input*, ou seja, um valor de EWT e um valor de temperatura ambiente como é possível verificar na Figura 32.

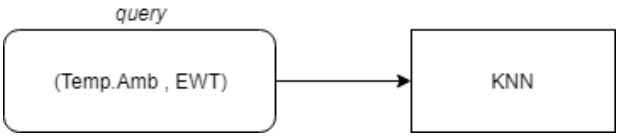


Figura 32 - KNN recebe uma *query* com um valor de temperatura ambiente e um EWT

Esta *query* funciona como um ponto num espaço bidimensional, assim como os pontos existes no *training data* usada. Ora veja-se novamente na figura seguinte (Figura 33) o gráfico de dispersão do *training data* desta vez com a *query* representada. Para este exemplo, admitiu-se que a *query* tinha os seguintes valores:

- Temperatura ambiente = 27.5 °C;
- EWT = 17 °C.

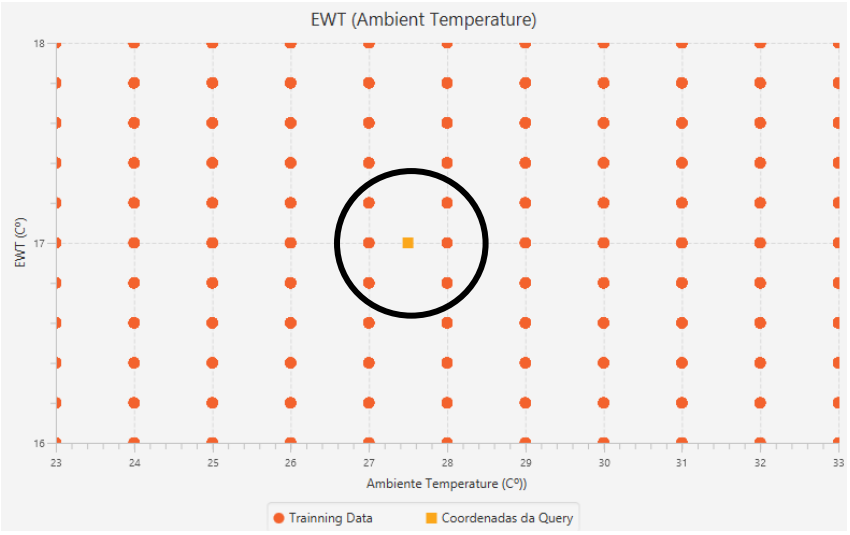


Figura 33 - Gráfico de dispersão do *training data* com a representação de uma *query*

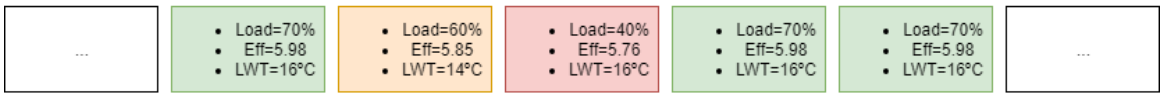
E assim é representada a *query* no meio do *training data* obtido. Recorde-se que cada ponto do *training data* é do tipo *InputData*, que possui valores para um determinado *setup* para os *chillers* consoante os valores de entrada (temperatura ambiente e EWT). Recebida a *query*, é calculada a distancia euclidiana entre esta e todos os pontos presentes no *training data*. Todas as distâncias euclidianas calculadas são guardadas na lista *ResultList*, que é do tipo de dados *Result*,

criado na classe do algoritmo. Este tipo de dados basicamente guarda o ponto do *training data* ao qual foi calculada a distância entre ele e a *query* e essa mesma distância.

Já com os cálculos efetuados e guardados na *ResultList*, foi criado o método *DistanceComparator*, que ordena de ordem crescente esta lista, em função das distâncias calculadas. Isto servirá para que se descubra quais os **k** vizinhos mais próximos. Como o *training data* possui 15214 amostras, existirá 15214 distâncias euclidianas calculadas. Uma vez em ordem crescente, retiram as **k** primeiras distâncias, ou seja, as 124 distâncias que correspondem aos pontos mais próximos da *query*. Note-se que, nesta fase, o algoritmo já filtrou para 124 as 15214 hipóteses que podem solucionar a *query*. Estas hipóteses são guardadas no vetor *KNNList*.

Neste momento, o algoritmo utiliza o método *findBetterEff()*. Este método irá percorrer o vetor *KNNList*, ou seja, as 124 hipóteses encontradas e descobrir qual a amostra mais frequente. A amostra mais frequência classificará a *query*.

Figura 34 - Esquema simplificado da lista *KNNList*, apresentando a melhor hipótese



Como se verifica, muito simplificado, na Figura 34, o método *findBetterEff* percorreu as posições do vetor KNNList encontrando o melhor valor de *Eff* (eficiência). Esta amostra será o *setup* que o KNN retorna, ou seja, a solução da *query* recebida. Neste caso em concreto, o *setup* devolvido será um *load* de 70% com uma LWT de 16ºC. Em caso de empate, ou seja, caso exista dois ou mais *setups* com o mesmo número de votos, o algoritmo decrementa o valor de **k** até esse empate ficar desfeito. Este decremento significa eliminar a última posição da lista *KNNList*. Uma vez que os valores se encontram ordenados por distância, a última posição contém a *sample* mais distante da *query*.

Transpondo para um caso real, a *query* recebida é enviada pelo *chiller*, a EWT que existem no momento, 27.5 °C e o algoritmo obtém a temperatura ambiente atual, 17 °C. O *chiller* envia esta *query* para o algoritmo KNN, que executa todo o processo explicado até agora. O KNN devolve um *setup*, que neste caso encontra-se representado na Figura 34:

$$Setup = \begin{cases} Load = 70\% \\ LWT = 16{,}0\ ^{\circ}\mathrm{C} \end{cases}$$

Uma vez devolvido o *setup*, o algoritmo adiciona o novo ponto ao *training data*, para que este "aprenda" que, para o determinado *input* recebido, o *setup* será este que foi encontrado. Adiciona-se então o seguinte ponto ao *training data*:

$$Nova\ amostra = \begin{cases} EWT = 17°C \\ Temp.Amb. = 27{,}5°C \\ Load = 70\% \\ LWT = 16{,}0°C \\ Eff. = 5{,}98 \end{cases}$$

Termina aqui o processamento da *query*. Num ambiente real, as *query* estarão sempre a entrar no algoritmo, que estará sempre em funcionamento colocando o *chiller* a atuar sempre da maneira mais eficiente.

Para visualizar melhor o funcionamento do algoritmo apresenta-se o seguinte diagrama de atividade na Figura 35.

No diagrama de atividade representado verifica-se a explicação anterior. O algoritmo começa por ser treinado. De seguida, verifica se existe uma próxima *query* (*query.hasNext()*) a tratar, caso exista, o KNN inicia o seu funcionamento. Através do método *calculateEuclideanDistancies()* o KNN irá calcula a distância euclidiana entre a *query* e todas as amostras presentes no *trainingdata*. Cada distância será guardada na lista *resultList*, juntamente com a respetiva amostra. Para ordenar estas distâncias na ordem crescente é utilizada uma ferramenta de *sorting* existente no *java*. Através desta ferramenta e do método *DistanceComparator()*, a lista *resultList* ficará na ordem crescente em relação às distâncias euclidianas. Com isto, é necessário retirar as **k** samples mais próximas da *query* e encontrar a amostra mais frequente das 124 amostras vizinahs, com o método *findBetterSample()*. Depois deste processo, é retornado essa *bestSample*, que será o *setup* para o *chiller*.

Para terminar, verifica-se se existe esta *query* no *training data*, caso exista, o algoritmo irá verificar se há mais *queries* a processar, caso não exista, é adicionada como uma nova *sample* e o KNN continua o seu processo de tratamento de *queries*.

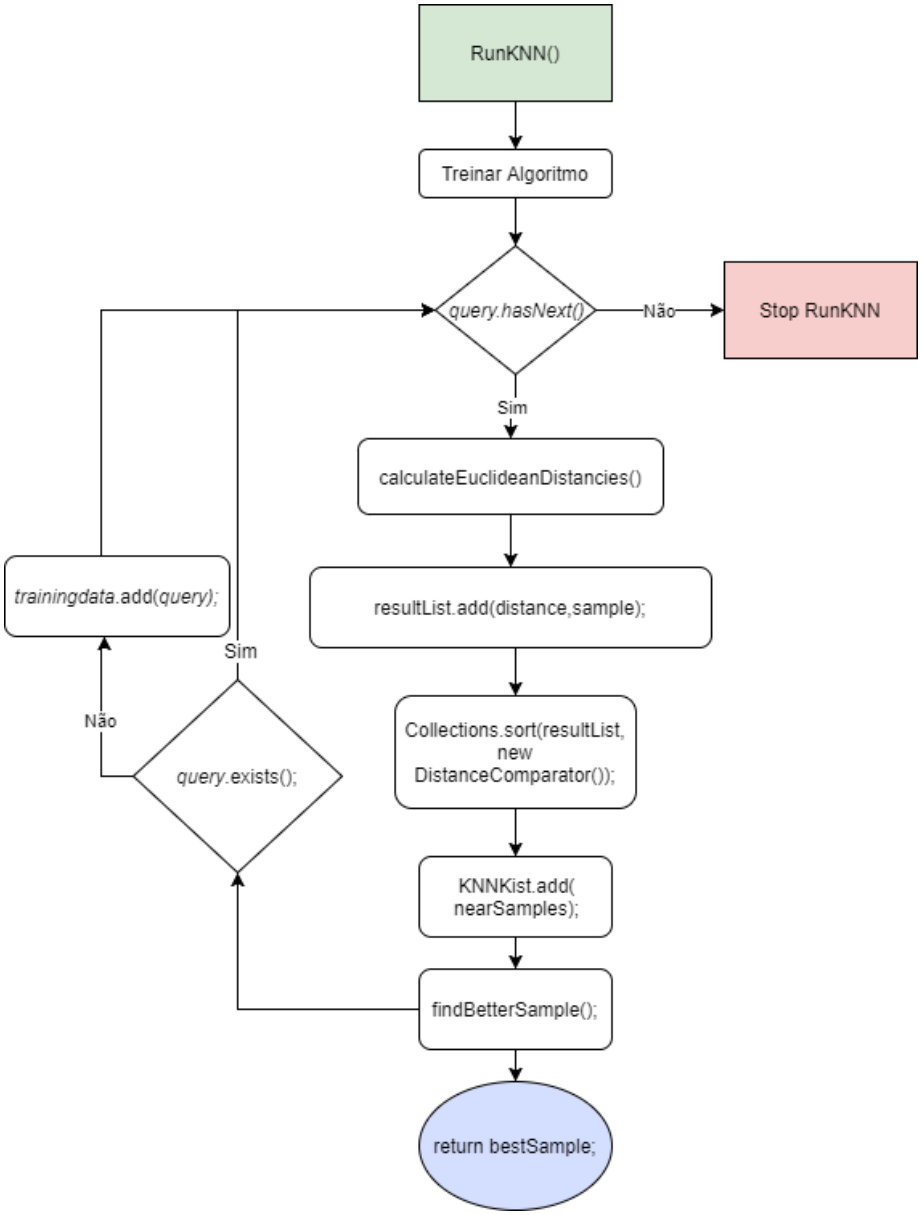


Figura 35 - Diagrama de atividade do algoritmo KNN

O diagrama de sequência na Figura 36 é apresentado a seguir, de modo a ser mais percetível as interações realizadas pelo KNN até chegar ao *setup* desejado.

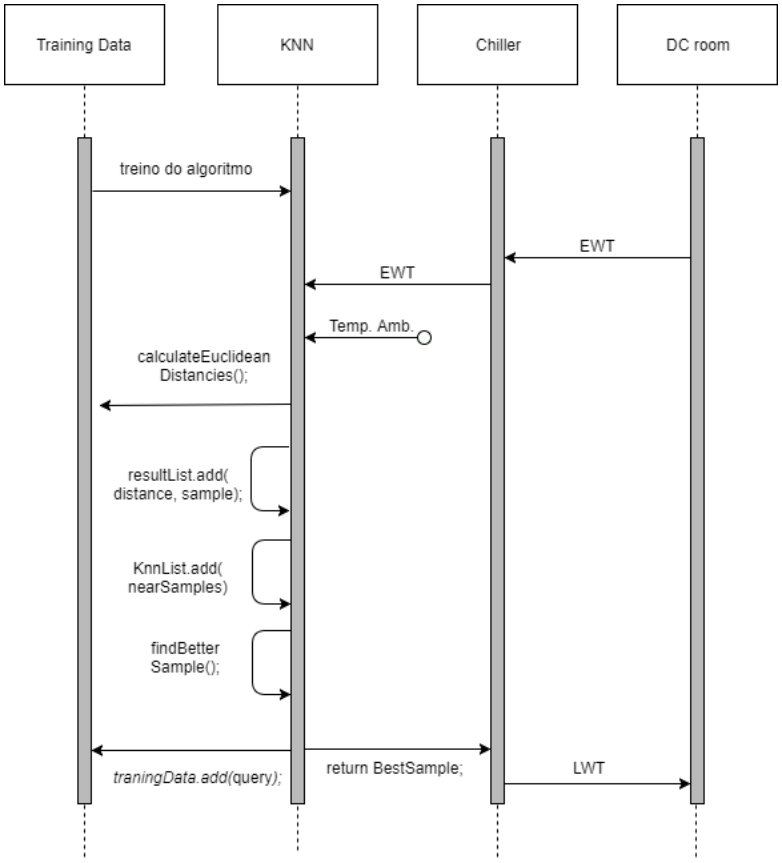


Figura 36 - Diagrama de sequência do funcionamento do KNN

4.1.2.2. *K-Means Clustering*

Explicado o funcionamento do *K-Means Clustering* no capítulo da arquitetura, será mostrado um exemplo referente ao algoritmo implementado para que se entenda melhor o seu funcionamento. O simulador lança o método *RunKMeansClustering* da classe *kMeansClustering*, que recebe como parâmetro a lista do training data. Prepara o training data para treinar o algoritmo e é chamado o método *generateCentroid* (Figura 37), que gera posições aleatórias para os *centroids* dentro dos limites especificados da temperatura ambiente (*randomX*) e EWT (*randomY*):

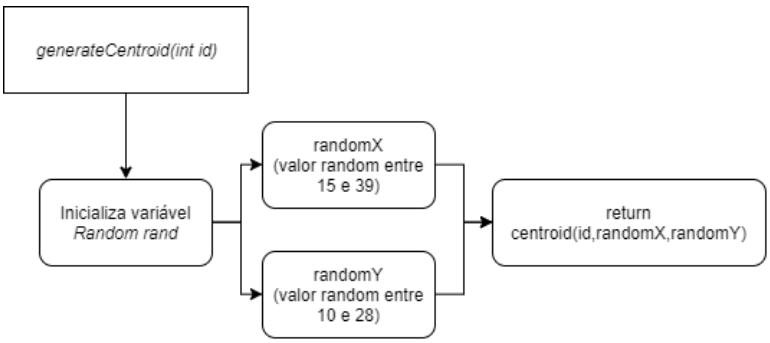


Figura 37 - Funcionamento do generateCentroid()

Os *centroids* criados são guardados na lista *centroidList*, do tipo *Centroid*. É importante referir que foram criados os objetos, *Centroid* e *Cluster*, para estruturar a informação e melhorar o desempenho do algoritmo:

$$Centroid = \begin{cases} int\ ID; \\ double\ x; \\ double\ y; \end{cases} \qquad ClusterPoint = \begin{cases} Centroid\ c; \\ InputData\ td; \end{cases}$$

Como exemplo, pode-se visualizar graficamente o *training data* com os *centroids* criados aleatoriamente na Figura 38.

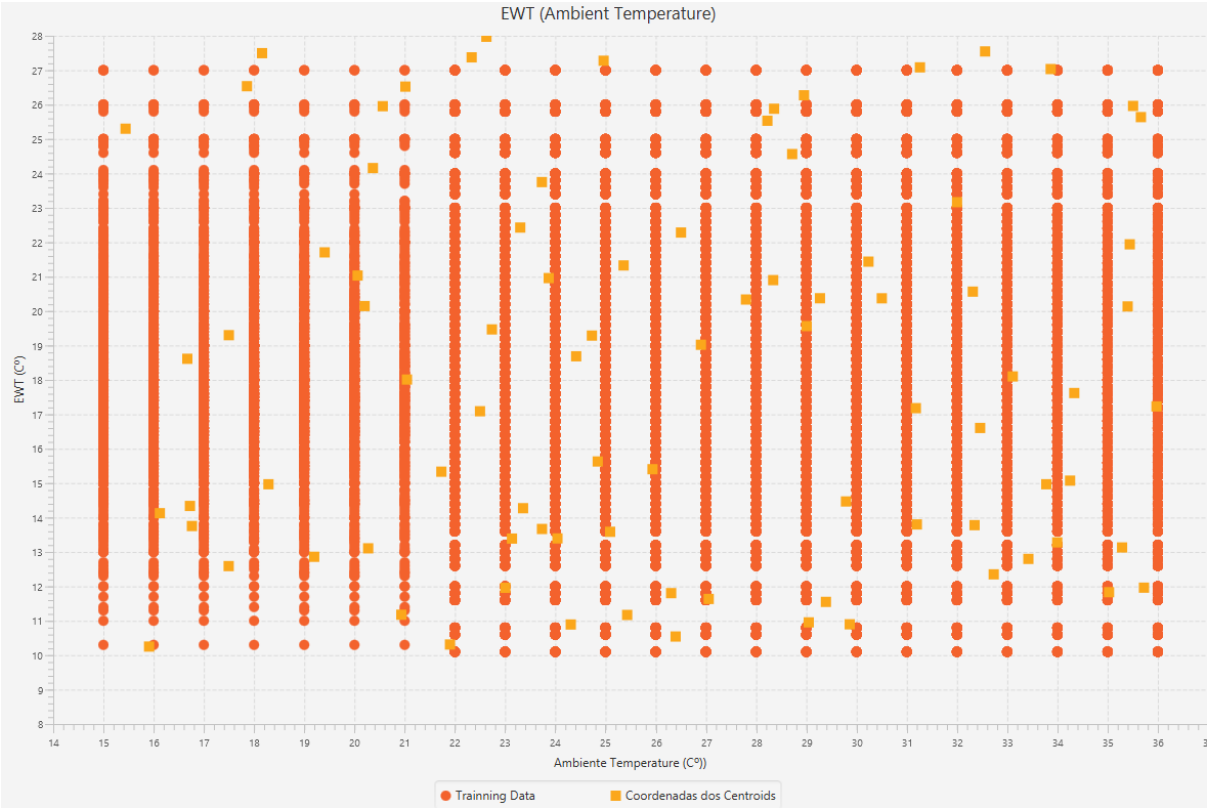


Figura 38 - *Training data* com os *centroids* (a amarelo) criados aleatoriamente

Uma vez criados os *centroids*, o algoritmo inicia o seu processo iterativo de encontrar as posições perfeitas para esses mesmos *centroids*. Durante uma iteração, o algoritmo executa várias etapas que se podem ver no esquema da Figura 39.

Figura 39 - Etapas de uma iteração

Primeira etapa começa por descobrir qual o *centroid* mais perto para cada amostra do *training data*. Para isto, calcula-se a distância euclidiana entre a amostra e todos os *centroids* existes. O *centroid* a menor distância é atribuído a essa amostra. Descoberto o *centroid*, a amostra é adicionada ao *cluster* respetivo na lista *ClustersList*. A todos os *centroids* criados foi atribuído um *ID* que permite identificar os *clusters*. Com todas as amostras atribuídas aos diferentes *clusters*, as coordenadas dos *cendroids* são recalculadas pelo método *centroidsNewPositions* (Figura 40) que recebe como parâmetro a lista *ClustersList* e retorna uma lista do tipo *Centroid*. As novas coordenadas de cada *centroid* serão calculadas pela média das posições das amostras do *cluster* referente a esse *centroid*:

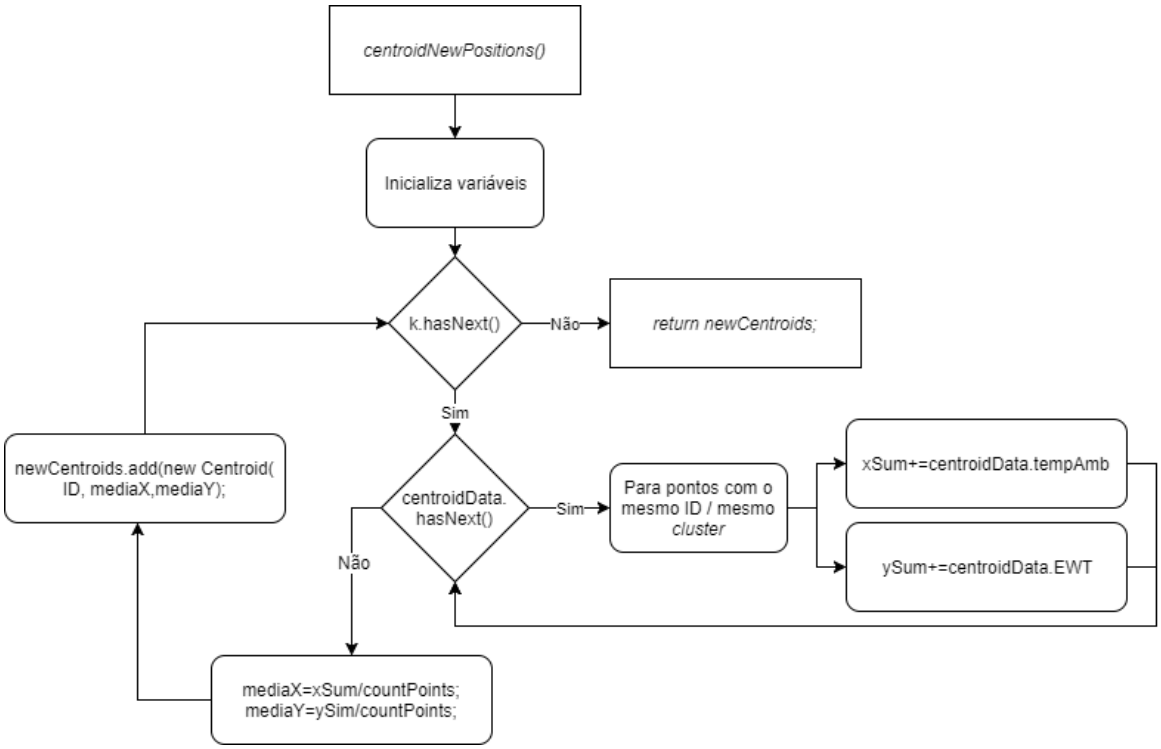


Figura 40 - Diagrama de atividade do método *centroidNewPositions*

Esta lista será a nova lista *centroidList* com os *centroids* reposicionados. Terminadas estas etapas, o algoritmo segue para a próxima iteração, realizando o mesmo processo.

Para este caso realizou-se alguns testes com o objetivo de encontrar o número mínimo de iterações necessárias para que as posições dos *centroids* convergisse. Como se repara, o *training data* é mais denso no centro. Caso o número de iterações seja muito elevado, os *centroids* irão convergir para todos para as mesmas posições. Realizados os testes, verifica-se que o número de iterações perfeito seria **10**.

Depois de todo este processo, cada *cluster* é caracterizado pela amostra dentro dele que possua o melhor valor de eficiência. Sendo assim, cada *cluster* será um *setup* diferente para o *chiller*. Assim, o algoritmo fica pronto a tratar de *queries*. Quando ocorre uma *query*, o algoritmo calcula as distâncias entre a *query* e todos os *centroids* até encontrar o mais próximo. O *cluster* referente a esse *centroid* irá caracterizar essa *query* e assim retornar o *setup* para o *chiller*. Esta *query* entra para o *cluster* que a carateriza e passa a ser uma amostra do *training data*, para que o algoritmo "aprenda" continuamente. No entanto, para não tornar o algoritmo demasiado "pesado", só apenas depois de 40 *queries* é que volta a reposicionar os *centroids*.

Na Figura 41 é possível observar toda a atividade do *K-Means Clustering* descrita anteriormente num diagrama de atividade.

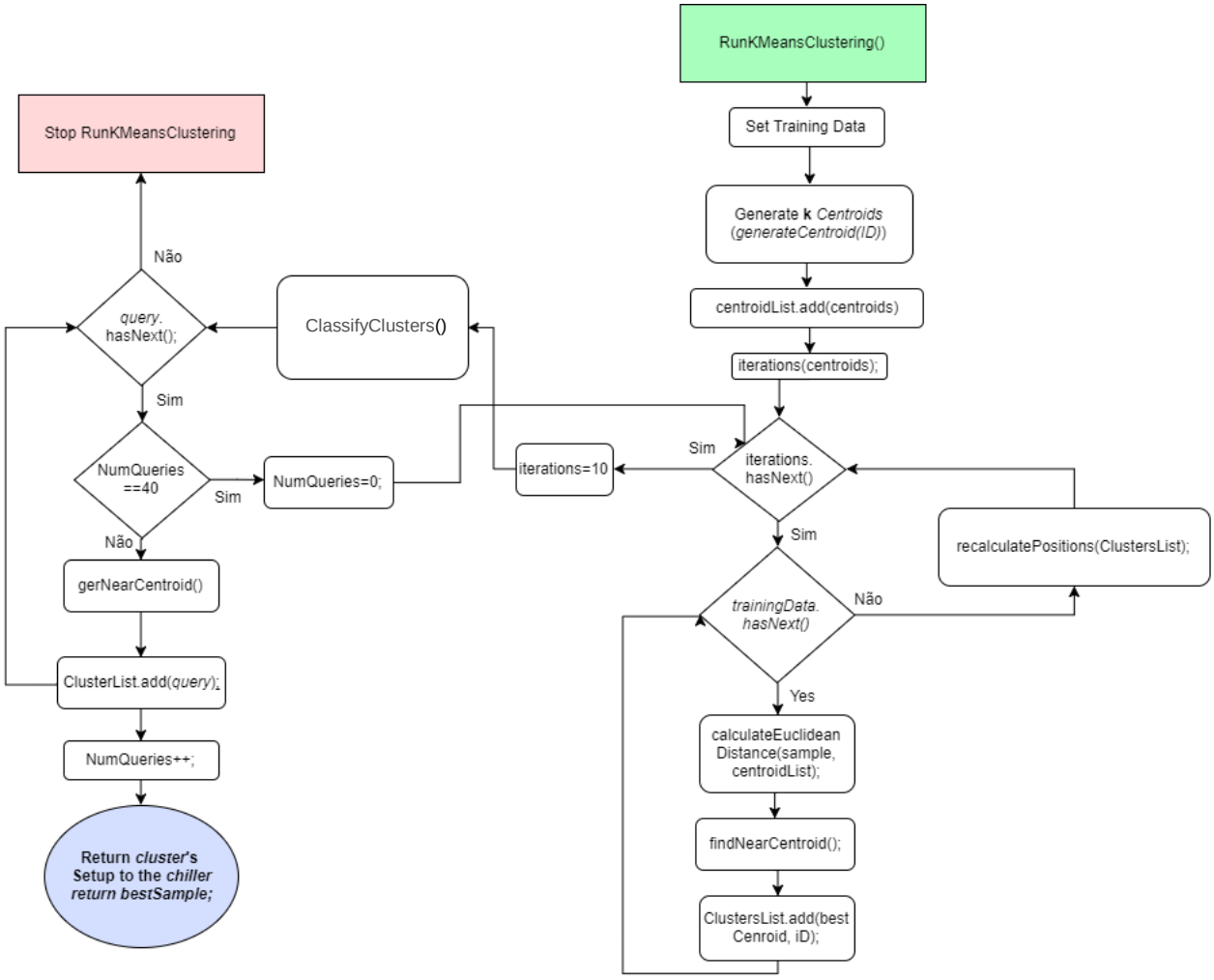


Figura 41 - Diagrama de atividade do algoritmo *K-Means Clustering*

Como é possível observar pelo diagrama de atividade, o algoritmo inicia o seu processo criando *centroids* aleatoriamente, através do método *generateCentroid()*, atribui um ID a cada um e guarda-os na lista *centroidList*. De seguida começa o processo iterativo. Consoante o número

escolhido para as iterações, o algoritmo irá iterar até as posições dos *centroids* convergirem. Uma vez com as posições finais obtidas, o algoritmo está pronto para tratar das *queries* que receber. Para classificar uma *query*, o algoritmo encontra o *centroid* mais próximo do ponto da *query* através do método *getNearCentroid()*. Ao encontrar o *centroid* mais próximo, a *query* é classificada pelo *cluster* referente a esse *centroid*, devolvendo o *setup* que classifica esse mesmo *cluster*.

Este processo é apresentado também no diagrama de sequência da Figura 42

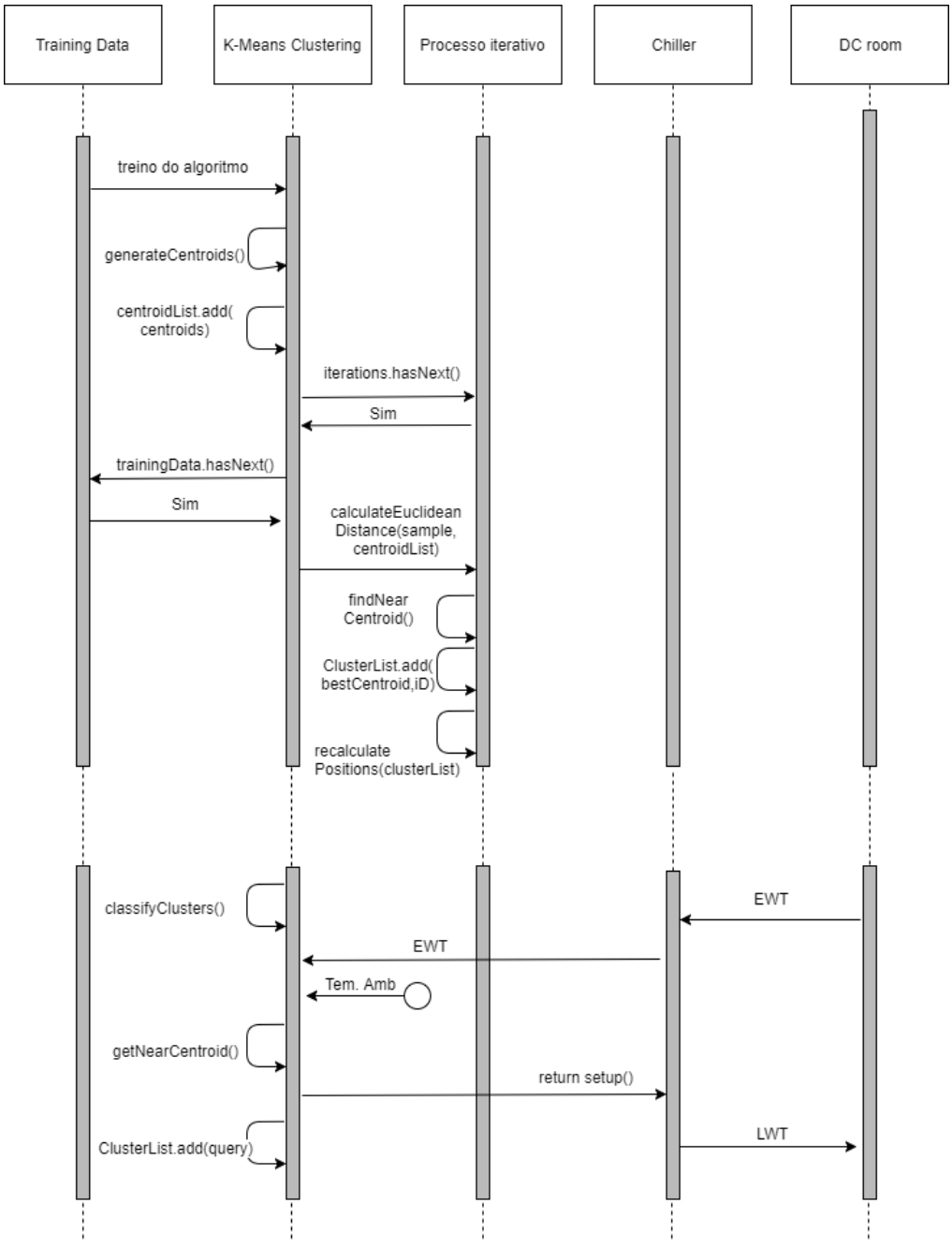


Figura 42 - Diagrama de sequência do processo iterativo e tratamento de *queries* do *K-Means*

4.1.3. Simulator

Este grupo apresenta apenas a classe *StartSim*, que é encarregada de iniciar a simulação. Classe onde se encontra a *main ()*, inicializa as variáveis e as classes dos diferentes algoritmos implementados e cria a lista de *queries* para testar esses algoritmos. Apresenta-se na Figura 43 o diagrama de classe do Simulador.

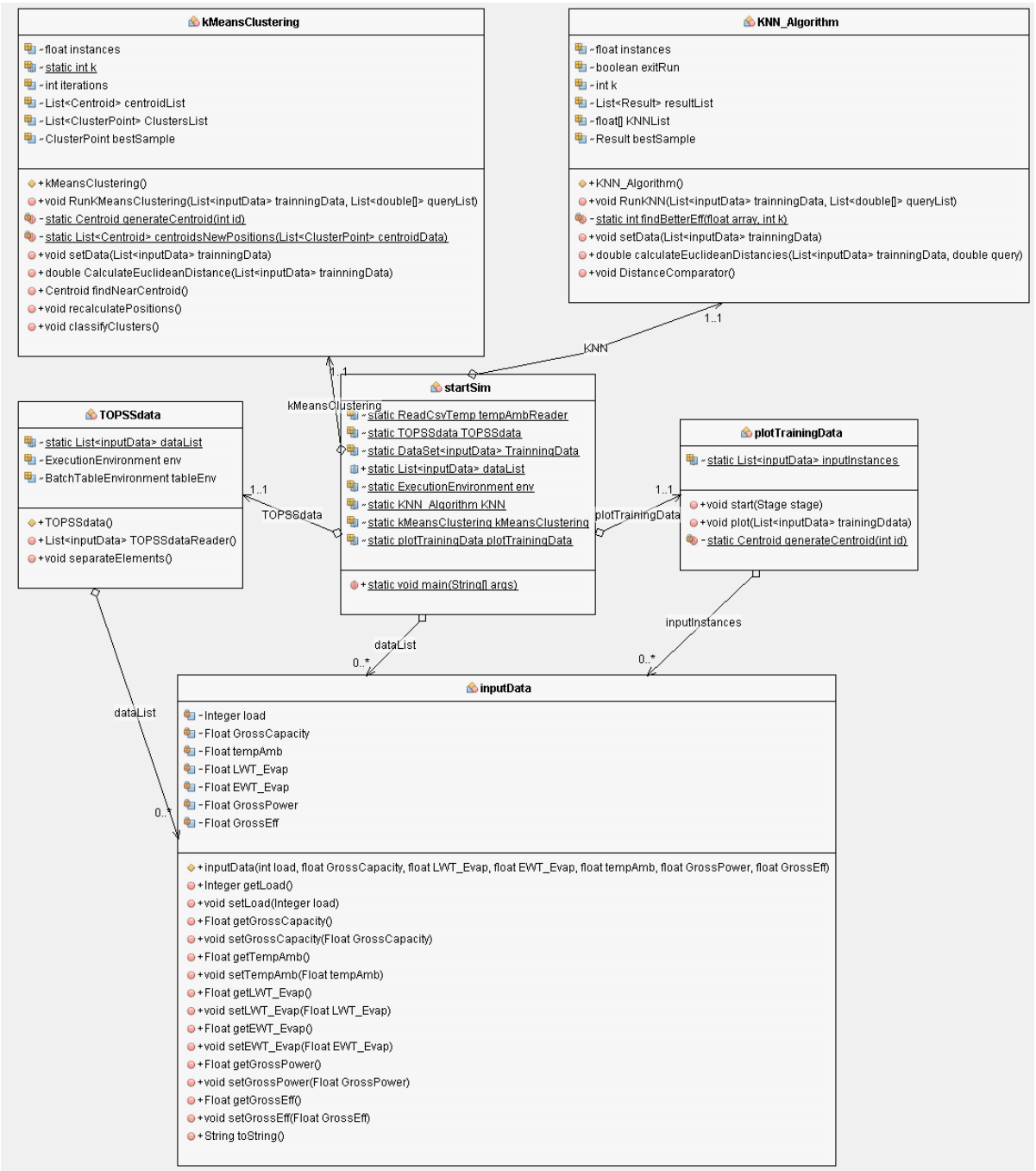


Figura 43 - Diagrama de classe do Simulador

O diagrama de classes mostra as relações que o grupo do simulador tem com as classes restantes, como os algoritmos e o *trainingData*. É aqui que tudo começa, sendo possível ler o *training data* dos ficheiros *.csv*, mostrar graficamente o *training data* pela classe *PlotTrainingData*, e escolher qual o algoritmo que se pretende testar, o KNN (*RunKNN*()) ou o *K-Means Clustering* (*RunKMeansClustering()*).

Tabela 5 - Descrição do grupo Simulator

Classe	Variáveis
StartSim	• *TOPSSdata TOPSSdata* • List<inputData> DataList • KNN_Algorithm KNN; • kMeansClustering kMeansClustering; • plotTrainingData plotTrainingData;

4.1.3.1. Classe *StartSim*

Na classe *StartSim* pode-se encontrar a *main()*, onde tudo começa. É aqui que são lançados todos os processos para a simulação. Existem três processos, explicados anteriormente, que são iniciados pela *StartSim*:

- *TOPSSdata;*
- *KNN_Algorithm;*
- *kMeansClustering.*

O *TOPSSdata* é inicializado para que seja possível chamar o método *TOPSSdataReader*, e assim ler os ficheiros *.csv* para criar o *training data*. Depois de guardar o *training data* na lista *DataList*, é pedido ao utilizador para que escolha qual o algoritmo que deseja utilizar para a simulação – KNN ou *K-Means Clustering*. Consoante a escolha do utilizador o objeto de cada algoritmo é inicializado e chamado o método *Run* para que se inicie o algoritmo escolhido pelo utilizador.

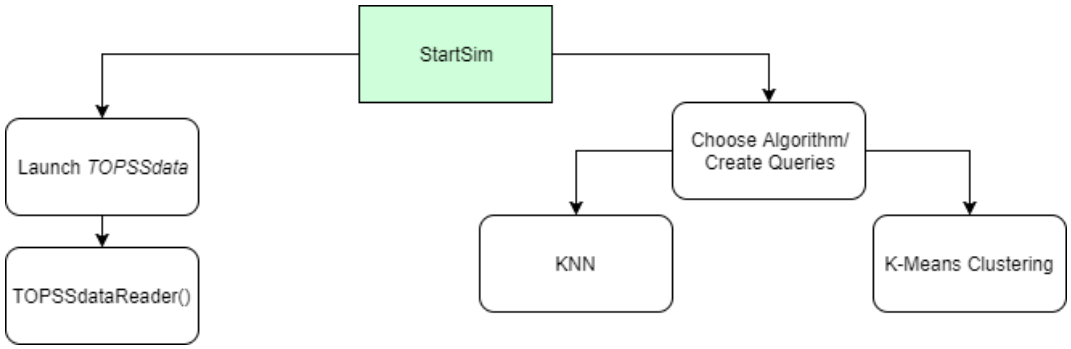


Figura 44 - Diagrama de atividade do StartSim

Através do diagrama de atividade apresentado, observa-se que é pela classe *StartSim* que tudo se inicia. Não só inicia o processo de ler os ficheiros *.csv* e criar o *training data* como é nela que existe possibilidade de escolher qual o algoritmo que se deseja testar. Verifica-se esta explicação no diagrama de sequência apresentado na Figura 45.

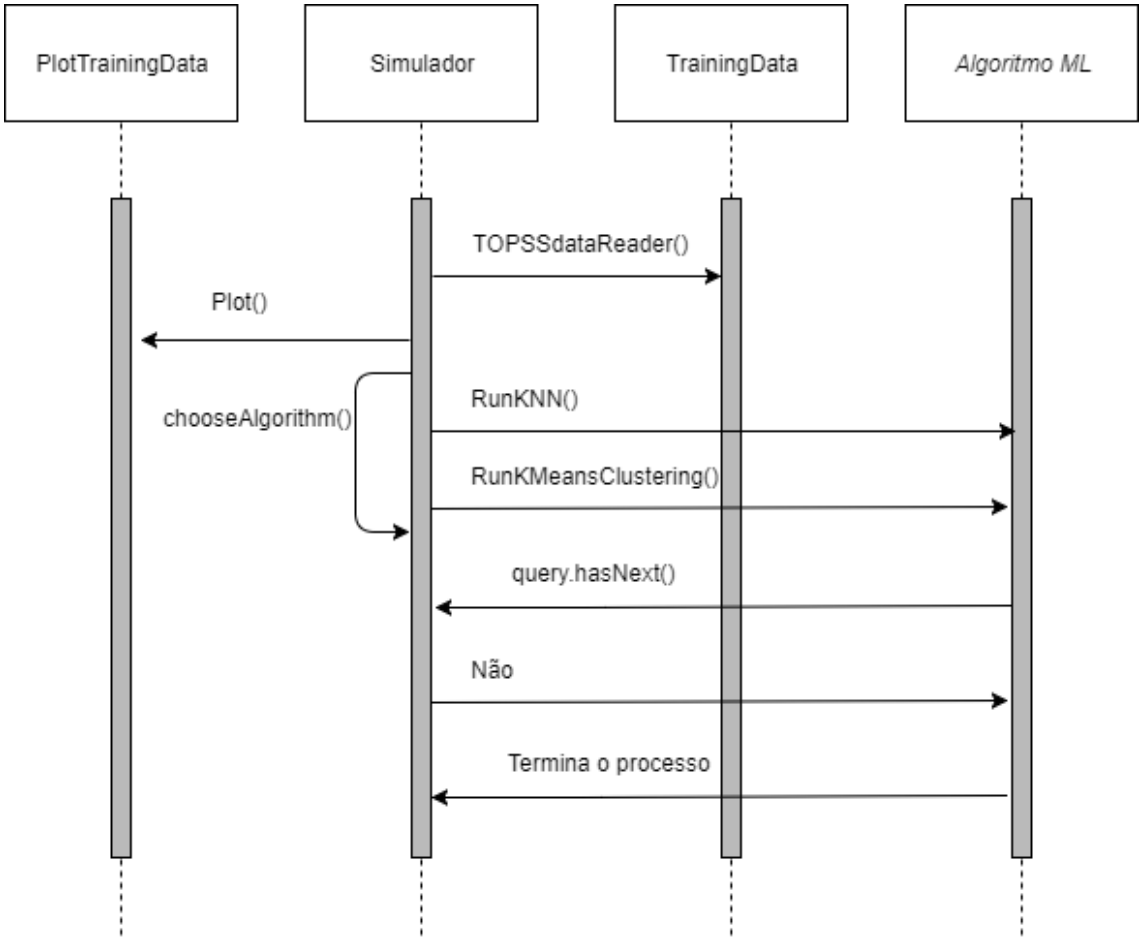


Figura 45 - Diagrama de sequência do Simulador

5

Análise de Resultados

Com os algoritmos implementados, procedeu-se à realização de diferentes testes com o propósito de comparar a *performance* de cada um. O objetivo principal será encontrar qual o algoritmo que melhor soluciona o problema, tendo em conta vários fatores como o tempo de treino, o tempo de resposta e a qualidade da resposta apresentada.

Sendo assim foi calculado o tempo de treino de cada algoritmo, ou seja, o tempo que demora a ser treinado pelo *training data* até estar pronto a receber *queries*. Outro teste foi calcular o tempo necessário a processar uma *query* até devolver um *setup* para o *chiller*. Foi verificada também a qualidade do *setup* devolvido por cada algoritmo, de modo a que se perceba qual soluciona melhor as *queries* apresentadas. Para comparar a qualidade, foram construídas matrizes de confusão para cada algoritmo, ferramenta padrão para avaliação de modelos estatísticos de *machine learning*.

O que se deseja encontrar é o equilíbrio entre todos estes fatores, sendo o mais importante a qualidade da resposta apresentada. No entanto, é desejado o mínimo tempo de treino possível, o mínimo tempo de processamento e o máximo valor de precisão. Como já foi referido, cada *input*

tem um *setup* que melhor de adequa para esse caso, é necessário descobrir qual dos algoritmos encontra o melhor *setup* no menor período de tempo.

Para a realização dos testes foi criado uma lista de vinte *queries* diferentes, registando todos os tempos e *setup* observados para cada uma delas. Foi feita uma média para cada valor final. Os testes foram realizados numa máquina com as seguintes características:

- Processador Intel Core i5-6500 CPU @ 3.20GHz;
- Memória instalada (RAM) 16GB;
- Sistema Operativo Windows 10 Pro 64 bits;
- Disco SSD Crucial MX300 275GB SATA 2.5” 7mm.

Para o cálculo do tempo de processamento foi criada a classe *Chronometer*. Esta classe possibilita a criação de um cronómetro que dispara e termina quando o programador desejar, devolvendo o tempo contado em milissegundos.

5.1. Classe *Chronometer*

A classe *Chronometer* foi implementada para ser possível criar o objeto *Chronometer*, com o objetivo de obter resultados do tempo de treino e processamento.

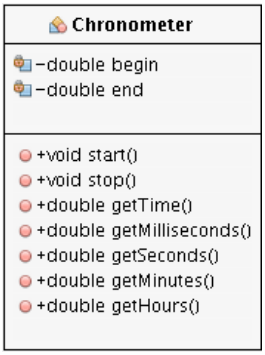


Figura 46 - Classe *Chronometer*

A utilização do objeto é bastante simples. É criada uma variável do tipo *Chronometer* em cada classe dos algoritmos. Para iniciar o cronómetro, é chamado o método *start()* que utiliza a função do *java System.currentTimeMillis()*. Esta função retorna o valor do tempo atual em milissegundos para a variável *begin* do objeto *Chronometer*. Para terminar o cronómetro, é chamado o método *stop()* que utiliza a mesma função do *start()* para obter o tempo atual em milissegundos, guardando o valor na variável *end* do objeto.

Os outros métodos existentes na classe retornam a diferença entre a variável *end* e *start*, sendo possível obter o tempo da diferença em milissegundos, segundos, minutos ou horas.

5.2. Testes ao algoritmo KNN

O KNN, com um funcionamento bem diferente do *K-Means Clustering*, inicia todo o processo calculando as distâncias entre a *query* e as amostras do *training data*, não sendo necessário agrupar. Por isso, o cronómetro dispara quando este recebe uma *query* e termina quando devolve um *setup*. Na Tabela 6 verifica-se os resultados.

Tabela 6 - Resultados de *setup* e tempo de processamento do KNN

QUERY ID	QUERY (TEMP.AMB; EWT)	EFF.	LWT (°C)	LOAD (%)	TEMPO (SEGUNDOS)
1	(27.5; 17.0)	6.00	15,0	30	0,023
2	(12,3; 14,0)	6,08	15,0	40	0,016
3	(21,3; 13,4)	6,00	9,0	60	0,015
4	(18,7; 13,0)	5,40	13,0	40	0,011
5	(28,3; 18,2)	5,98	12,0	40	0,013
6	(17,0; 25,4)	6,00	16,0	70	0,012
7	(38,4; 20,8)	6,00	11,0	50	0,005
8	(35,4; 12,3)	6,02	11,0	50	0,005
9	(24,5; 25,0)	6,00	15,0	80	0,005
10	(20,4; 13,4)	5,75	10,0	60	0,005
11	(16,8; 18,5)	6,09	15,0	70	0,005
12	(30,0; 40,4)	5,97	16,0	100	0,005
13	(14,3; 28,7)	6,00	15,0	80	0,005
14	(17,5; 26,4)	6,02	15,0	70	0,005
15	(22,4; 22,3)	5,99	13,0	80	0,005
16	(32,2; 17,4)	6,00	16,0	90	0,014
17	(41,5; 27,2)	5,96	8,0	90	0,005
18	(35,4; 19,5)	5,99	7,0	60	0,005
19	(20,0; 17,5)	6,35	15,0	70	0,005
20	(18,4; 20,5)	5,89	16,0	50	0,005
MÉDIA		**5,975**	**-**	**TOTAL**	**0,169**

Como foi explicado na arquitetura, o KNN inicia logo o seu processo pelo tratamento de *queries*, não passando por um processo de preparação aleatória e iterativa como o *K-Means Clustering*. Por isto, o KNN irá sempre retornar os mesmos *setups* para o mesmo conjunto de *queries* recebidas em diferentes utilizações do algoritmo. Para as mesmas 20 *queries* do teste, escolhidas aleatoriamente, o algoritmo apresentou resultados iguais em utilizações diferentes, como previsto. Assim, é apresentado apenas os resultados de uma utilização do KNN.

A Tabela 6 mostra para cada *query*, os valores do *setup* encontrado na amostra mais frequente dos vizinhos e o tempo que demora a processar cada *query*. Relembra-se que, no algoritmo KNN, o seu processamento passa por encontrar as 124 amostras do *training data* mais próximas da *query*. Quando encontradas as 124 amostras, o algoritmo percorre-as até encontrar a *sample* mais frequente nos vizinhos. Caso exista 2 ou mais *samples* com o mesmo número de aparições, o valor **k** (número de vizinho) é decrementado até desfazer o empate. Quando o KNN encontrar a *sample* mais frequente, retornará o *setup* para os *chillers*.

Ao observar os *setups* devolvidos pelo KNN, verifica-se, num aspeto prático e realista, que os resultados fazem sentido. Pode-se ver como exemplo as *queries* 16 e 17. Ambas apresentam uma temperatura ambiente elevada, mas dentro do DC a *query* 16 apresenta uma temperatura com 10 °C abaixo da *query* 17. Faz sentido que para refrigerar a sala na situação da *query* 17 seja necessária uma temperatura LWT menor do que para a *query* 16, situação que se verifica nos resultados. Este é um bom exemplo de *setups* pois os valores de eficiência são bastante semelhantes e o valor do *load* igual nos dois casos. Para outras *queries* é difícil comparar devido aos diferentes valores de *load* e LWT. Outro fator que também pesa bastante é a veracidade dos dados recolhidos no TOPSS.

Em relação ao tempo, verifica-se que o processamento de cada *query* é rápido, sendo a primeira *query* a mais lenta a ser processada. No entanto, processar 20 *queries* em 0,169 segundos é bastante satisfatório, pois segundo informações da empresa motivadora deste tema, a EWT lida pelo *chiller* é guardada nas bases de dados do DC em cada minuto, embora o sensor do *chiller* esteja constantemente a ler. Ora isto, em termos práticos, abre uma janela de 1 minuto para que a *query* seja totalmente processada, que em relação ao tempo obtido neste estudo, é bastante suficiente. Verifica-se também um valor de tempo fora do normal no *query* 16, algo que pode ser explicado pela provável existência do referido empate, onde não ocorre uma maioria nas votações do vizinho mais frequente.

5.3. Testes ao algoritmo *K-Means Clustering*

O *K-Means Clustering*, ao contrário do KNN, necessita de passar por uma fase de treino inicial. Como foi explicado anteriormente, o algoritmo gera *centroids* aleatórios, que são reposicionados, pelo *training data*, o número de vezes que foi atribuído ao valor de iterações escolhidas. O objetivo é criar *clusters* proporcionalmente distribuídos no *training data* de modo a classificar as *queries* recebidas. Visto isto, conclui-se que para diferentes utilizações do algoritmo os resultados poderão não ser iguais. Por isso, foram realizados trinta testes idênticos com o mesmo conjunto de *queries*, de modo a comparar as divergências dos resultados.

Antes de verificar os resultados obtidos, foi calculado o tempo de treino necessário. Para isto utilizou-se classe *Chronometer*, a mesma utilizada nos testes do KNN.

O algoritmo demora, em média realizada nos 30 testes, 0,227 segundos a preparar-se para iniciar o processamento de cada *query*.

Em seguida foi realizado o mesmo teste do que ao KNN. Foram retirados os resultados (LWT, *Eff.* e *Load*), bem como o tempo de processamento de cada uma das *queries*. No caso do *K-Means Clustering*, os resultados foram retirados em 30 testes diferentes para que seja possível observar melhor a qualidade do algoritmo.

Dos 30 testes realizados, apresenta-se os resultados de 3 testes onde se obteve as melhores médias de eficiência.

Os resultados dos testes podem ser observados na Tabela 7, Tabela 8 e Tabela 9. Começa-se por apresentar os resultados referentes a um dos testes que apresentou melhor média de eficiência.

Tabela 7 – 1ª tabela de resultados do *K-Means Clustering*

QUERY ID	QUERY (TEMP.AMB; EWT)	EFF.	LWT (°C)	LOAD (%)	TEMPO (SEGUNDOS)
1	(27,5; 17,0)	6,00	9,0	40	0,006
2	(12,3; 14,0)	5,91	14,0	30	0,003
3	(21,3; 13,4)	5,97	15,0	40	0,003
4	(18,7; 13,0)	5,15	12,0	30	0,003
5	(28,3; 18,2)	6,00	10,0	70	0,003
6	(17,0; 25,4)	5,80	16,0	80	0,002

7	(38,4; 20,8)	6,00	10,0	50	0,004
8	(35,4; 12,3)	5,98	11,0	40	0,001
9	(24,5; 25,0)	6,00	10,0	90	0,003
10	(20,4; 13,4)	5,15	12,0	30	0,003
11	(16,8; 18,5)	6,08	16,0	70	0,003
12	(30,0; 40,4)	5,97	16,0	100	0,003
13	(14,3; 28,7)	5,80	16,0	80	0,004
14	(17,5; 26,4)	5,80	16,0	80	>0,001
15	(22,4; 22,3)	5,99	12,0	100	>0,001
16	(32,2; 17,4)	6,00	16,0	90	>0,001
17	(41,5; 27,2)	5,96	8,0	90	>0,001
18	(35,4; 19,5)	6,00	10,0	50	>0,001
19	(20,0; 17,5)	5,69	16,0	70	0,001
20	(18,4; 20,5)	5,96	16,0	70	>0,001
MÉDIA		**5,8605**	**-**	**TOTAL**	**0,042**

De seguida, são apresentados os resultados do segundo teste. Note-se que os tempos de processamento diminuiem ao longo do processamento de cada *query*.

Tabela 8 - 2ª tabela de resultados do *K-Means Clustering*

QUERY ID	QUERY (TEMP.AMB; EWT)	EFF.	LWT (°C)	LOAD (%)	TEMPO (SEGUNDOS)
1	(27,5; 17,0)	6,00	9,0	40	0,005
2	(12,3; 14,0)	5,78	13,0	30	0,003
3	(21,3; 13,4)	5,97	15,0	40	0,004
4	(18,7; 13,0)	5,27	12,0	30	0,003
5	(28,3; 18,2)	6,00	10,0	70	0,003
6	(17,0; 25,4)	6,09	16,0	70	0,003
7	(38,4; 20,8)	6,00	10,0	50	0,003
8	(35,4; 12,3)	6,00	15,0	40	0,003
9	(24,5; 25,0)	6,00	13,0	70	0,003
10	(20,4; 13,4)	5,27	12,0	30	0,003
11	(16,8; 18,5)	6,09	16,0	70	0,003
12	(30,0; 40,4)	5,79	13,0	100	0,003

13	(14,3; 28,7)	5,55	16,0	90	0,002
14	(17,5; 26,4)	5,69	16,0	80	>0,001
15	(22,4; 22,3)	5,99	12,0	100	>0,001
16	(32,2; 17,4)	6,00	16,0	90	>0,001
17	(41,5; 27,2)	5,96	8,0	90	>0,001
18	(35,4; 19,5)	6,00	10,0	50	>0,001
19	(20,0; 17,5)	5,69	16,0	70	>0,001
20	(18,4; 20,5)	6,09	16,0	70	>0,001
MÉDIA		**5,8615**	-	**TOTAL**	**0,041**

Neste teste também se verifica a diminuição dos tempos ao longo do processo. Observe-se a tabela última tabela de resultados.

Tabela 9 - 3ª tabela de resultados do *K-Means Clustering*

QUERY ID	QUERY (TEMP.AMB; EWT)	EFF.	LWT (°C)	LOAD (%)	TEMPO (SEGUNDOS)
1	(27,5; 17,0)	6,00	9,0	40	0,004
2	(12,3; 14,0)	5,91	14,0	30	0,003
3	(21,3; 13,4)	5,91	14,0	30	0,003
4	(18,7; 13,0)	5,02	11,0	30	0,002
5	(28,3; 18,2)	6,00	10,0	40	0,003
6	(17,0; 25,4)	6,22	16,0	70	0,003
7	(38,4; 20,8)	6,00	16,0	60	0,003
8	(35,4; 12,3)	5,96	11,0	50	0,003
9	(24,5; 25,0)	6,00	15,0	80	0,004
10	(20,4; 13,4)	5,22	13,0	40	0,003
11	(16,8; 18,5)	6,33	16,0	70	0,003
12	(30,0; 40,4)	5,99	14,0	100	0,004
13	(14,3; 28,7)	6,35	16,0	70	0,003
14	(17,5; 26,4)	5,96	16,0	70	0,003
15	(22,4; 22,3)	5,99	13,0	80	0,002
16	(32,2; 17,4)	6,00	14,0	70	<0,001
17	(41,5; 27,2)	5,99	11,0	100	>0,001
18	(35,4; 19,5)	6,00	8,0	30	>0,001

19	(20,0; 17,5)	5,69	16,0	70	0,001
20	(18,4; 20,5)	6,09	16,0	70	>0,001
MÉDIA		**5,932**	**-**	**TOTAL**	**0,046**

Ao observar as tabelas resultantes destes 3 testes, pode-se afirmar que o algoritmo funcionou da forma esperada. É de esperar que o *K-Means Clustering* tenha um funcionamento difícil, pois, por vezes, a qualidade do seu funcionamento depende dos *centroids* gerados aleatoriamente, independentemente do processo de iteração que passam para que se possam reposicionar em posições mais eficientes no *training data*. No entanto, verificando atentamente os *setup* devolvidos pelo algoritmo, conclui-se que fazem sentido face às *queries* recebidas.

Ao que era de esperar, os 30 testes realizados apresentam por vezes *setups* diferentes para as mesmas *queries*. Isto deve-se às diferentes posições dos *centroids* em cada funcionamento do algoritmo, onde cada *cluster* é classificado por amostras diferentes. Por isto, pode-se afirmar que, embora o algoritmo obtenha o melhor *setup* do *cluster* correspondente à *query*, esse *setup* pode não ser o melhor. No entanto, e comprovando a boa implementação do *K-Means Clustering*, algumas *queries* apresentam o mesmo *setup* para testes diferentes, nomeadamente a *query* 20 na Tabela 8 e na Tabela 9, ou até mesmo a *query* 19 que apresenta um *setup* igual em todos os testes. O tempo utilizado para cada *query* é bastante pequeno, ou seja, o algoritmo é rápido a processar cada *query*, sendo sempre a primeira a mais duradoura. Nota-se que, depois do processamento da 12º *query*, o algoritmo começa a processar as restantes *queries* num menor tempo, comparando com as anteriores.

5.4. Comparação entre algoritmos

Uma vez retirados os resultados de cada algoritmo, foi realizada uma comparação entre eles. Esta comparação tem como base a qualidade dos resultados e os tempos obtidos de cada um. Note-se que para o caso do *K-Means Clustering* realizou-se a média dos tempos obtidos nos trinta testes realizados.

Para finalizar a comparação entre os algoritmos, foram construídas as matrizes de confusão com o objetivo de visualizar a *performance* de cada algoritmo e a sua precisão. As matrizes de confusão servem para comparar os resultados previsos com os obtidos de algoritmos ML, retirando características como a precisão ou o *recall*.

5.4.1. Comparação de Tempos

Para começar compararam-se os tempos na Tabela 10.

Tabela 10 - Comparação dos tempos obtidos por cada algoritmo

	KNN	*K-MEANS CLUSTERING*
TEMPO DE TREINO (SEGUNDOS)	-	0.227
TEMPO DE PROCESSAMENTO (20 QUERIES) (SEGUNDOS)	0,169	0,043
TOTAL (SEGUNDOS)	**0,169**	**0,270**

Observando a tabela, verifica-se que o tempo total gasto para processar 20 *queries* foi menor no KNN, em comparação com o *K-Means Clustering*. No entanto o KNN não passa por um processo de treino antes de resolver as *queries*, como o *K-Means*. O KNN é treinado ao longo do seu funcionamento, quando uma *query* é tratada, esta deixa de ser *query* e passa a pertencer ao *training data*. Já no *K-Means*, existe um processo de treino que prepara os *clusters* criados no *training data*. Este é o processo mais demorado (0,227 segundos), o que faz com que o *K-Means* seja o algoritmo que consume mais tempo. No entanto, o *K-Means* não passa por este processo em vão. Uma vez realizado o processo de treino, o *K-Means* apresenta um tempo de processamento de *queries* mais rápido que o KNN, nomeadamente depois de processar a 12ª *query*. Ou seja, comparando apenas pelos temos de processamento de *queries*, o *K-Means* é mais rápido, face ao tempo de processamento obtido no KNN. Isto deve-se à criação dos *clusters*, que oferecem eficiência no desempenho do algoritmo. Verifica-se os tempos no gráfico da Figura 47.

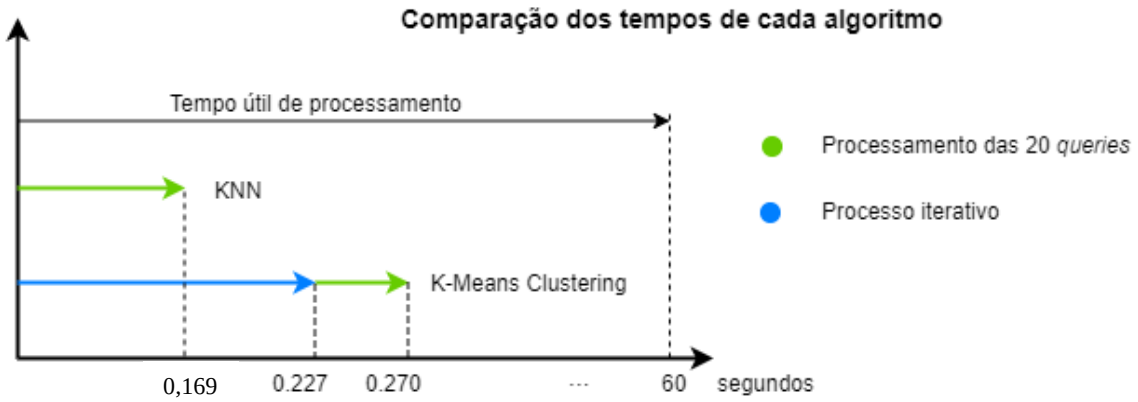


Figura 47 - Gráfico de comparação dos tempos de cada algoritmo

No entanto, no caso de o *K-Means Clustering* processar 40 *queries*, é realizado um novo ajuste às posições dos *clusters*. Como foi explicado, o processo de treino é demorado, e reposicionar os *clusters* devido a uma *query* pouco ou nada irá afetar. Por isso foi implementado

que, quando receber a 40º *query* seguida o algoritmo adiciona as *queries*, atualiza o *training data* e consequentemente as posições dos *clusters*.

Como foi referido, na utilização dos algoritmos num caso real, existe um período de 1 minuto para que as *queries* sejam processadas. Comparando os tempos obtidos pelos dois algoritmos, pode-se concluir que ambos realizam o processo no tempo adequado, embora para um caso real o tempo estimado possa ser maior. Na prática, o processamento passa pelo acesso do algoritmo à base de dados que guarda os valores EWT lidos pelo *chiller* de minuto a minuto, obter a temperatura ambiente atual, ler estes valores e iniciar o processamento, e no fim devolver o *setup* ao *chiller*. Este processo será obviamente mais demorado que a simulação realizada.

5.4.2. Comparação de Resultados

Ao observar as tabelas referentes aos resultados, é possível encontrar a média dos valores de eficiência encontrados para cada *query*.

Tabela 11 - Comparação da eficiência média de cada algoritmo

	KNN	K-MEANS CLUSTERING MELHOR MÉDIA OBTIDA NOS 30 TESTES
EFICIENCIA MÉDIA	**5,975**	**5,932**

O algoritmo que alcançou melhores valores de eficiência para cada *query* foi o KNN, com uma média de 5.975. No entanto, este aspeto não permite afirmar que um algoritmo seja melhor que o outro. O algoritmo poderá ter encontrado a amostra de melhor eficiência, no entanto, o *setup* que essa amostra devolve pode não ser o melhor para a *query* em causa. Para entender qual dos algoritmos retorna melhores *setups* foram encontrados os *setups* perfeitos para cada *query*, que serão explicados na Tabela 13.

Na Tabela 12 são apresentados os resultados dos dois algoritmos lado a lado, sendo que para o *K-Means Clustering* foram escolhidos os resultados com a melhor média de eficiências obtida nos 30 testes.

Tabela 12 - Tabela com os resultados dos 2 algoritmos

		KNN			K-MEANS CLUSTERING		
QUERY ID	Query (TEMP.AMB; EWT)	Eff.	LWT (°C)	LOAD (%)	Eff.	LWT (°C)	LOAD (%)
1	(27,5; 17,0)	6,00	15,0	30	6,00	9,0	40
2	(12,3; 14,0)	6,08	15,0	40	5,91	14,0	30
3	(21,3; 13,4)	6,00	9,0	60	5,91	14,0	30
4	(18,7; 13,0)	5,40	13,0	40	5,02	11,0	30
5	(28,3; 18,2)	5,98	12,0	40	6,00	10,0	40
6	(17,0; 25,4)	6,00	16,0	70	6,22	16,0	70
7	(38,4; 20,8)	6,00	11,0	50	6,00	16,0	60
8	(35,4; 12,3)	6,02	11,0	50	5,96	11,0	50
9	(24,5; 25,0)	6,00	15,0	80	6,00	15,0	80
10	(20,4; 13,4)	5,75	10,0	60	5,22	13,0	40
11	(16,8; 18,5)	6,09	15,0	70	6,33	16,0	70
12	(30,0; 40,4)	5,97	16,0	100	5,99	14,0	100
13	(14,3; 28,7)	6,00	15,0	80	6,35	16,0	70
14	(17,5; 26,4)	6,02	15,0	70	5,96	16,0	70
15	(22,4; 22,3)	5,99	13,0	80	5,99	13,0	80
16	(32,2; 17,4)	6,00	16,0	90	6,00	14,0	70
17	(41,5; 27,2)	5,96	8,0	90	5,99	11,0	100
18	(35,4; 19,5)	5,99	7,0	60	6,00	8,0	30
19	(20,0; 17,5)	6,35	15,0	70	5,69	16,0	70
20	(18,4; 20,5)	5,89	16,0	50	6,09	16,0	70
MÉDIA		**5,975**	-	-	**5,932**		-

Como é possível verificar, muitas das *queries* recebem soluções semelhantes dos dois algoritmos, embora haja algumas falhas. Uma das falhas encontradas é na *query* 2, na solução do KNN. A *query* indica que a sala do DC está a 14,0 °C, o KNN responde com um *setup* de LWT igual a 15ºC. Ou seja, este *setup* irá aquecer a sala, objetivo que efetivamente não é desejado num sistema de refrigeração. Outra falha parecida a esta está na *query* 3, neste caso na solução proposta pelo *K-Means Clustering*.

Estas falhas podem acontecer por diversos motivos, tais como:

- Fidedignidade dos dados gerados pelo TOPSS é menor em comparação a dados reais, extraídos de um *chiller* real a atuar num DC. O *software* pode gerar dados que não sejam possíveis numa situação real, ou até mesmo o utilizador pode por lapso pedir dados com um *input* irrealista;
- Os algoritmos podem ser bem implementados, mas sem um *training data* de qualidade podem pecar na sua utilização;
- Os algoritmos podem necessitar de mais treino. Isto significa duas coisas – ou o *training data* não é suficientemente grande, ou o algoritmo necessita de várias utilizações para aprender continuamente. A segunda opção é a mais aceitável neste caso, dado que foram extraídas mais de 15000 amostras do TOPSS. Como o conceito de *machine learning* indica, o algoritmo aprende continuamente, devido a isso acredita-se que passadas várias utilizações a processar *queries* e a aumentar continuamente o *training data*, a qualidade dos resultados aumente.

Embora existam falhas, a maioria dos *setups* devolvidos pelos algoritmos estão, por lógica, corretos. Uma temperatura EWT mais baixa pode ser tratada por uma LWT mais elevada e um *load* elevado. E vice-versa para um caso de EWT elevado. Estes cenários podem ser encontrados nos resultados.

Para ter uma melhor perceção, foram descobertos os *setups* perfeitos para cada *query* do teste, que podem ser verificados na tabela 14.

Tabela 13 - *Queries* e os melhores *setups*, respetivamente

QUERY		**MELHOR SETUP**		
Nº	(TEMP.AMB; EWT)	Eff.	LWT (°C)	LOAD (%)
1	(27,5; 17,0)	6,00	15,0	30
2	(12,3; 14,0)	5,81	13,0	40
3	(21,3; 13,4)	6,00	12,0	40
4	(18,7; 13,0)	5,40	13,0	40
5	(28,3; 18,2)	6,00	10,0	40
6	(17,0; 25,4)	6,22	16,0	70
7	(38,4; 20,8)	6,00	10,0	50
8	(35,4; 12,3)	5,98	11,0	40
9	(24,5; 25,0)	6,00	15,0	80
10	(20,4; 13,4)	6,00	12,0	40
11	(16,8; 18,5)	5,82	16,0	70

12	(30,0; 40,4)	6,00	14,0	100
13	(14,3; 28,7)	6,35	16,0	70
14	(17,5; 26,4)	6,09	16,0	70
15	(22,4; 22,3)	5,99	13,0	80
16	(32,2; 17,4)	6,00	11,0	50
17	(41,5; 27,2)	5,96	8,0	90
18	(35,4; 19,5)	6,00	10,0	50
19	(20,0; 17,5)	5,69	16,0	70
20	(18,4; 20,5)	5,96	16,0	70

Para o KNN, os resultados podem ser vistos na tabela Tabela 14, que indica quantos resultados foram obtidos como sendo os perfeitos e quantos resultados não se verificam na tabela dos melhores *setups*.

Tabela 14 – Comparação dos resultados previstos com os resultados obtidos do KNN

	Nº DE SETUPS	**PERCENTAGEM DE SUCESSO (%)**
RESULTADOS POSITIVOS	5	25
RESULTADOS NEGATIVOS	15	75

Verificando a Tabela 12 e a Tabela 14, em 20 *queries* o KNN encontrou corretamente 5 *setups* perfeitos, possuindo uma taxa de sucesso de 25%.

No entanto, este problema possibilita alguma flexibilidade nos valores retornados pelos algoritmos. Devido a isto pode-se afirmar que os 15 resultados negativos não estão totalmente errados. Um exemplo deste aspeto é o caso da *query* 5, onde o algoritmo devolve uma eficiência igual a 5.98 para um *setup* LWT=12,0 e *load*=40, e o *setup* que deveria devolver seria LWT=10,0 e *load*=40% com uma eficiência de 6.0. Comparando estes valores, pode-se afirmar que não são muito diferentes, apenas a LWT apresenta um valor menor que o *setup* perfeito, e consequentemente, uma diferença de eficiência muito pouco significativa.

Pelo que foi explicado anteriormente, o facto de o algoritmo apresentar uma precisão de 25% em 20 *queries* é bastante satisfatório, não esquecendo que a razão dos 75% de resultados negativos deve-se pelo facto de não correspondem aos *setups* perfeitos. No entanto podem corresponder a *setups* muito parecidos e igualmente eficientes.

No caso do *K-Means Clustering*, a Tabela 15 mostra os resultados obtidos.

Tabela 15 - Comparação dos resultados previstos com os resultados obtidos no K-Means Clustering

	Nº DE SETUPS	**PERCENTAGEM DE SUCESSO (%)**
RESULTADOS POSITIVOS	6	30
RESULTADOS NEGATIVOS	14	70

Para o *K-Means Clustering* verifica-se uma taxa de sucesso de 30%, que corresponde a 6 *setups* corretos, em 20 *queries*. No entanto, como foi referido para o KNN, o facto de o algoritmo não acertar no *setup* perfeito, existem muitos outros *setups* que tornam o funcionamento dos *chillers* mais eficiente que o funcionamento atual, embora o valor de eficiência seja ligeiramente menor. Como exemplo, na *query* 7 é desejado um *setup* com LWT=10.0ºC e *load*= 50%, que corresponde a uma eficiência de 6,0. O algoritmo retornou um de *setup* com LWT=11,0ºC e *load*=50%, apresentando uma eficiência de 5,96. Verificando este exemplo, é possível afirmar que o *setup* devolvido pelo algoritmo é bastante razoável para combater a *query* recebida, embora não seja o *setup* perfeito.

5.4.3. Matriz de Confusão

Para fazer uma comparação à qualidade dos algoritmos foram realizadas matrizes de confusão para cada algoritmo. Uma matriz de confusão é normalmente usada para descrever a *performance* de um modelo de classificação para um determinado *training data* onde os valores corretos são conhecidos. Transpondo para este caso, a matriz de confusão será importante para determinar a eficiência dos dois algoritmos, comparado os resultados de cada um com o melhor resultado possível existente no *training data*.

Antes de se observar as matrizes, é importante perceber o seu funcionamento e benefícios. Para isto, observe-se um exemplo de matriz de confusão para um classificador de 2 classes. Este classificador, quando recebe uma *query* classifica com 0 ou 1. Então, a matriz de confusão terá o seguinte aspeto (Figura 48):

		Obtido	
		0	1
Previsto	0	A	B
	1	C	D

Figura 48 - Exemplo de uma matriz de confusão para classificador de 2 classes

Uma matriz de confusão possui informações sobre as classificações obtidas e previstas, realizadas por um processo de classificação (algoritmo KNN ou *K-Means Clustering* por exemplo). O desempenho destes processos é avaliado usando os dados da matriz de confusão. Os *inputs* da matriz têm os seguintes significados:

- A – Número de resultados previstos para a classe 0;
- B – Número de resultados não previstos para a classe 0;
- C – Número de resultados não previstos para a classe 1;
- D – Numero de resultados previstos para a classe 1.

Como se observa, a diagonal apresenta os valores corretos, isto é, que foram previstos e obtidos (A e D). As restantes células apresentam valores falhados (C e B). Com estes números é possível calcular a precisão e *recall* de cada *query*, bem como a precisão total e *recall* total referente ao algoritmo ML:

$$Precisão_{(0)} = \frac{A}{A+B};\ Precisão_{(1)} = \frac{D}{C+D}$$

$$Recall_{(0)} = \frac{A}{A+C};\ Recall_{(1)} = \frac{D}{B+D}$$

$$Precisão\ total = \ Média\ das\ precisões\ de\ cada\ classes = \frac{Precisão_{(0)} + Precisão_{(1)}}{2}$$

$$Recall\ total = \ Média\ dos\ recalls\ de\ cada\ classes = \frac{Recall_{(0)} + Recall_{(1)}}{2}$$

Precisão consiste na fração dos *setups* obtidos e que se esperam obter enquanto que o *recall* é a fração de *setups* que se espera obter e que foram obtidos com sucesso. Já a precisão total é a média dos valores de precisão para cada classe e assim para o *recall* total.

Transpondo esta explicação para o caso de estudo, para construir as matrizes dos algoritmos implementados, foram geradas aleatoriamente 45 *queries* e processadas 50 vezes pelos algoritmos, resultado num total de 2250 processamentos. Estes parâmetros foram escolhidos

tendo em conta a visibilidade da matriz e a qualidade da informação que se poderia tirar dela. Uma matriz completa para este caso resultaria em mais de 10000 colunas e 10000 linhas, algo que seria bastante difícil de tirar conclusões. Assim, encurta-se a matriz para 45 classes, processando 45 *queries* 50 vezes, o que permite visualizar com facilidade a qualidade do algoritmo. Esta explicação pode ser observada nos subcapítulos 5.4.3.1 e 5.4.3.2.

5.4.3.1. KNN

Neste capítulo apresenta-se a matriz de confusão referente ao KNN. Como foi mencionado, foram processadas 45 *queries*, cada uma 50 vezes.

Na Figura 49 podemos observar a matriz, onde os *inputs* são os resultados obtidos e os resultados previstos. Os valores correspondentes ao fundo branco são os resultados falhados pelo algoritmo, significando que o *setup* retornado pelo KNN está errado. Na diagonal (fundo verde) encontram-se o número de vezes que o algoritmo acertou no *setup*. Na coluna da direita é possível ver o valor da precisão que o algoritmo apresentou para essa *query* da linha correspondente. Na linha de baixo encontra-se o valor do *recall* para cada coluna de valores recebidos.

Como exemplo, observe-se a primeira linha dos resultados da matriz. Esta indica que para essa *query*, o *setup* esperado seria de *Eff.* (eficiência)=6.00 para um LWT=7º e LOAD=30%. A linha diz que, em 50 vezes que processou a *query* em questão, o algoritmo acertou 2 vezes no resultado esperado. Os restantes *setups* foram todos diferentes do previsto. Para esta *query*, o algoritmo obteve uma precisão de 4% (0.04) nos 50 processamentos.

A precisão total do algoritmo pode ser calculada, como mencionado, pela média das várias precisões encontradas na coluna da direita:

$$Precisão\ total[\%]_{KNN} = \frac{\sum(valores\ de\ precisão)}{45} * 100 = 18{,}9\%,$$

$$sendo\ que\ o\ Erro = 1 - Precisão\ total[\%]_{KNN} * 100 = 81{,}1\%$$

Assim como o *recall*:

$$Recall\ total[\%]_{KNN} = \frac{\sum(valores\ de\ recall)}{45} * 100 = 16{,}7\%$$

Apenas analisando os resultados calculados, pode-se afirmar que o algoritmo não teve muito sucesso. Uma precisão de 18.9% é um valor que fica muito aquém do que se pretende num algoritmo ML. O mesmo de refere ao *recall*, pois quanto mais elevando melhor.

Setups Obtidos

Setups Previstos

Eff.			6,00	5,02	5,40	5,64	6,00	5,92	5,98	6,00	6,00	6,00	6,00	5,98	5,97	6,08	5,96	5,98	6,00	6,00	6,00	6,00	6,00	6,00	6,00	6,00	6,00	6,22	5,70	5,82	5,96	6,20	6,22	6,33	6,35	5,99	6,00	6,03	5,95	5,96	6,00	5,89	5,99	5,99	5,93	6,00	5,98	Precisão
	LWT(ºC)		7	11	12	12	14	15	16	8	9	10	12	13	15	15	16	16	7	12	13	7	10	16	10	13	14	15	16	16	16	16	16	16	16	15	15	16	8	8	10	13	14	11	14	14	16	
		LOAD (%)	30	30	30	30	30	30	30	40	40	40	40	40	40	40	40	40	50	50	50	60	60	60	70	70	70	70	70	70	70	70	70	70	70	80	80	80	90	90	90	90	90	100	100	100	100	
6,00	7	30	2	0	0	0	0	0	0	39	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,04
5,02	11	30	0	6	40	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,12
5,40	12	30	0	0	0	2	0	0	0	0	0	0	48	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,00
5,64	12	30	0	0	0	50	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1,00
6,00	14	30	0	0	0	0	0	0	0	0	0	0	4	46	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,00
5,92	15	30	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,00
5,98	16	30	0	0	0	0	0	50	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,00
6,00	8	40	47	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,06
6,00	9	40	0	0	0	0	0	0	0	0	50	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1,00
6,00	10	40	0	35	0	0	0	0	0	0	10	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,10
6,00	12	40	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,00
5,98	13	40	0	0	0	50	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,00
5,97	15	40	0	0	0	0	0	0	0	0	0	0	0	0	0	50	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,00
6,08	15	40	0	0	0	0	0	0	0	0	0	0	0	0	0	50	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1,00
5,96	16	40	0	0	0	0	0	45	0	0	0	0	0	0	0	0	3	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,06
5,98	16	40	0	0	0	0	0	0	0	0	0	0	0	0	49	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,00
6,00	7	50	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	50	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1,00
6,00	12	50	0	0	0	0	0	0	0	0	0	0	0	0	0	50	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,00
6,00	13	50	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,00
6,00	7	60	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	42	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	8	0	0	0	0	0	0	0	0	0,00
6,00	10	60	0	0	0	0	0	0	0	0	0	0	50	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,00
6,00	16	60	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	48	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0,96
6,00	10	70	0	0	0	0	0	0	0	0	0	0	0	0	50	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,00
6,00	13	70	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	49	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,00
6,00	14	70	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	50	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1,00
6,22	15	70	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	0	0	0	0	0	45	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,00
5,70	16	70	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,00
5,82	16	70	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	47	0	0	0	0	0	0	0	0	0	0	0	0	0	0,06
5,96	16	70	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	50	0	0	0	0	0	0	0	0	0	0	0	0	0	0,00
6,20	16	70	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,00
6,22	16	70	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,00
6,33	16	70	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	50	0	0	0	0	0	0	0	0	0	0	0	0	0	1,00
6,35	16	70	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,00
5,99	15	80	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	45	0	0	0	0	0	0	0	0	0,00
6,00	15	80	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	43	7	0	0	0	0	0	0	0	0	0	0	0,00
6,03	16	80	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	50	0	0,00
5,95	8	90	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	50	0	0	0	0	0	0	0	0	1,00
5,96	8	90	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	50	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,00
6,00	10	90	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	49	1	0	0	0	0	0	0	0	0	0	0	0,00
5,89	13	90	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	50	0	0	0	0,00
5,99	14	90	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	50	0	0	0,00
5,99	11	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	45	0	0	0	0	0	0	5	0	0	0	0,10
5,93	14	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	50	0	0	0	0	0	0	0	0	0	0,00
6,00	14	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	50	0	0	0	0	0	0	0	0	0	0,00
5,98	16	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	19	31	0	0	0	0	0	0	0	0	0,00
Recall			0,04	0,15	0,00	0,47	0,00	0,00	0,00	0,07	0,72	1,00	0,00	0,00	0,00	0,33	0,75	0,00	0,54	0,00	0,00	0,00	0,00	0,52	0,00	0,00	1,00	0,00	0,00	1,00	0,00	0,00	0,00	0,34	0,00	0,00	0,00	0,00	0,49	0,00	0,00	0,00	0,00	0,09	0,00	0,00	0,00	

Figura 49 - Matriz de confusão para o KNN

No entanto, olhando para os resultados que a matriz apresenta, é possível tirar outras observações interessantes.

Ao observar individualmente cada *query* e olhando para o valor da diagonal, nota-se que o KNN conseguiu acertar em 14 *queries* diferentes das 45 processadas. Para algumas acertou nos 50 processamentos enquanto que nas outras acertou menos vezes. Para que o algoritmo obtivesse 100% de precisão, a diagonal deveria estar preenchida com 50, significando que tinha sempre acertado em todos os processamentos. É possível observar todos estes comportamentos na matriz. Nos casos em que acerta apenas algumas vez (por exemplo, logo na primeira *query*) deve-se provavelmente ao treino que o algoritmo vai sofrendo ao longo dos 2250 processamentos que executa. Nos casos onde o KNN falha em todos os processamentos, pode-se afirmar que, embora tenha falhado, os *setups* devolvidos nalguns casos são bastante agradáveis (teoria explicada no capítulo 5.4.2).

Visto isto, embora o cálculo da precisão total seja um indicador bastante importante que se retira da matriz de confusão e que caracteriza o algoritmo ML, é importante perceber o que acontece por dentro de cada processamento. Há *setups* com variações de valores tão significativas dos *setups* perfeitos que deveriam ser considerados como certos, algo que neste tipo de testes é impossível configurar.

5.4.3.2. *K-Means Clustering*

Neste capítulo apresenta-se a matriz de confusão referente ao *K-Means Clustering*. Como foi mencionado, foram processadas 45 *queries*, cada uma 50 vezes.

Na Figura 50 é possível observar a matriz, que possui a mesma estrutura que a matriz de confusão do KNN:

- Em cima encontram-se os resultados obtidos, à esquerda os resultados esperados;
- Diagonal (fundo verde) apresenta os resultados corretos;
- Os restantes valores (fundo branco) são os resultados falhados;
- Na coluna da direita encontra-se as precisões para cada *query* assim como na linha de baixo se encontra o *recall*.

Explicada a estrutura da matriz no capítulo 5.4.3.1, procedeu-se ao cálculo da precisão total e *recall* pelo mesmo método utilizado para a matriz do algoritmo KNN:

$$Precisão\ total[\%]_{K-Means\ Clustering} = \frac{\sum(valores\ de\ precisão)}{45} = 17{,}7\%$$

$$Recall\ total[\%]_{K-Means\ Clustering} = \frac{\sum(valores\ de\ recall)}{45} = 13{,}8\%$$

Sendo o erro:

$$Erro = 1 - Precisão\ total[\%]_{K-Means\ Clustering} * 100 = 82{,}3\%$$

Tal como aconteceu com o KNN, os resultados do *K-Means Clustering* também não foram os esperados. Olhando apenas para os resultados, uma precisão de 17,7% e um *recall* de 13.8% são valores baixos, no entanto, e assim como ocorreu no KNN, é preciso analisar os valores que a matriz apresenta. Observando a diagonal com fundo verde, pode-se afirmar que o *K-Means Clustering* acertou em 8 dos valores previstos, nas 45 *queries*, independentemente se acertou nas 50 vezes que cada uma foi processada ou não. Esta parte é importante referir, pois como se trata de um algoritmo ML, ele vai sendo treinado ao longo do seu funcionamento e, portanto, os *setups* devolvidos poderão alterar. Com isto, pode-se concluir que em 50 processamentos para uma *query* o algoritmo pode devolver valores diferentes, e assim pode-se concluir que o algoritmo poderá estar a "aprender".

Tal como no KNN, também se pode afirmar que muitos dos resultados obtidos poderão ser igualmente eficientes como os resultados previstos. Os *setups* previstos são bastante específicos, tornando-se bastante difícil de os obter com o *K-Means Clustering*, visto que este classifica as *queries* com um número de *centroids* escolhido pelo programador (neste caso 87 classes). No entanto, o que se procura é que o *K-Means Clustering* procure o *centroid* mais próximo da *query*, podendo este ser considerado um dos *setups* previstos ou um *setup* bastante semelhante ao previsto. Por esta razão, e através da análise dos *setups* retornados pelo *K-Means Clustering*, confirma-se o seu bom funcionamento no tratamento de *queries*

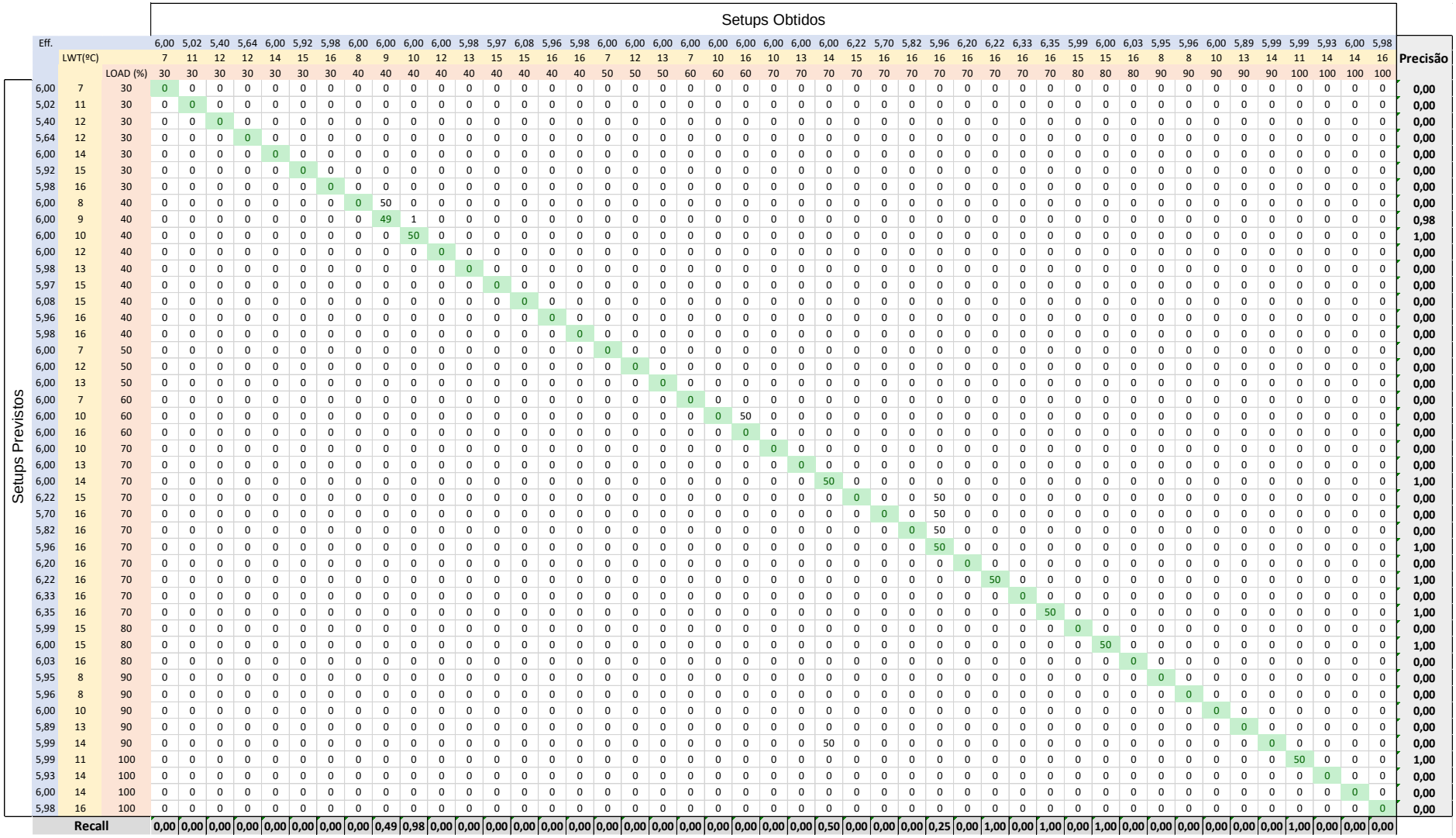


Figura 50 - Matriz de confusão do K-Means Clustering

5.4.3.3. Resumo dos resultados obtidos nas matrizes de confusão

Depois de se comparar as duas matrizes, apresenta-se aqui um resumo dos valores obtidos (precisão, *recall* e erros), bem como outros valores que são observados nas matrizes.

Tabela 16 - Comparação dos resultados das Matrizes de Confusão

	KNN	K-MEANS CLUSTERING
PRECISÃO TOTAL [%]	18,9	17,7
RECALL TOTAL [%]	16,7	13,8
ERRO [%]	81,1	82,3
Nº DE DIFERENTES QUERIES ACERTADAS*	14	8

*Independentemente do número de vezes que acertou. Caso o algoritmo acerte 1 em 50 vezes, considerou-se como uma *query* acertada.

Como explicado ao longo das análises das matrizes para os diferentes algoritmos, embora estes parâmetros sejam bons indicadores sobre a qualidade do algoritmo, é preciso observar outros fatores bastante importantes. Um destes fatores é investigar a qualidade dos *setups* errados, como já foi explicado diversas vezes. Muitos dos *setups* considerados como errados poderão ser na realidade *setups* suficientemente bons para serem escolhidos. As diferenças entre estes *setups* e os previstos são mínimas, e alguns até são iguais.

Deste modo, afirma-se que as conclusões tiradas deste teste coincidem com as retiradas do capítulo 5.4.2.

Concluindo, a matriz de confusão realizada para cada algoritmo permitiu obter uma outra visão sobre os resultados de cada algoritmo. Analisar um processamento de 2250 *queries* não seria fácil sem esta forma de qualificar a qualidade de algoritmos ML.

6

Conclusões e Trabalho Futuro

6.1. Conclusões Gerais

Num exemplo assustador, a melhoria de, por exemplo, 0.1% no uso de energia poderá poupar milhares nos custos para os proprietários de DCs. A energia, especialmente usada na refrigeração, tornou-se um problema sério para os DCs nos dias de hoje. Através do trabalho desenvolvido foi possível demonstrar que a arquitetura apresentada é uma solução válida para este enorme problema que afeta os DCs. Os *chillers*, máquinas responsáveis pela refrigeração das salas dos servidores, combinados com algoritmos *machine learning* que lhes fornece uma inteligência artificial para combater diferentes situações que se apresentam, podem tornar-se numa alternativa muito eficiente no combate à deficiente gestão energética.

Os resultados obtidos dos algoritmos implementos foram os esperados. Algumas falhas acabaram por acontecer como foi mencionado anteriormente, devido a vários fatores como os mencionados na página 64. No entanto, foram encontrados resultados bastante interessantes para para afirmar que este caso de estudo merecia um trabalho futuro mais sólido e realista.

De notar que a implementação de algoritmos *machine learning* nem sempre é fácil, como foi o caso nesta solução proposta. A sua implementação é complexa e feita por etapas que requerem sempre várias validações para que se prossiga para o próximo passo. Existem vários algoritmos implementados que podem ser adaptados a diferentes situações. No entanto, alguns algoritmos não se adequam para certos casos. Essa foi uma das dificuldades encontradas para o desenvolvimento da solução. A procura de algoritmos ML que se adequassem ao caso de estudo foi uma tarefa de algum trabalho, pois alguns foram inicialmente implementados até se perceber que não era o procurado. O *training data* obtido é disperso e não linear, então foram procurados algoritmos que se adequassem para estas características. O KNN e o *K-Means Clustering* foram os algoritmos escolhidos, sendo o KNN talvez um dos mais simples para este tipo de *training data*. A implementação do *K-Means Clustering* apresentou algumas dificuldades que foram ultrapassadas com sucesso. Para compensar a complexidade da sua implementação, o algoritmo é bastante eficaz no processamento das *queries* recebidas, apresentando um menor tempo de processamento em relação ao KNN e uma boa qualidade nos resultados adquiridos.

Ambos os algoritmos implementados são boas soluções para o problema apresentado no caso de estudo. Para melhorar a qualidade dos resultados seria necessário um *training data* composto com dados reais de um *chiller* em funcionamento. Embora o TOPSS tivesse fornecido dados bastante razoáveis para a criação de um simulador, dados reais são sempre mais fidedignos e geram menos falhas dos resultados desenvolvidos.

Conclui-se que esta iniciativa pode trazer grandes avanços nos sistemas de refrigeração e no funcionamento dos *chillers*. É possível que traga melhorias na eficiência energética que, em consequência, diminuirá drasticamente os custos para as organizações proprietárias de DCs. Os métodos de *machine learning* são cada vez mais usados e constituem uma boa ferramenta para a resolução de casos parecidos a este apresentado.

6.2. Trabalho Futuro

Como trabalho futuro, existem vários processos que podem ser pensados para chegar à melhor solução e assim ser possível testar num ambiente realista. A solução proposta peca por apenas ser possível testa-la através da criação de um simulador. Por isto, o objetivo principal do trabalho futuro será testar os métodos de *machine learning* criados num ambiente real e assim testar se são verdadeiramente opções a utilizar.

Para aperfeiçoamento inicial, o trabalho futuro deve começar por retirar dados reais dos *chillers*, que se encontram muito possivelmente em bases de dados SQL. Estas bases de dados

estão constantemente a receber dados sobre vários fatores correspondentes à sala do DC e ao funcionamento dos *chillers*.

Então, será preciso obter dados anteriores para constituir um *training data* robusto e de qualidade e estar conectado em tempo real à base de dados SQL para receber os novos dados. Estes dados deverão ser lidos das tabelas SQL e colocados em memória da máquina que os irá processar. A utilização de uma *framework* de distribuição também é uma boa hipótese para melhorar a eficiência do algoritmo, sendo o *Apache Flink* o mais indicado por possuir características de *batch* e *stream* em tempo real, bem como uma boa biblioteca ML.

A *query* será sempre composta por uma temperatura EWT e uma temperatura ambiente. Como referido, a EWT é fornecida pela leitura dos sensores dos *chillers* e guardada nas bases de dados. A temperatura ambiente, neste caso de estudo, foi imposta, sendo testada para vários valores. Num sistema real, esta temperatura deverá ser lida de uma base de dados oficial e assim ser retirada a temperatura atual da localização do DC em causa.

A implementação de mais algoritmos ML é também necessária para que haja uma comparação mais abrangente destas tecnologias. Com isto, é possível encontrar a melhor solução possível ou até mesmo implementar um método híbrido que junte as melhores funcionalidades de cada algoritmo. O mais indicado seria criar um algoritmo ML específico para este caso, apenas para tratar o uso ineficiente dos *chillers*.

Os testes realizados aos algoritmos foram meramente virtuais, sendo totalmente impossível afirmar que se tratam de métodos com viabilidade, embora os resultados tenham tido alguma qualidade. Num trabalho futuro, deve-se testar com uma maior quantidade de *queries*.

Para testar verdadeiramente os algoritmos é necessário que estejam ligados aos controlos dos *chillers* e assim testar nesses equipamentos. Uma vez lida a *query* em tempo real da base de dados, o algoritmo devolve um *setup* que atuará no *chiller*. Esse *setup*, como mencionado várias vezes neste estudo, irá colocar o *chiller* a funcionar da maneira mais eficiente (energicamente falando) para responder à *query* recebida. Este teste necessitará de decorrer por um longo período de tempo, até ao momento de ser possível verificar se foi poupada energia ou não face aos períodos de tempo anteriores, quando o *chiller ainda* funcionava igualmente para qualquer tipo de situação. Caso exista poupança na energia despendida poderá afirmar-se que se trata de uma solução viável capaz de combater o grande problema apresentado nos DCs.

Este deverá ser o caminho do trabalho futuro. É importante que estas situações sejam estudadas, pois não são só os proprietários de DCs que ganham em ter um sistema mais eficiente. Uma menor quantidade de energia utilizada significa também menor poluição do meio ambiente.

Bibliografia

Air-Cooled Screw Chillers RTAF 300-1900 kW Trane Sintesis. (2014), 1–28. Retrieved from http://commercial.trane.com/content/dam/Trane/europe-tour/english/RTAF-SLB001-E4_0814.pdf

Al-Fares, M., Radhakrishnan, S., & Raghavan, B. (2010). Hedera: Dynamic Flow Scheduling for Data Center Networks. *Nsdi*, 19. Retrieved from http://dl.acm.org/citation.cfm?id=1855730%5Cnhttps://www.usenix.org/legacy/event/nsdi10/tech/full_papers/al-fares.pdf

Atzori, L., Iera, A., & Morabito, G. (2016). Understanding the Internet of Things: definition, potentials, and societal role of a fast evolving paradigm. *Ad Hoc Networks*, *56*, 122–140. https://doi.org/10.1016/j.adhoc.2016.12.004

Bajaber, F., Elshawi, R., Batarfi, O., Altalhi, A., Barnawi, A., & Sakr, S. (2016). Big Data 2.0 Processing Systems: Taxonomy and Open Challenges. *Journal of Grid Computing*, *14*(3), 379–405. https://doi.org/10.1007/s10723-016-9371-1

Bengio, Y., Courville, A., & Vincent, P. (2013). Representation Learning: A Review and New Perspectives. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, *35*(8), 1798–1828. https://doi.org/10.1109/TPAMI.2013.50

Berral, J. L., Goiri, Í., Nou, R., Julià, F., Guitart, J., Gavaldà, R., & Torres, J. (2010). Towards energy-aware scheduling in data centers using machine learning. *Proceedings of the 1st*

International Conference on EnergyEfficient Computing and Networking eEnergy 10, *2*, 215. https://doi.org/10.1145/1791314.1791349

Bez, J. (2015). Plataformas de Big Data: Spark, Storm e Flink. https://doi.org/10.13140/RG.2.1.3147.1209

Borthakur, D., Rash, S., Schmidt, R., Aiyer, A., Gray, J., Sarma, J. Sen, … Menon, A. (2011). Apache hadoop goes realtime at Facebook. *SIGMOD '11 - Proceedings of the 2011 International Conference on Management of Data*. https://doi.org/10.1145/1989323.1989438

Carbone, P., Ewen, S., Haridi, S., Katsifodimos, A., Markl, V., & Tzoumas, K. (2015a). Apache Flink: Unified Stream and Batch Processing in a Single Engine. *Data Engineering*, *36*(4), 28–38. Retrieved from http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-198940

Carbone, P., Ewen, S., Haridi, S., Katsifodimos, A., Markl, V., & Tzoumas, K. (2015b). Apache Flink: Unified Stream and Batch Processing in a Single Engine. *Data Engineering*, *36*(4), 28–38. Retrieved from http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-198940

Cattell, R. (2011). Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, *39*(4), 12. https://doi.org/10.1145/1978915.1978919

Com, R. M. (2010). Web-Scale Bayesian Click-Through Rate Prediction for Sponsored Search Advertising in Microsoft ' s Bing Search Engine. *Search*, (April 2009), 13–20. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.165.5644&rep=rep1&type=pdf

Farooq, M. U., & Waseem, M. (2015). A Review on Internet of Things (IoT). *International Journal of Computer Applications (0975 8887)*, *113*(1), 1–7. https://doi.org/10.5120/19787-1571

Fergus, P., Hussain, A. J., Hearty, J., Fairclough, S., Boddy, L., Mackintosh, K., … Lunn, J. (2017). A machine learning approach to measure and monitor physical activity in children. *Neurocomputing*, *228*, 220–230. https://doi.org/10.1016/j.neucom.2016.10.040

Frank, M. R., Omiecinski, E. R., & Navathe, S. B. (1992). Adaptive and Automated Index Selection in Rdbms. *Advances in Database Technology - Edbt 92*, *580*, 277–292.

Gandhi, A., Harchol-Balter, M., Raghunathan, R., & Kozuch, M. A. (2012). AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers. *ACM Trans. Comput. Syst*, *30*(26). https://doi.org/10.1145/2382553.2382556

Gantz, J., & Reinsel, D. (2012). THE DIGITAL UNIVERSE IN 2020: Big Data, Bigger Digi tal Shadows, and Biggest Growth in the Far East. *Idc*, *2007*(December 2012), 1–16.

Gao, J., & Jamidar, R. (2014). Machine Learning Applications for Data Center Optimization. *Google White Paper*, 1–13.

Gentz, M., Sunny, D., & Lucas. (2016). Quando usar NoSQL versus SQL | Microsoft Docs. Retrieved January 30, 2017, from https://docs.microsoft.com/pt-br/azure/documentdb/documentdb-nosql-vs-sql

Hall, P., Park, B. U., & Samworth, R. J. (2008). Choice of neighbor order in nearest-neighbor classification. *Annals of Statistics*, *36*(5), 2135–2152. https://doi.org/10.1214/07-AOS537

Hung, N. T., Giang, D. H., Keong, N. W., & Zhu, H. (2012). Cloud-enabled data sharing model. *ISI 2012 - 2012 IEEE International Conference on Intelligence and Security Informatics: Cyberspace, Border, and Immigration Securities*, 1–6. https://doi.org/10.1109/ISI.2012.6281922

Inoubli, W., Aridhi, S., Mezni, H., & Jung, A. (2016). Big Data Frameworks: A Comparative Study. Retrieved from http://arxiv.org/abs/1610.09962

Iqbal, M. H., & Soomro, T. R. (2015). Big Data Analysis: Apache Storm Perspective. *International Journal of Computer Trends and Technology (IJCTT)*, *19*(1), 9–14. https://doi.org/10.14445/22312803/IJCTT-V19P103

Josyula, V., Orr, M., Page, G., & Press, C. (2011). Cloud Computing: Automating the Virtualized Data Center Cloud Computing: Automating the Virtualized Data Center Warning and Disclaimer ii Cloud Computing: Automating the Virtualized Data Center.

Junghanns, M., Petermann, A., Teichmann, N., Gómez, K., & Rahm, E. (2016). Analyzing extended property graphs with Apache Flink. In *1st ACM SIGMOD Workshop on Network Data Analytics* (pp. 1–8). https://doi.org/10.1145/2980523.2980527

Kavakiotis, I., Tsave, O., Salifoglou, A., Maglaveras, N., Vlahavas, I., & Chouvarda, I. (2017). Machine Learning and Data Mining Methods in Diabetes Research. *Computational and Structural Biotechnology Journal*, *15*, 104–116. https://doi.org/10.1016/j.csbj.2016.12.005

Kodinariya, T. M., & Makwana, P. R. (2013). Review on determining number of Cluster in K-Means Clustering. *International Journal of Advance Research in Computer Science and Management Studies*, *1*(6), 2321–7782. Retrieved from http://www.ijarcsms.com/docs/paper/volume1/issue6/V1I6-0015.pdf

Kotsiantis, S. B. (2007). Supervised machine learning: A review of classification techniques. *Informatica*, *31*, 249–268. https://doi.org/10.1115/1.1559160

Kumar, R., Gupta, N., Charu, S., Bansal, S., & Yadav, K. (2014). Comparison of SQL with HiveQL. *International Journal for Research in Technological Studies*|, *1*(9online), 2348–1439.

Kumar, R., Gupta, N., Charu, S., & Jangir, S. K. (2014). Architectural Paradigms of Big Data. *National Conference on Innovation in Wireless Communication and Networking Technology*, (APRIL), 1–5. https://doi.org/10.13140/2.1.2392.5123

Kune, R., Konugurthi, P. K., Agarwal, A., Chillarige, R. R., & Buyya, R. (2011). The Anatomy of Big Data Computing.

Li, Y., & Manoharan, S. (2013). A performance comparison of SQL and NoSQL databases. In *IEEE Pacific RIM Conference on Communications, Computers, and Signal Processing - Proceedings* (pp. 15–19). https://doi.org/10.1109/PACRIM.2013.6625441

Moniruzzaman, A. B. M., & Hossain, S. A. (2013). Nosql database: New era of databases for big data analytics-classification, characteristics and comparison. *Nosql Database: New Era of Databases for Big Data Analytics-Classification, Characteristics and Comparison*, *6*(4), 1–14. Retrieved from http://scholar.google.com/scholar?q=Nosql database: New era of databases for big data analytics-classification, characteristics and comparison&btnG=&hl=en&num=20&as_sdt=0%2C22 VN - readcube.com

Naji, S., Keivani, A., Shamshirband, S., Alengaram, U. J., Jumaat, M. Z., Mansor, Z., & Lee, M.

(2016). Estimating building energy consumption using extreme learning machine method. *Energy*, *97*, 506–516. https://doi.org/10.1016/j.energy.2015.11.037

Özköse, H., Arı, E. S., & Gencer, C. (2015). Yesterday, Today and Tomorrow of Big Data. *Procedia - Social and Behavioral Sciences*, *195*, 1042–1050. https://doi.org/10.1016/j.sbspro.2015.06.147

Pääkkönen, P., & Pakkala, D. (2015). Reference Architecture and Classification of Technologies, Products and Services for Big Data Systems. *Big Data Research*. https://doi.org/10.1016/j.bdr.2015.01.001

Panduit Corporation. (2013). The Cloud-Enabled Data Center.

Rana, N., & Deshmukh, S. (2015). Shuffle Performance in Apache Spark, *4*(2), 177–180.

Rong, H., Zhang, H., Xiao, S., Li, C., & Hu, C. (2016). Optimizing energy consumption for data centers. *Renewable and Sustainable Energy Reviews*, *58*, 674–691. https://doi.org/10.1016/j.rser.2015.12.283

Shoro, A. G., & Soomro, T. R. (2015). Big Data Analysis: Apache Spark Perspective. *Global Journal of Computer Science and Technology*, *15*(1).

Shute, J., Vingralek, R., Samwel, B., Handy, B., Whipkey, C., Rollins, E., … Apte, H. (2013). F1: A Distributed SQL Database That Scales.

Snášel, V., Nowaková, J., Xhafa, F., & Barolli, L. (2017). Geometrical and topological approaches to Big Data. *Future Generation Computer Systems*, *67*, 286–296. https://doi.org/10.1016/j.future.2016.06.005

Song, Z., Zhang, X., & Eriksson, C. (2015). Data Center Energy and Cost Saving Evaluation. In *Energy Procedia* (Vol. 75, pp. 1255–1260). https://doi.org/10.1016/j.egypro.2015.07.178

Uddin, M., Darabidarabkhani, Y., Shah, A., & Memon, J. (2015). Evaluating power efficient algorithms for efficiency and carbon emissions in cloud data centers: A review. *Renewable and Sustainable Energy Reviews*, *51*, 1553–1563. https://doi.org/10.1016/j.rser.2015.07.061

van der Veen, J. S., van der Waaij, B., & Meijer, R. J. (2012). Sensor Data Storage Performance: SQL or NoSQL, Physical or Virtual. *2012 IEEE Fifth International Conference on Cloud Computing*, 431–438. https://doi.org/10.1109/CLOUD.2012.18

Vicent, J. (2016). Google uses DeepMind AI to cut data center energy bills - The Verge. Retrieved January 30, 2017, from http://www.theverge.com/2016/7/21/12246258/google-deepmind-ai-data-center-cooling

Vilaça, R., Cruz, F., Pereira, J., & Oliveira, R. (2013). An effective scalable SQL engine for NoSQL databases. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (Vol. 7891 LNCS, pp. 155–168). https://doi.org/10.1007/978-3-642-38541-4_12

Voyant, C., Notton, G., Kalogirou, S., Nivet, M.-L., Paoli, C., Motte, F., & Fouilloy, A. (2017). Machine Learning methods for solar radiation forecasting: a review. *Renewable Energy*, *105*, 569–582. https://doi.org/10.1016/j.renene.2016.12.095

Wang, H., Zaniolo, C., & Luo, C. R. (2003). ATLAS: A Small but Complete {SQL} Extension for Data Mining and Data Streams. In *Proceedings 2003 {VLDB} Conference* (pp. 1113–

1116). https://doi.org/http://dx.doi.org/10.1016/B978-012722442-8/50118-X

Xiao, L., & Wang, Z. (2011). Internet of Things: a New Application for Intelligent Traffic Monitoring System. *Journal of Networks*, *6*(6), 887–894. https://doi.org/10.4304/jnw.6.6.887-894

Yazici, A., Alayyoub, M., & Karakaya, Z. (2016). A Systematic Mapping Study for Big Data Stream Processing Frameworks.